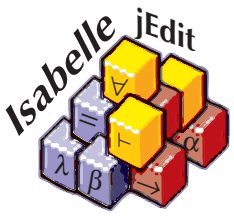


Isabelle/jEdit

Makarius Wenzel

12 December 2016

Abstract

Isabelle/jEdit is a fully-featured Prover IDE, based on Isabelle/Scala and the jEdit text editor. This document provides an overview of general principles and its main IDE functionality.

> *Isabelle's user interface is no advance over LCF's, which is widely condemned as "user-unfriendly": hard to use, bewildering to beginners. Hence the interest in proof editors, where a proof can be constructed and modified rule-by-rule using windows, mouse, and menus. But Edinburgh LCF was invented because real proofs require millions of inferences. Sophisticated tools — rules, tactics and tacticals, the language ML, the logics themselves — are hard to learn, yet they are essential. We may demand a mouse, but we need better education and training.*
>
> Lawrence C. Paulson, "Isabelle: The Next 700 Theorem Provers"

Acknowledgements

Research and implementation of concepts around PIDE and Isabelle/jEdit has started in 2008 and was kindly supported by:

- TU München http://www.in.tum.de
- BMBF http://www.bmbf.de
- Université Paris-Sud http://www.u-psud.fr
- Digiteo http://www.digiteo.fr
- ANR http://www.agence-nationale-recherche.fr

Contents

List of Figures

Chapter 1

Introduction

1.1 Concepts and terminology

Isabelle/jEdit is a Prover IDE that integrates *parallel proof checking* [6, 11] with *asynchronous user interaction* [7, 10, 12, 13], based on a document-oriented approach to *continuous proof processing* [8, 9]. Many concepts and system components are fit together in order to make this work. The main building blocks are as follows.

Isabelle/ML is the implementation and extension language of Isabelle, see also [4]. It is integrated into the logical context of Isabelle/Isar and allows to manipulate logical entities directly. Arbitrary add-on tools may be implemented for object-logics such as Isabelle/HOL.

Isabelle/Scala is the system programming language of Isabelle. It extends the pure logical environment of Isabelle/ML towards the outer world of graphical user interfaces, text editors, IDE frameworks, web services etc. Special infrastructure allows to transfer algebraic datatypes and formatted text easily between ML and Scala, using asynchronous protocol commands.

PIDE is a general framework for Prover IDEs based on Isabelle/Scala. It is built around a concept of parallel and asynchronous document processing, which is supported natively by the parallel proof engine that is implemented in Isabelle/ML. The traditional prover command loop is given up; instead there is direct support for editing of source text, with rich formal markup for GUI rendering.

jEdit is a sophisticated text editor[1] implemented in Java[2]. It is easily extensible by plugins written in any language that works on the JVM. In the context of Isabelle this is always Scala[3].

[1] http://www.jedit.org
[2] http://www.java.com
[3] http://www.scala-lang.org

Isabelle/jEdit is the main application of the PIDE framework and the default user-interface for Isabelle. It targets both beginners and experts. Technically, Isabelle/jEdit consists of the original jEdit code base with minimal patches and a special plugin for Isabelle. This is integrated as a desktop application for the main operating system families: Linux, Windows, Mac OS X.

End-users of Isabelle download and run a standalone application that exposes jEdit as a text editor on the surface. Thus there is occasionally a tendency to apply the name "jEdit" to any of the Isabelle Prover IDE aspects, without proper differentiation. When discussing these PIDE building blocks in public forums, mailing lists, or even scientific publications, it is particularly important to distinguish Isabelle/ML versus Standard ML, Isabelle/Scala versus Scala, Isabelle/jEdit versus jEdit.

1.2 The Isabelle/jEdit Prover IDE

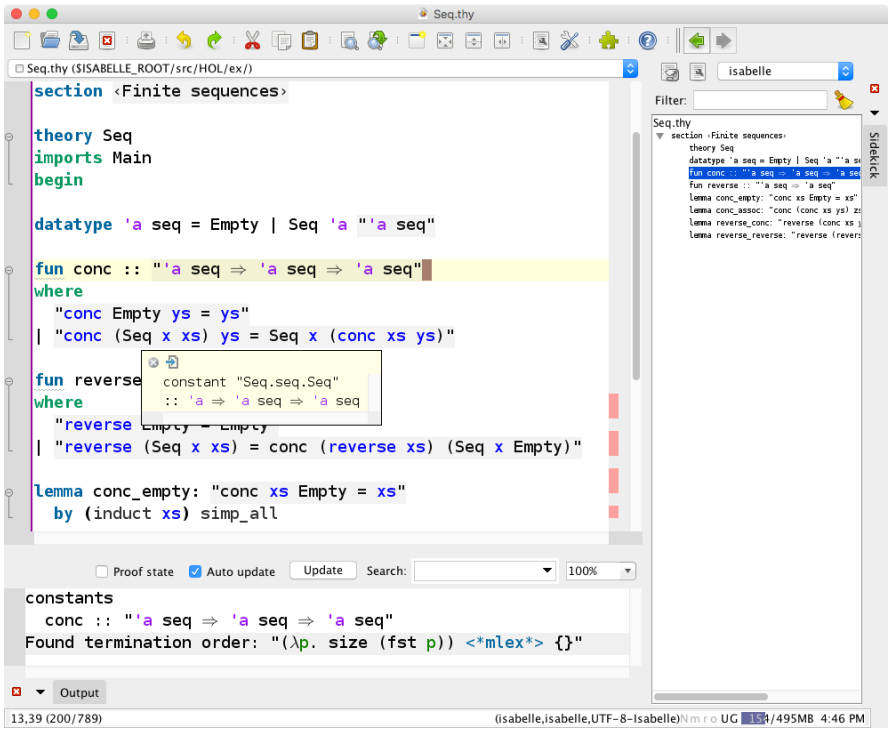


Figure 1.1: The Isabelle/jEdit Prover IDE

Isabelle/jEdit (figure 1.1) consists of some plugins for the jEdit text editor, while preserving its general look-and-feel as far as possible. The main plugin is called "Isabelle" and has its own menu *Plugins / Isabelle* with access to several panels (see also §2.1), as well as *Plugins / Plugin Options / Isabelle* (see also §1.2.3).

The options allow to specify a logic session name, but the same selector is also accessible in the *Theories* panel (§3.1.2). After application startup, the selected logic session image is provided automatically by the Isabelle build tool [3]: if it is absent or outdated wrt. its sources, the build process updates it while the text editor is running. Prover IDE functionality is only activated after successful termination of the build process. A failure may require changing some options and restart the application. Changing the logic session, or the underlying ML system platform (32 bit versus 64 bit) requires a restart of the application to take effect.

The main job of the Prover IDE is to manage sources and their changes, taking the logical structure as a formal document into account (see also §3.1). The editor and the prover are connected asynchronously in a lock-free manner. The prover is free to organize the checking of the formal text in parallel on multiple cores, and provides feedback via markup, which is rendered in the editor via colors, boxes, squiggly underlines, hyperlinks, popup windows, icons, clickable output etc.

Using the mouse together with the modifier key `CONTROL` (Linux, Windows) or `COMMAND` (Mac OS X) exposes formal content via tooltips and/or hyperlinks (see also §3.5). Output (in popups etc.) may be explored recursively, using the same techniques as in the editor source buffer.

Thus the Prover IDE gives an impression of direct access to formal content of the prover within the editor, but in reality only certain aspects are exposed, according to the possibilities of the prover and its add-on tools.

1.2.1 Documentation

The *Documentation* panel of Isabelle/jEdit provides access to some example theory files and the standard Isabelle documentation. PDF files are opened by regular desktop operations of the underlying platform. The section "Original jEdit Documentation" contains the original *User's Guide* of this sophisticated text editor. The same is accessible via the `Help` menu or `F1` keyboard shortcut, using the built-in HTML viewer of Java/Swing. The latter also includes *Frequently Asked Questions* and documentation of individual plugins.

Most of the information about jEdit is relevant for Isabelle/jEdit as well, but one needs to keep in mind that defaults sometimes differ, and the official jEdit documentation does not know about the Isabelle plugin with its support for continuous checking of formal source text: jEdit is a plain text editor, but Isabelle/jEdit is a Prover IDE.

1.2.2 Plugins

The *Plugin Manager* of jEdit allows to augment editor functionality by JVM modules (jars) that are provided by the central plugin repository, which is accessible via various mirror sites.

Connecting to the plugin server-infrastructure of the jEdit project allows to update bundled plugins or to add further functionality. This needs to be done with the usual care for such an open bazaar of contributions. Arbitrary combinations of add-on features are apt to cause problems. It is advisable to start with the default configuration of Isabelle/jEdit and develop some sense how it is meant to work, before loading too many other plugins.

The main *Isabelle* plugin is an integral part of Isabelle/jEdit and needs to remain active at all times! A few additional plugins are bundled with Isabelle/jEdit for convenience or out of necessity, notably *Console* with its Isabelle/Scala sub-plugin (§2.3) and *SideKick* with some Isabelle-specific parsers for document tree structure (§2.6). The *Navigator* plugin is particularly important for hyperlinks within the formal document-model (§3.5). Further plugins (e.g. *ErrorList*, *Code2HTML*) are included to saturate the dependencies of bundled plugins, but have no particular use in Isabelle/jEdit.

1.2.3 Options

Both jEdit and Isabelle have distinctive management of persistent options.

Regular jEdit options are accessible via the dialogs *Utilities / Global Options* or *Plugins / Plugin Options*, with a second chance to flip the two within the central options dialog. Changes are stored in `$JEDIT_SETTINGS/properties` and `$JEDIT_SETTINGS/keymaps`.

Isabelle system options are managed by Isabelle/Scala and changes are stored in `$ISABELLE_HOME_USER/etc/preferences`, independently of other jEdit properties. See also [3], especially the coverage of sessions and command-line tools like `isabelle build` or `isabelle options`.

Those Isabelle options that are declared as `public` are configurable in Isabelle/jEdit via *Plugin Options / Isabelle / General*. Moreover, there

are various options for rendering of document content, which are configurable via *Plugin Options / Isabelle / Rendering*. Thus *Plugin Options / Isabelle* in jEdit provides a view on a subset of Isabelle system options. Note that some of these options affect general parameters that are relevant outside Isabelle/jEdit as well, e.g. `threads` or `parallel_proofs` for the Isabelle build tool [3], but it is possible to use the settings variable `ISABELLE_BUILD_OPTIONS` to change defaults for batch builds without affecting the Prover IDE.

The jEdit action `isabelle.options` opens the options dialog for the Isabelle plugin; it can be mapped to editor GUI elements as usual.

Options are usually loaded on startup and saved on shutdown of Isabelle/jEdit. Editing the machine-generated `$JEDIT_SETTINGS/properties` or `$ISABELLE_HOME_USER/etc/preferences` manually while the application is running is likely to cause surprise due to lost update!

1.2.4 Keymaps

Keyboard shortcuts are managed as a separate concept of *keymap* that is configurable via *Global Options / Shortcuts*. The `imported` keymap is derived from the initial environment of properties that is available at the first start of the editor; afterwards the keymap file takes precedence and is no longer affected by change of default properties.

Users may change their keymap later, but need to keep its content `$JEDIT_SETTINGS/keymaps` in sync with `shortcut` properties in `$JEDIT_HOME/src/jEdit.props`.

The action `isabelle.keymap-merge` helps to resolve pending Isabelle keymap changes that are in conflict with the current jEdit keymap; non-conflicting changes are always applied implicitly. This action is automatically invoked on Isabelle/jEdit startup.

1.3 Command-line invocation

Isabelle/jEdit is normally invoked as a single-instance desktop application, based on platform-specific executables for Linux, Windows, Mac OS X.

It is also possible to invoke the Prover IDE on the command-line, with some extra options and environment settings. The command-line usage of `isabelle jedit` is as follows:

```
Usage: isabelle jedit [OPTIONS] [FILES ...]

  Options are:
    -D NAME=X    set JVM system property
    -J OPTION    add JVM runtime option
    -b           build only
    -d DIR       include session directory
    -f           fresh build
    -j OPTION    add jEdit runtime option
    -l NAME      logic image name
    -m MODE      add print mode for output
    -n           no build of session image on startup
    -p CMD       ML process command prefix (process policy)
    -s           system build mode for session image

  Start jEdit with Isabelle plugin setup and open FILES
  (default "$USER_HOME/Scratch.thy" or ":" for empty buffer).
```

The `-l` option specifies the session name of the logic image to be used for proof processing. Additional session root directories may be included via option `-d` to augment that name space of `isabelle build` [3].

By default, the specified image is checked and built on demand. The `-s` option determines where to store the result session image of `isabelle build`. The `-n` option bypasses the implicit build process for the selected session image.

The `-m` option specifies additional print modes for the prover process. Note that the system option `jedit_print_mode` allows to do the same persistently (e.g. via the *Plugin Options* dialog of Isabelle/jEdit), without requiring command-line invocation.

The `-J` and `-j` options allow to pass additional low-level options to the JVM or jEdit, respectively. The defaults are provided by the Isabelle settings environment [3], but note that these only work for the command-line tool described here, and not the regular application.

The `-D` option allows to define JVM system properties; this is passed directly to the underlying `java` process.

The `-b` and `-f` options control the self-build mechanism of Isabelle/jEdit. This is only relevant for building from sources, which also requires an auxiliary `jedit_build` component from http://isabelle.in.tum.de/components. The official Isabelle release already includes a pre-built version of Isabelle/jEdit.

It is also possible to connect to an already running Isabelle/jEdit process via `isabelle jedit_client`:

```
Usage: isabelle jedit_client [OPTIONS] [FILES ...]

  Options are:
    -c           only check presence of server
    -n           only report server name
    -s NAME      server name (default Isabelle)

  Connect to already running Isabelle/jEdit instance and open FILES
```

The `-c` option merely checks the presence of the server, producing a process return code accordingly.

The `-n` option reports the server name, and the `-s` option provides a different server name. The default server name is the official distribution name (e.g. `Isabelle2016-1`). Thus `isabelle jedit_client` can connect to the Isabelle desktop application without further options.

The `-p` option allows to override the implicit default of the system option `ML_process_policy` for ML processes started by the Prover IDE, e.g. to control CPU affinity on multiprocessor systems.

The JVM system property `isabelle.jedit_server` provides a different server name, e.g. use `isabelle jedit -Disabelle.jedit_server=`*name* and `isabelle jedit_client -s` *name* to connect later on.

1.4 GUI rendering

1.4.1 Look-and-feel

jEdit is a Java/AWT/Swing application with some ambition to support "native" look-and-feel on all platforms, within the limits of what Oracle as Java provider and major operating system distributors allow (see also §7).

Isabelle/jEdit enables platform-specific look-and-feel by default as follows.

Linux: The platform-independent *Metal* is used by default.

The Linux-specific *GTK+* also works under the side-condition that the overall GTK theme and options are selected in a way that works with Java AWT/Swing. The JVM has no direct influence of GTK rendering.

Windows: Regular *Windows* is used by default.

Mac OS X: Regular *Mac OS X* is used by default.

The bundled *MacOSX* plugin provides various functions that are expected from applications on that particular platform: quit from menu

or dock, preferences menu, drag-and-drop of text files on the application, full-screen mode for main editor windows. It is advisable to have the *MacOSX* plugin enabled all the time on that platform.

Users may experiment with different Swing look-and-feels, but need to keep in mind that this extra variance of GUI functionality is unlikely to work in arbitrary combinations. The platform-independent *Metal* and *Nimbus* should always work on all platforms, although they are technically and stylistically outdated. The historic *CDE/Motif* should be ignored.

After changing the look-and-feel in *Global Options / Appearance*, Isabelle/jEdit should be restarted to take full effect.

1.4.2 Displays with very high resolution

In distant past, displays with 1024×768 or 1280×1024 pixels were considered "high resolution" and bitmap fonts with 12 or 14 pixels as adequate for text rendering. In 2016, we routinely see much higher resolutions, e.g. "Full HD" at 1920×1080 pixels or "Ultra HD" / "4K" at 3840×2160.

GUI frameworks are usually lagging behind, with hard-wired icon sizes and tiny fonts. Java and jEdit do provide reasonable support for very high resolution, but this requires manual adjustments as described below.

The **operating-system** usually provides some configuration for global scaling of text fonts, e.g. 120%–250% on Windows. This impacts regular GUI elements, when used with native look-and-feel: Linux *GTK+*, *Windows*, *Mac OS X*, respectively. Alternatively, it is possible to use the platform-independent *Metal* look-and-feel and readjust its main font sizes via jEdit options explained below. The Isabelle/jEdit **application** provides further options to adjust font sizes in particular GUI elements. Here is a summary of all relevant font properties:

- *Global Options / Text Area / Text font*: the main text area font, which is also used as reference point for various derived font sizes, e.g. the *Output* (§3.2) and *State* (§3.3) panels.
- *Global Options / Gutter / Gutter font*: the font for the gutter area left of the main text area, e.g. relevant for display of line numbers (disabled by default).
- *Global Options / Appearance / Button, menu and label font* as well as *List and text field font*: this specifies the primary and secondary font for the *Metal* look-and-feel (§1.4.1).

- *Plugin Options / Isabelle / General / Reset Font Size*: the main text area font size for action `isabelle.reset-font-size`, e.g. relevant for quick scaling like in common web browsers.
- *Plugin Options / Console / General / Font*: the console window font, e.g. relevant for Isabelle/Scala command-line.

In figure 1.2 the *Metal* look-and-feel is configured with custom fonts at 30 pixels, and the main text area and console at 36 pixels. This leads to decent rendering quality, despite the old-fashioned appearance of *Metal*.

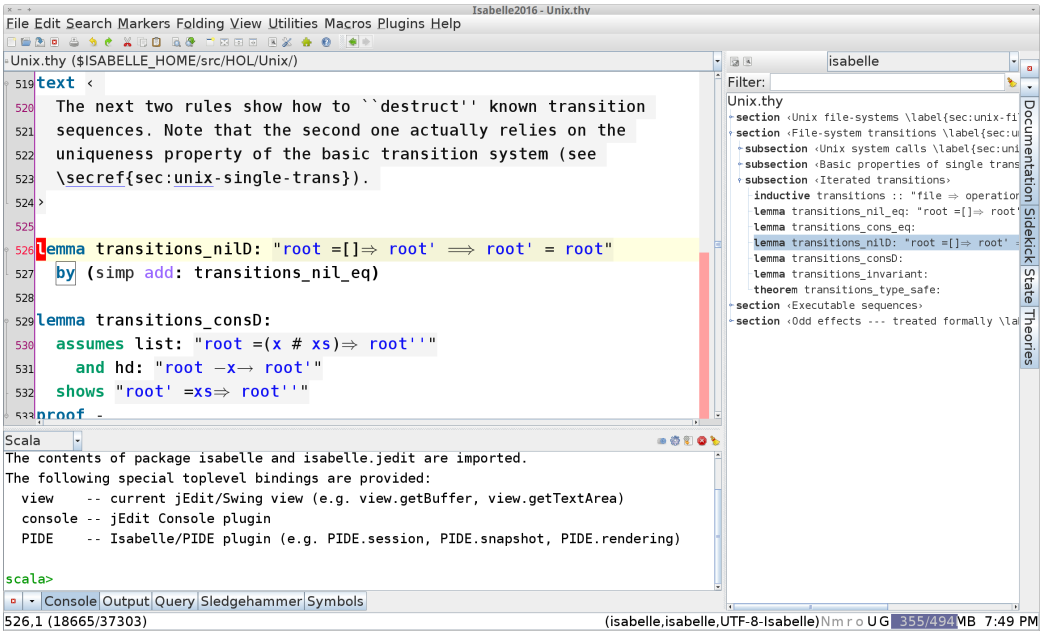


Figure 1.2: Metal look-and-feel with custom fonts for very high resolution

Chapter 2

Augmented jEdit functionality

2.1 Dockable windows

In jEdit terminology, a *view* is an editor window with one or more *text areas* that show the content of one or more *buffers*. A regular view may be surrounded by *dockable windows* that show additional information in arbitrary format, not just text; a *plain view* does not allow dockables. The *dockable window manager* of jEdit organizes these dockable windows, either as *floating* windows, or *docked* panels within one of the four margins of the view. There may be any number of floating instances of some dockable window, but at most one docked instance; jEdit actions that address *the* dockable window of a particular kind refer to the unique docked instance.

Dockables are used routinely in jEdit for important functionality like *HyperSearch Results* or the *File System Browser*. Plugins often provide a central dockable to access their main functionality, which may be opened by the user on demand. The Isabelle/jEdit plugin takes this approach to the extreme: its plugin menu provides the entry-points to many panels that are managed as dockable windows. Some important panels are docked by default, e.g. *Documentation*, *State*, *Theories Output*, *Query*. The user can change this arrangement easily and persistently.

Compared to plain jEdit, dockable window management in Isabelle/jEdit is slightly augmented according to the the following principles:

- Floating windows are dependent on the main window as *dialog* in the sense of Java/AWT/Swing. Dialog windows always stay on top of the view, which is particularly important in full-screen mode. The desktop environment of the underlying platform may impose further policies on such dependent dialogs, in contrast to fully independent windows, e.g. some window management functions may be missing.

- Keyboard focus of the main view vs. a dockable window is carefully managed according to the intended semantics, as a panel mainly for output or input. For example, activating the *Output* (§3.2) or State

(§3.3) panel via the dockable window manager returns keyboard focus to the main text area, but for *Query* (§3.4) or *Sledgehammer* §3.9 the focus is given to the main input field of that panel.

- Panels that provide their own text area for output have an additional dockable menu item *Detach*. This produces an independent copy of the current output as a floating *Info* window, which displays that content independently of ongoing changes of the PIDE document-model. Note that Isabelle/jEdit popup windows (§3.5) provide a similar *Detach* operation as an icon.

2.2 Isabelle symbols

Isabelle sources consist of *symbols* that extend plain ASCII to allow infinitely many mathematical symbols within the formal sources. This works without depending on particular encodings and varying Unicode standards.[1] See [8].

For the prover back-end, formal text consists of ASCII characters that are grouped according to some simple rules, e.g. as plain "`a`" or symbolic "`\<alpha>`". For the editor front-end, a certain subset of symbols is rendered physically via Unicode glyphs, in order to show "`\<alpha>`" as "α", for example. This symbol interpretation is specified by the Isabelle system distribution in `$ISABELLE_HOME/etc/symbols` and may be augmented by the user in `$ISABELLE_HOME_USER/etc/symbols`.

The appendix of [5] gives an overview of the standard interpretation of finitely many symbols from the infinite collection. Uninterpreted symbols are displayed literally, e.g. "`\<foobar>`". Overlap of Unicode characters used in symbol interpretation with informal ones (which might appear e.g. in comments) needs to be avoided. Raw Unicode characters within prover source files should be restricted to informal parts, e.g. to write text in non-latin alphabets in comments.

Encoding. Technically, the Unicode interpretation of Isabelle symbols is an *encoding* called `UTF-8-Isabelle` in jEdit (*not* in the underlying JVM). It is provided by the Isabelle/jEdit plugin and enabled by default for all source files. Sometimes such defaults are reset accidentally, or malformed UTF-8

[1] Raw Unicode characters within formal sources would compromise portability and reliability in the face of changing interpretation of special features of Unicode, such as Combining Characters or Bi-directional Text.

sequences in the text force jEdit to fall back on a different encoding like `ISO-8859-15`. In that case, verbatim "`\<alpha>`" will be shown in the text buffer instead of its Unicode rendering "α". The jEdit menu operation *File / Reload with Encoding / UTF-8-Isabelle* helps to resolve such problems (after repairing malformed parts of the text).

Font. Correct rendering via Unicode requires a font that contains glyphs for the corresponding codepoints. There are also various unusual symbols with particular purpose in Isabelle, e.g. control symbols and very long arrows. Isabelle/jEdit prefers its own application fonts `IsabelleText`, which ensures that standard collection of Isabelle symbols is actually shown on the screen (or printer) as expected.

Note that a Java/AWT/Swing application can load additional fonts only if they are not installed on the operating system already! Some outdated version of `IsabelleText` that happens to be provided by the operating system would prevent Isabelle/jEdit to use its bundled version. This could lead to missing glyphs (black rectangles), when the system version of `IsabelleText` is older than the application version. This problem can be avoided by refraining to "install" any version of `IsabelleText` in the first place, although it might be tempting to use the same font in other applications.

HTML pages generated by Isabelle refer to the same `IsabelleText` font as a server-side resource. Thus a web-browser can use that without requiring a locally installed copy.

Input methods. In principle, Isabelle/jEdit could delegate the problem to produce Isabelle symbols in their Unicode rendering to the underlying operating system and its *input methods*. Regular jEdit also provides various ways to work with *abbreviations* to produce certain non-ASCII characters. Since none of these standard input methods work satisfactorily for the mathematical characters required for Isabelle, various specific Isabelle/jEdit mechanisms are provided.

This is a summary for practically relevant input methods for Isabelle symbols.

1. The *Symbols* panel: some GUI buttons allow to insert certain symbols in the text buffer. There are also tooltips to reveal the official Isabelle representation with some additional information about *symbol abbreviations* (see below).

2. Copy/paste from decoded source files: text that is rendered as Unicode already can be re-used to produce further text. This also works between

different applications, e.g. Isabelle/jEdit and some web browser or mail client, as long as the same Unicode interpretation of Isabelle symbols is used.

3. Copy/paste from prover output within Isabelle/jEdit. The same principles as for text buffers apply, but note that *copy* in secondary Isabelle/jEdit windows works via the keyboard shortcuts `C+c` or `C+INSERT`, while jEdit menu actions always refer to the primary text area!

4. Completion provided by the Isabelle plugin (see §3.7). Isabelle symbols have a canonical name and optional abbreviations. This can be used with the text completion mechanism of Isabelle/jEdit, to replace a prefix of the actual symbol like `\<lambda>`, or its name preceded by backslash `\lambda`, or its ASCII abbreviation `%` by the Unicode rendering.

   The following table is an extract of the information provided by the standard `$ISABELLE_HOME/etc/symbols` file:

symbol	**name with backslash**	**abbreviation**
λ	`\lambda`	`%`
$\Rightarrow$	`\Rightarrow`	`=>`
$\Longrightarrow$	`\Longrightarrow`	`==>`
$\bigwedge$	`\And`	`!!`
$\equiv$	`\equiv`	`==`
$\forall$	`\forall`	`!`
$\exists$	`\exists`	`?`
$\longrightarrow$	`\longrightarrow`	`-->`
$\wedge$	`\and`	`&`
$\vee$	`\or`	`|`
$\neg$	`\not`	`~`
$\neq$	`\noteq`	`~=`
$\in$	`\in`	`:`
$\notin$	`\notin`	`~:`

   Note that the above abbreviations refer to the input method. The logical notation provides ASCII alternatives that often coincide, but sometimes deviate. This occasionally causes user confusion with old-fashioned Isabelle source that use ASCII replacement notation like `!` or `ALL` directly in the text.

On the other hand, coincidence of symbol abbreviations with ASCII replacement syntax syntax helps to update old theory sources via explicit completion (see also `C+b` explained in §3.7).

Control symbols. There are some special control symbols to modify the display style of a single symbol (without nesting). Control symbols may be applied to a region of selected text, either using the *Symbols* panel or keyboard shortcuts or jEdit actions. These editor operations produce a separate control symbol for each symbol in the text, in order to make the whole text appear in a certain style.

style	symbol	shortcut	action
superscript	`\<^sup>`	`C+e UP`	`isabelle.control-sup`
subscript	`\<^sub>`	`C+e DOWN`	`isabelle.control-sub`
bold face	`\<^bold>`	`C+e RIGHT`	`isabelle.control-bold`
emphasized	`\<^emph>`	`C+e LEFT`	`isabelle.control-emph`
reset		`C+e BACK_SPACE`	`isabelle.control-reset`

To produce a single control symbol, it is also possible to complete on `\sup`, `\sub`, `\bold`, `\emph` as for regular symbols.

The emphasized style only takes effect in document output (when used with a cartouche), but not in the editor.

2.3 Scala console

The *Console* plugin manages various shells (command interpreters), e.g. *BeanShell*, which is the official jEdit scripting language, and the cross-platform *System* shell. Thus the console provides similar functionality than the Emacs buffers `*scratch*` and `*shell*`.

Isabelle/jEdit extends the repertoire of the console by *Scala*, which is the regular Scala toplevel loop running inside the same JVM process as Isabelle/jEdit itself. This means the Scala command interpreter has access to the JVM name space and state of the running Prover IDE application. The default environment imports the full content of packages `isabelle` and `isabelle.jedit`.

For example, `PIDE` refers to the Isabelle/jEdit plugin object, and `view` to the current editor view of jEdit. The Scala expression `PIDE.snapshot(view)` makes a PIDE document snapshot of the current buffer within the current editor view.

This helps to explore Isabelle/Scala functionality interactively. Some care is required to avoid interference with the internals of the running application.

2.4 File-system access

File specifications in jEdit follow various formats and conventions according to *Virtual File Systems*, which may be also provided by additional plugins. This allows to access remote files via the `http:` protocol prefix, for example. Isabelle/jEdit attempts to work with the file-system model of jEdit as far as possible. In particular, theory sources are passed directly from the editor to the prover, without indirection via physical files.

Despite the flexibility of URLs in jEdit, local files are particularly important and are accessible without protocol prefix. The file path notation is that of the Java Virtual Machine on the underlying platform. On Windows the preferred form uses backslashes, but happens to accept forward slashes like Unix/POSIX as well. Further differences arise due to Windows drive letters and network shares.

The Java notation for files needs to be distinguished from the one of Isabelle, which uses POSIX notation with forward slashes on *all* platforms. Isabelle/ML on Windows uses Unix-style path notation, too, and driver letter representation as in Cygwin (e.g. `/cygdrive/c`). Moreover, environment variables from the Isabelle process may be used freely, e.g. `$ISABELLE_HOME/etc/symbols` or `$POLYML_HOME/README`. There are special shortcuts: `~` for `$USER_HOME` and `~~` for `$ISABELLE_HOME`.

Since jEdit happens to support environment variables within file specifications as well, it is natural to use similar notation within the editor, e.g. in the file-browser. This does not work in full generality, though, due to the bias of jEdit towards platform-specific notation and of Isabelle towards POSIX. Moreover, the Isabelle settings environment is not yet active when starting Isabelle/jEdit via its standard application wrapper, in contrast to `isabelle jedit` run from the command line (§1.3).

Isabelle/jEdit imitates important system settings within the Java process environment, in order to allow easy access to these important places from the editor: `$ISABELLE_HOME`, `$ISABELLE_HOME_USER`, `$JEDIT_HOME`, `$JEDIT_SETTINGS`. The file browser of jEdit also includes *Favorites* for these two important locations.

Path specifications in prover input or output usually include formal markup that turns it into a hyperlink (see also §3.5). This allows to open the corre-

sponding file in the text editor, independently of the path notation. If the path refers to a directory, the jEdit file browser is opened on it.

Formally checked paths in prover input are subject to completion (§3.7): partial specifications are resolved via directory content and possible completions are offered in a popup.

2.5 Indentation

Isabelle/jEdit augments the existing indentation facilities of jEdit to take the structure of theory and proof texts into account. There is also special support for unstructured proof scripts.

Syntactic indentation follows the outer syntax of Isabelle/Isar.

Action `indent-lines` (shortcut `C+i`) indents the current line according to command keywords and some command substructure: this approximation may need further manual tuning.

Action `isabelle.newline` (shortcut `ENTER`) indents the old and the new line according to command keywords only: this leads to precise alignment of the main Isar language elements. This depends on option `jedit_indent_newline` (enabled by default).

Semantic indentation adds additional white space to unstructured proof scripts (**apply** etc.) via number of subgoals. This requires information of ongoing document processing and may thus lag behind, when the user is editing too quickly; see also option `jedit_script_indent` and `jedit_script_indent_limit`.

The above options are accessible in the menu *Plugins / Plugin Options / Isabelle / General*.

Automatic indentation has a tendency to produce trailing whitespace. That can be purged manually with the jEdit action `remove-trailing-ws` (shortcut `C+e r`). Moreover, the *WhiteSpace* plugin provides some aggressive options to trim whitespace on buffer-save.

2.6 SideKick parsers

The *SideKick* plugin provides some general services to display buffer structure in a tree view. Isabelle/jEdit provides SideKick parsers for its main mode

for theory files, ML files, as well as some minor modes for the `NEWS` file (see figure 2.1), session `ROOT` files, system `options`, and BibTEX files (§4.3).

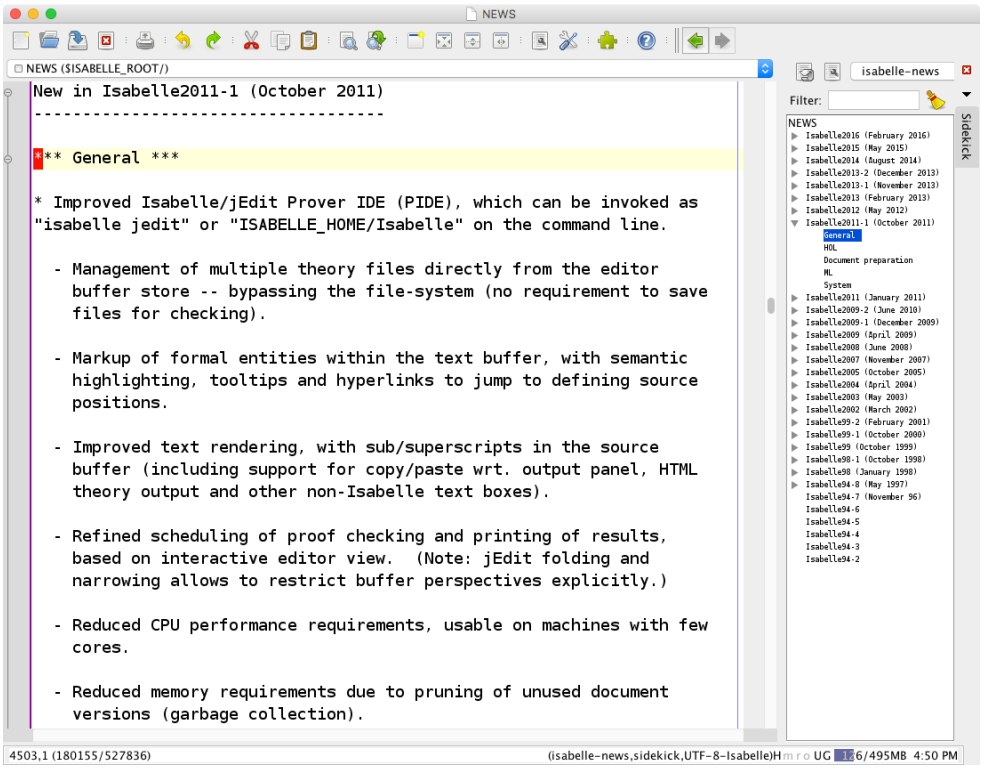


Figure 2.1: The Isabelle NEWS file with SideKick tree view

The default SideKick parser for theory files is `isabelle`: it provides a tree-view on the formal document structure, with section headings at the top and formal specification elements at the bottom. The alternative parser `isabelle-context` shows nesting of context blocks according to **begin** ... **end** structure.

Isabelle/ML files are structured according to semi-formal comments that are explained in [4]. This outline is turned into a tree-view by default, by using the `isabelle-ml` parser. There is also a folding mode of the same name, for hierarchic text folds within ML files.

The special SideKick parser `isabelle-markup` exposes the uninterpreted markup tree of the PIDE document model of the current buffer. This is occasionally useful for informative purposes, but the amount of displayed information might cause problems for large buffers.

Chapter 3

Prover IDE functionality

3.1 Document model

The document model is central to the PIDE architecture: the editor and the prover have a common notion of structured source text with markup, which is produced by formal processing. The editor is responsible for edits of document source, as produced by the user. The prover is responsible for reports of document markup, as produced by its processing in the background.

Isabelle/jEdit handles classic editor events of jEdit, in order to connect the physical world of the GUI (with its singleton state) to the mathematical world of multiple document versions (with timeless and stateless updates).

3.1.1 Editor buffers and document nodes

As a regular text editor, jEdit maintains a collection of *buffers* to store text files; each buffer may be associated with any number of visible *text areas*. Buffers are subject to an *edit mode* that is determined from the file name extension. The following modes are treated specifically in Isabelle/jEdit:

mode	**file name**	**content**
`isabelle`	`*.thy`	theory source
`isabelle-ml`	`*.ML`	Isabelle/ML source
`sml`	`*.sml` or `*.sig`	Standard ML source
`isabelle-root`	`ROOT`	session root
`isabelle-options`		Isabelle options
`isabelle-news`		Isabelle NEWS

All jEdit buffers are automatically added to the PIDE document-model as *document nodes*. The overall document structure is defined by the theory nodes in two dimensions:

1. via **theory imports** that are specified in the *theory header* using concrete syntax of the **theory** command [5];

2. via **auxiliary files** that are loaded into a theory by special *load commands*, notably **ML_file** and **SML_file** [5].

In any case, source files are managed by the PIDE infrastructure: the physical file-system only plays a subordinate role. The relevant version of source text is passed directly from the editor to the prover, using internal communication channels.

3.1.2 Theories

The *Theories* panel (see also figure 3.1) provides an overview of the status of continuous checking of theory nodes within the document model. Unlike batch sessions of `isabelle build` [3], theory nodes are identified by full path names; this allows to work with multiple (disjoint) Isabelle sessions simultaneously within the same editor session.

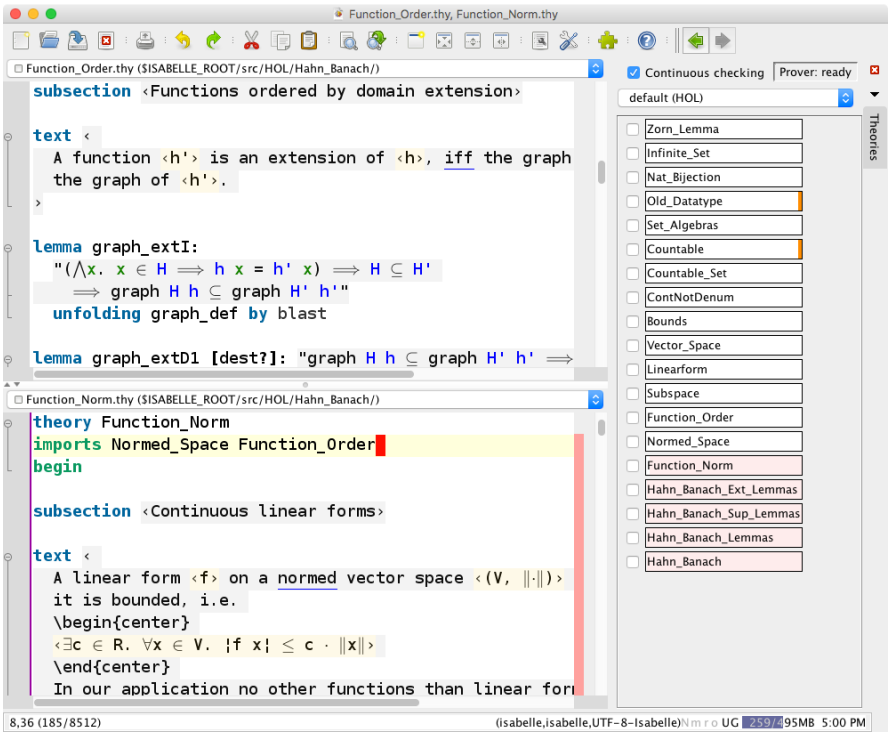


Figure 3.1: Theories panel with an overview of the document-model, and jEdit text areas as editable views on some of the document nodes

Certain events to open or update editor buffers cause Isabelle/jEdit to resolve dependencies of theory imports. The system requests to load additional

files into editor buffers, in order to be included in the document model for further checking. It is also possible to let the system resolve dependencies automatically, according to the system option `jedit_auto_load`.

The visible *perspective* of Isabelle/jEdit is defined by the collective view on theory buffers via open text areas. The perspective is taken as a hint for document processing: the prover ensures that those parts of a theory where the user is looking are checked, while other parts that are presently not required are ignored. The perspective is changed by opening or closing text area windows, or scrolling within a window.

The *Theories* panel provides some further options to influence the process of continuous checking: it may be switched off globally to restrict the prover to superficial processing of command syntax. It is also possible to indicate theory nodes as *required* for continuous checking: this means such nodes and all their imports are always processed independently of the visibility status (if continuous checking is enabled). Big theory libraries that are marked as required can have significant impact on performance!

Formal markup of checked theory content is turned into GUI rendering, based on a standard repertoire known from mainstream IDEs for programming languages: colors, icons, highlighting, squiggly underlines, tooltips, hyperlinks etc. For outer syntax of Isabelle/Isar there is some traditional syntax-highlighting via static keywords and tokenization within the editor; this buffer syntax is determined from theory imports. In contrast, the painting of inner syntax (term language etc.) uses semantic information that is reported dynamically from the logical context. Thus the prover can provide additional markup to help the user to understand the meaning of formal text, and to produce more text with some add-on tools (e.g. information messages with *sendback* markup by automated provers or disprovers in the background).

3.1.3 Auxiliary files

Special load commands like **ML_file** and **SML_file** [5] refer to auxiliary files within some theory. Conceptually, the file argument of the command extends the theory source by the content of the file, but its editor buffer may be loaded / changed / saved separately. The PIDE document model propagates changes of auxiliary file content to the corresponding load command in the theory, to update and process it accordingly: changes of auxiliary file content are treated as changes of the corresponding load command.

As a concession to the massive amount of ML files in Isabelle/HOL itself, the content of auxiliary files is only added to the PIDE document-model on

demand, the first time when opened explicitly in the editor. There are further tricks to manage markup of ML files, such that Isabelle/HOL may be edited conveniently in the Prover IDE on small machines with only 8 GB of main memory. Using `Pure` as logic session image, the exploration may start at the top `$ISABELLE_HOME/src/HOL/Main.thy` or the bottom `$ISABELLE_HOME/src/HOL/HOL.thy`, for example.

Initially, before an auxiliary file is opened in the editor, the prover reads its content from the physical file-system. After the file is opened for the first time in the editor, e.g. by following the hyperlink (§3.5) for the argument of its **ML_file** command, the content is taken from the jEdit buffer.

The change of responsibility from prover to editor counts as an update of the document content, so subsequent theory sources need to be re-checked. When the buffer is closed, the responsibility remains to the editor: the file may be opened again without causing another document update.

A file that is opened in the editor, but its theory with the load command is not, is presently inactive in the document model. A file that is loaded via multiple load commands is associated to an arbitrary one: this situation is morally unsupported and might lead to confusion.

Output that refers to an auxiliary file is combined with that of the corresponding load command, and shown whenever the file or the command are active (see also §3.2).

Warnings, errors, and other useful markup is attached directly to the positions in the auxiliary file buffer, in the manner of standard IDEs. By using the load command **SML_file** as explained in `$ISABELLE_HOME/src/Tools/SML/Examples.thy`, Isabelle/jEdit may be used as fully-featured IDE for Standard ML, independently of theory or proof development: the required theory merely serves as some kind of project file for a collection of SML source modules.

3.2 Output

Prover output consists of *markup* and *messages*. Both are directly attached to the corresponding positions in the original source text, and visualized in the text area, e.g. as text colours for free and bound variables, or as squiggly underlines for warnings, errors etc. (see also figure 3.2). In the latter case, the corresponding messages are shown by hovering with the mouse over the highlighted text — although in many situations the user should already get some clue by looking at the position of the text highlighting, without seeing the message body itself.

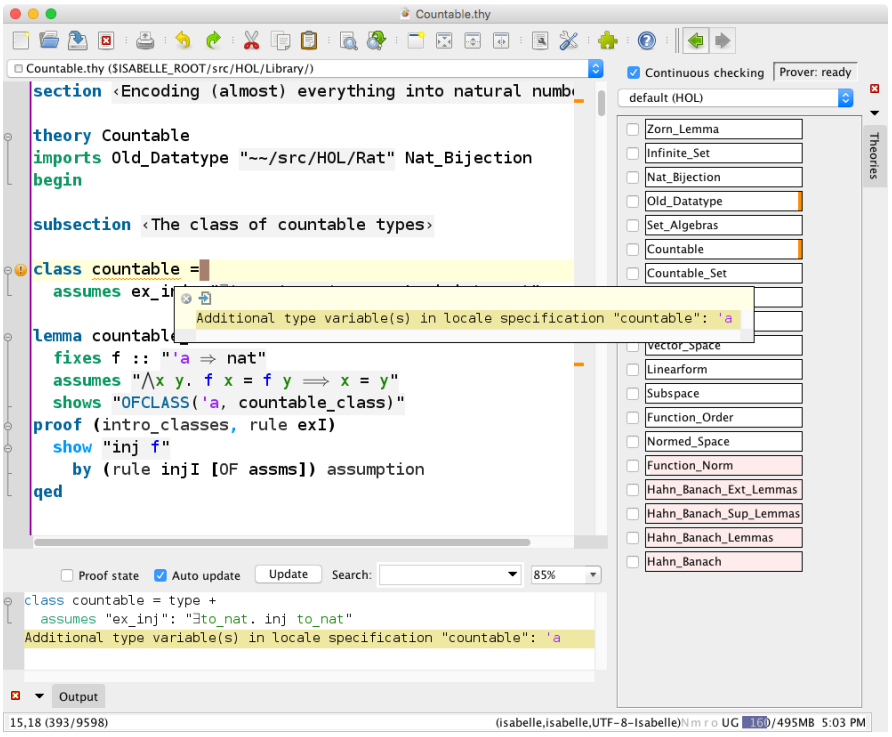


Figure 3.2: Multiple views on prover output: gutter with icon, text area with popup, text overview column, *Theories* panel, *Output* panel

The "gutter" on the left-hand-side of the text area uses icons to provide a summary of the messages within the adjacent text line. Message priorities are used to prefer errors over warnings, warnings over information messages; other output is ignored.

The "text overview column" on the right-hand-side of the text area uses similar information to paint small rectangles for the overall status of the whole text buffer. The graphics is scaled to fit the logical buffer length into the given window height. Mouse clicks on the overview area move the cursor approximately to the corresponding text line in the buffer.

The *Theories* panel provides another course-grained overview, but without direct correspondence to text positions. The coloured rectangles represent the amount of messages of a certain kind (warnings, errors, etc.) and the execution status of commands. A double-click on one of the theory entries with their status overview opens the corresponding text buffer, without moving the cursor to a specific point.

The *Output* panel displays prover messages that correspond to a given com-

mand, within a separate window. The cursor position in the presently active text area determines the prover command whose cumulative message output is appended and shown in that window (in canonical order according to the internal execution of the command). There are also control elements to modify the update policy of the output wrt. continued editor movements: *Auto update* and *Update*. This is particularly useful for multiple instances of the *Output* panel to look at different situations. Alternatively, the panel can be turned into a passive *Info* window via the *Detach* menu item.

Proof state is handled separately (§3.3), but it is also possible to tick the corresponding checkbox to append it to regular output (figure 3.3). This is a globally persistent option: it affects all open panels and future editor sessions.

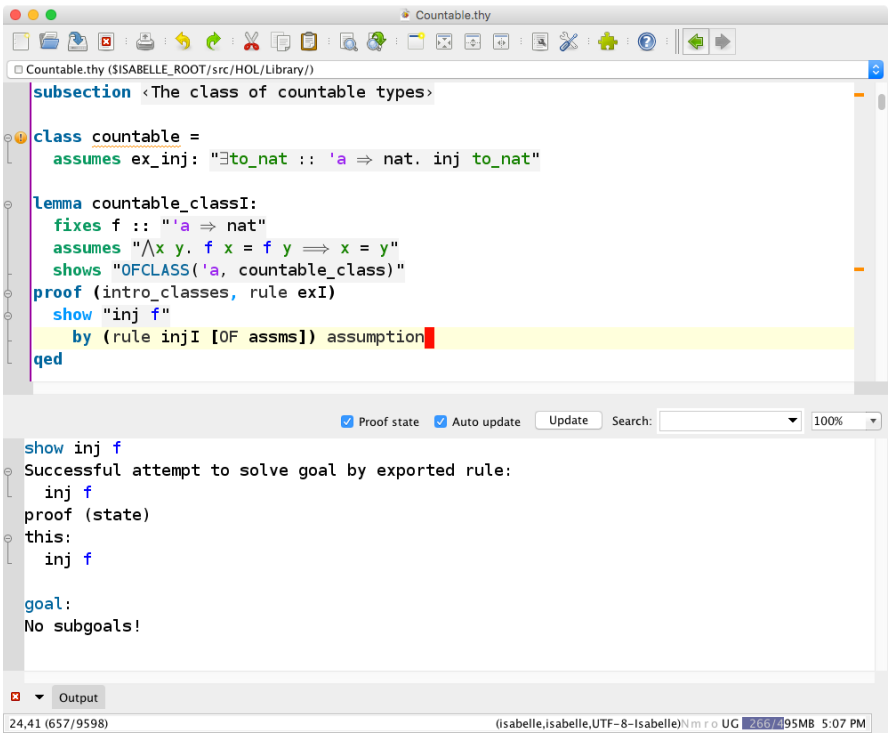


Figure 3.3: Proof state display within the regular output panel

Following the IDE principle, regular messages are attached to the original source in the proper place and may be inspected on demand via popups. This excludes messages that are somehow internal to the machinery of proof checking, notably *proof state* and *tracing*.

In any case, the same display technology is used for small popups and big output windows. The formal text contains markup that may be explored

recursively via further popups and hyperlinks (see §3.5), or clicked directly to initiate certain actions (see §3.8 and §3.9).

3.3 Proof state

The main purpose of the Prover IDE is to help the user editing proof documents, with ongoing formal checking by the prover in the background. This can be done to some extent in the main text area alone, especially for well-structured Isar proofs.

Nonetheless, internal proof state needs to be inspected in many situations of exploration and "debugging". The *State* panel shows exclusively such proof state messages without further distraction, while all other messages are displayed in *Output* (§3.2). Figure 3.4 shows a typical GUI layout where both panels are open.

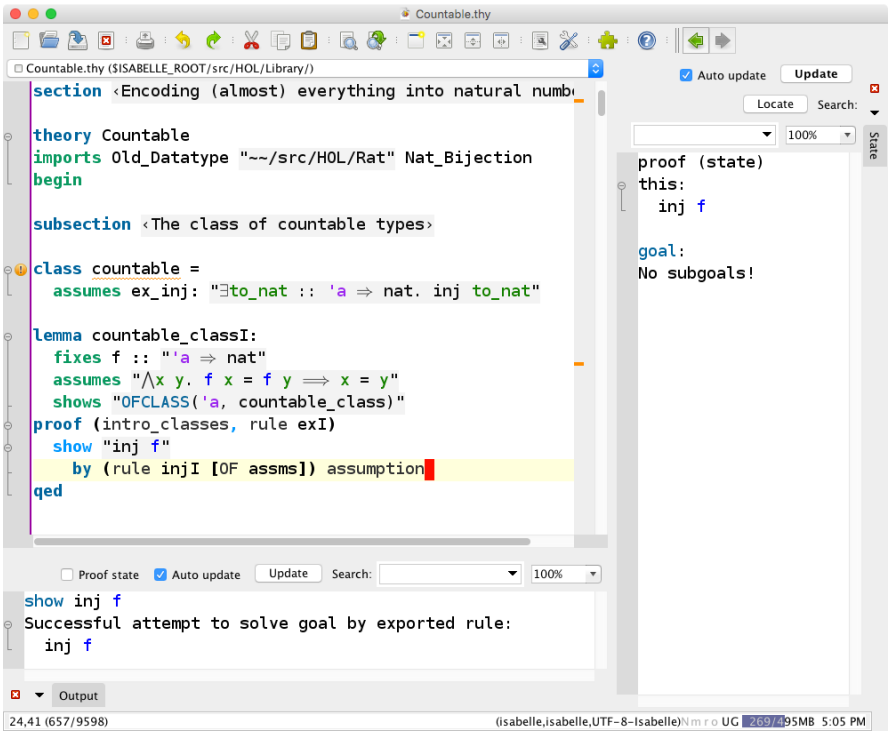


Figure 3.4: Separate proof state display (right) and other output (bottom).

Another typical arrangement has more than one *State* panel open (as floating windows), with *Auto update* disabled to look at an old situation while

the proof text in the vicinity is changed. The *Update* button triggers an explicit one-shot update; this operation is also available via the action `isabelle.update-state` (keyboard shortcut `S+ENTER`).

On small screens, it is occasionally useful to have all messages concatenated in the regular *Output* panel, e.g. see figure 3.3.

The mechanics of *Output* versus *State* are slightly different:

- *Output* shows information that is continuously produced and already present when the GUI wants to show it. This is implicitly controlled by the visible perspective on the text.
- *State* initiates a real-time query on demand, with a full round trip including a fresh print operation on the prover side. This is controlled explicitly when the cursor is moved to the next command (*Auto update*) or the *Update* operation is triggered.

This can make a difference in GUI responsibility and resource usage within the prover process. Applications with very big proof states that are only inspected in isolation work better with the *State* panel.

3.4 Query

The *Query* panel provides various GUI forms to request extra information from the prover, as a replacement of old-style diagnostic commands like **find_theorems**. There are input fields and buttons for a particular query command, with output in a dedicated text area.

The main query modes are presented as separate tabs: *Find Theorems*, *Find Constants*, *Print Context*, e.g. see figure 3.5. As usual in jEdit, multiple *Query* windows may be active at the same time: any number of floating instances, but at most one docked instance (which is used by default).

The following GUI elements are common to all query modes:

- The spinning wheel provides feedback about the status of a pending query wrt. the evaluation of its context and its own operation.
- The *Apply* button attaches a fresh query invocation to the current context of the command where the cursor is pointing in the text.

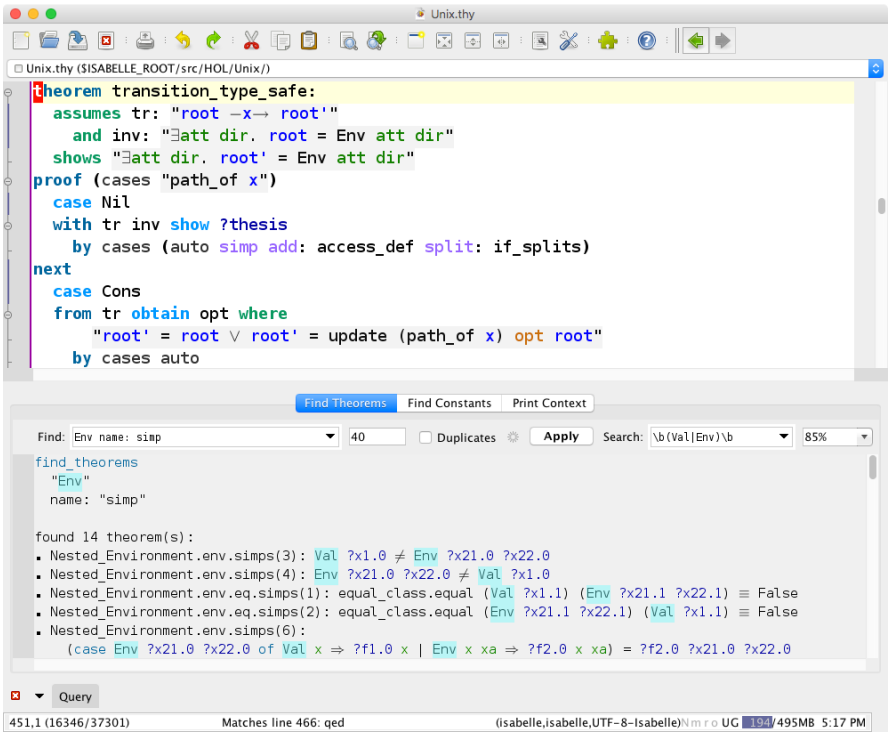


Figure 3.5: An instance of the Query panel: find theorems

- The *Search* field allows to highlight query output according to some regular expression, in the notation that is commonly used on the Java platform.[1] This may serve as an additional visual filter of the result.
- The *Zoom* box controls the font size of the output area.

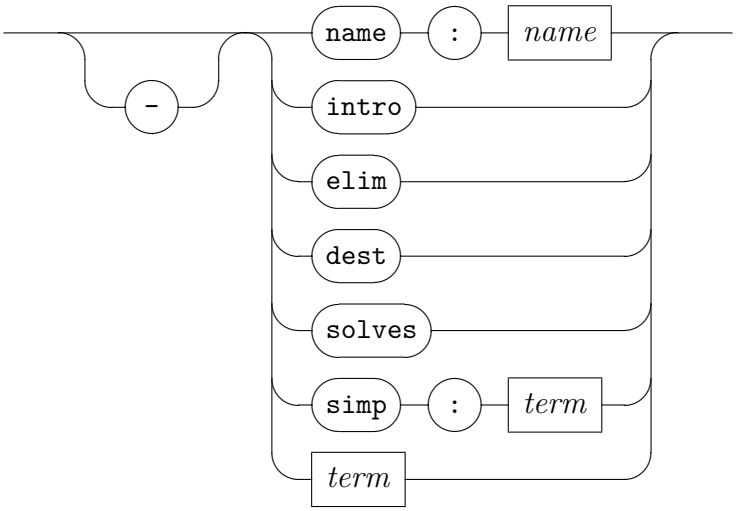
All query operations are asynchronous: there is no need to wait for the evaluation of the document for the query context, nor for the query operation itself. Query output may be detached as independent *Info* window, using a menu operation of the dockable window manager. The printed result usually provides sufficient clues about the original query, with some hyperlink to its context (via markup of its head line).

3.4.1 Find theorems

The *Query* panel in *Find Theorems* mode retrieves facts from the theory or proof context matching all of given criteria in the *Find* text field. A single

[1] https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html

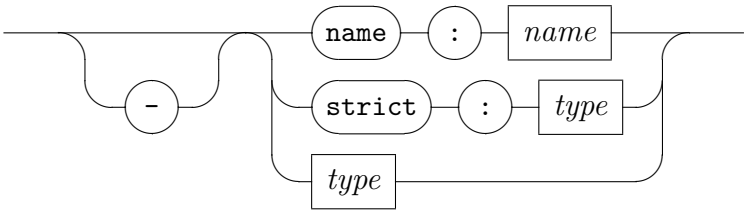
criterium has the following syntax:



See also the Isar command **find_theorems** in [5].

3.4.2 Find constants

The *Query* panel in *Find Constants* mode prints all constants whose type meets all of the given criteria in the *Find* text field. A single criterium has the following syntax:



See also the Isar command **find_consts** in [5].

3.4.3 Print context

The *Query* panel in *Print Context* mode prints information from the theory or proof context, or proof state. See also the Isar commands **print_context**, **print_cases**, **print_term_bindings**, **print_theorems**, described in [5].

3.5 Tooltips and hyperlinks

Formally processed text (prover input or output) contains rich markup that can be explored by using the CONTROL modifier key on Linux and Windows, or COMMAND on Mac OS X. Hovering with the mouse while the modifier is pressed reveals a *tooltip* (grey box over the text with a yellow popup) and/or a *hyperlink* (black rectangle over the text with change of mouse pointer); see also figure 3.6.

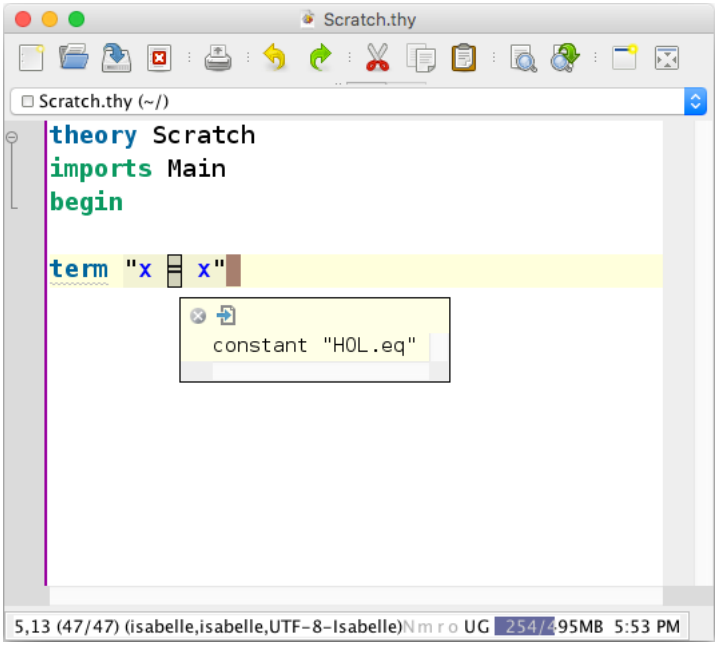


Figure 3.6: Tooltip and hyperlink for some formal entity

Tooltip popups use the same rendering technology as the main text area, and further tooltips and/or hyperlinks may be exposed recursively by the same mechanism; see figure 3.7.

The tooltip popup window provides some controls to *close* or *detach* the window, turning it into a separate *Info* window managed by jEdit. The ESCAPE key closes *all* popups, which is particularly relevant when nested tooltips are stacking up.

A black rectangle in the text indicates a hyperlink that may be followed by a mouse click (while the CONTROL or COMMAND modifier key is still pressed). Such jumps to other text locations are recorded by the *Navigator* plugin, which is bundled with Isabelle/jEdit and enabled by default. There are usually

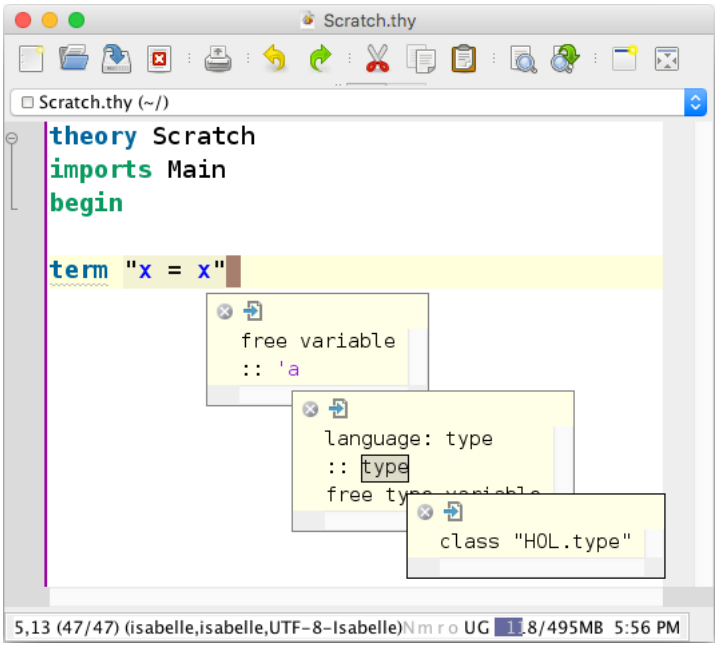


Figure 3.7: Nested tooltips over formal entities

navigation arrows in the main jEdit toolbar.

Note that the link target may be a file that is itself not subject to formal document processing of the editor session and thus prevents further exploration: the chain of hyperlinks may end in some source file of the underlying logic image, or within the ML bootstrap sources of Isabelle/Pure.

3.6 Formal scopes and semantic selection

Formal entities are semantically annotated in the source text as explained in §3.5. A *formal scope* consists of the defining position with all its referencing positions. This correspondence is highlighted in the text according to the cursor position, see also figure 3.8. Here the referencing positions are rendered with an additional border, in reminiscence to a hyperlink: clicking there moves the cursor to the original defining position.

The action `isabelle.select-entity` (shortcut `CS+ENTER`) supports semantic selection of all occurrences of the formal entity at the caret position. This facilitates systematic renaming, using regular jEdit editing of a multi-selection, see also figure 3.9.

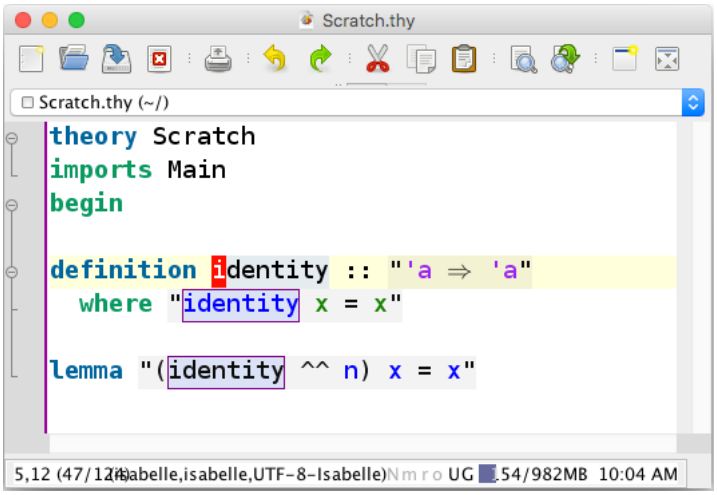


Figure 3.8: Scope of formal entity: defining vs. referencing positions

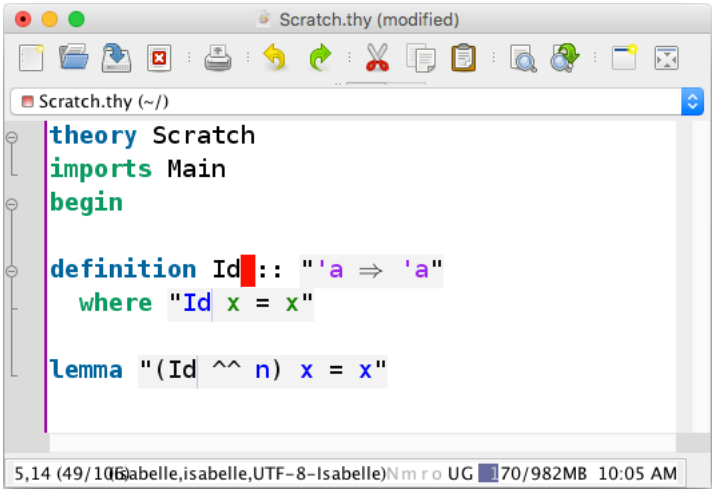


Figure 3.9: The result of semantic selection and systematic renaming

3.7 Completion

Smart completion of partial input is the IDE functionality *par excellance*. Isabelle/jEdit combines several sources of information to achieve that. Despite its complexity, it should be possible to get some idea how completion works by experimentation, based on the overview of completion varieties in §3.7.1. The remaining subsections explain concepts around completion more systematically.

Explicit completion is triggered by the action `isabelle.complete`, which is bound to the keyboard shortcut `C+b`, and thus overrides the jEdit default for `complete-word`.

Implicit completion hooks into the regular keyboard input stream of the editor, with some event filtering and optional delays.

Completion options may be configured in *Plugin Options / Isabelle / General / Completion*. These are explained in further detail below, whenever relevant. There is also a summary of options in §3.7.6.

The asynchronous nature of PIDE interaction means that information from the prover is delayed — at least by a full round-trip of the document update protocol. The default options already take this into account, with a sufficiently long completion delay to speculate on the availability of all relevant information from the editor and the prover, before completing text immediately or producing a popup. Although there is an inherent danger of non-deterministic behaviour due to such real-time parameters, the general completion policy aims at determined results as far as possible.

3.7.1 Varieties of completion

Built-in templates

Isabelle is ultimately a framework of nested sub-languages of different kinds and purposes. The completion mechanism supports this by the following built-in templates:

`‘` (single ASCII back-quote) or `"` (double ASCII quote) support *quotations* via text cartouches. There are three selections, which are always presented in the same order and do not depend on any context information. The default choice produces a template "‹□›", where the box indicates the cursor position after insertion; the other choices help to repair the block structure of unbalanced text cartouches.

`@{` is completed to the template "@{□}", where the box indicates the cursor position after insertion. Here it is convenient to use the wildcard "`__`" or a more specific name prefix to let semantic completion of name-space entries propose antiquotation names.

With some practice, input of quoted sub-languages and antiquotations of embedded languages should work fluently. Note that national keyboard layouts might cause problems with back-quote as dead key, but double quote can be used instead.

Syntax keywords

Syntax completion tables are determined statically from the keywords of the "outer syntax" of the underlying edit mode: for theory files this is the syntax of Isar commands according to the cumulative theory imports.

Keywords are usually plain words, which means the completion mechanism only inserts them directly into the text for explicit completion (§3.7.3), but produces a popup (§3.7.4) otherwise.

At the point where outer syntax keywords are defined, it is possible to specify an alternative replacement string to be inserted instead of the keyword itself. An empty string means to suppress the keyword altogether, which is occasionally useful to avoid confusion, e.g. the rare keyword **simproc_setup** vs. the frequent name-space entry *simp*.

Isabelle symbols

The completion tables for Isabelle symbols (§2.2) are determined statically from `$ISABELLE_HOME/etc/symbols` and `$ISABELLE_HOME_USER/etc/symbols` for each symbol specification as follows:

completion entry	example
literal symbol	`\<forall>`
symbol name with backslash	`\forall`
symbol abbreviation	`ALL` or `!`

When inserted into the text, the above examples all produce the same Unicode rendering ∀ of the underlying symbol `\<forall>`.

A symbol abbreviation that is a plain word, like `ALL`, is treated like a syntax keyword. Non-word abbreviations like `-->` are inserted more aggressively, except for single-character abbreviations like `!` above.

Completion via abbreviations like `ALL` or `-->` depends on the semantic language context (§3.7.2). In contrast, backslash sequences like `\forall` `\<forall>` are always possible, but require additional interaction to confirm (via popup).

The latter is important in ambiguous situations, e.g. for Isabelle document source, which may contain formal symbols or informal LaTeX macros. Backslash sequences also help when input is broken, and thus escapes its normal semantic context: e.g. antiquotations or string literals in ML, which do not allow arbitrary backslash sequences.

User-defined abbreviations

The theory header syntax supports abbreviations via the **abbrevs** keyword [5]. This is a slight generalization of built-in templates and abbreviations for Isabelle symbols, as explained above. Examples may be found in the Isabelle sources, by searching for "`abbrevs`" in `*.thy` files.

The *Symbols* panel shows the abbreviations that are available in the current theory buffer (according to its **imports**) in the `Abbrevs` tab.

Name-space entries

This is genuine semantic completion, using information from the prover, so it requires some delay. A *failed name-space lookup* produces an error message that is annotated with a list of alternative names that are legal. The list of results is truncated according to the system option `completion_limit`. The completion mechanism takes this into account when collecting information on the prover side.

Already recognized names are *not* completed further, but completion may be extended by appending a suffix of underscores. This provokes a failed lookup, and another completion attempt while ignoring the underscores. For example, in a name space where `foo` and `foobar` are known, the input `foo` remains unchanged, but `foo_` may be completed to `foo` or `foobar`.

The special identifier "`__`" serves as a wild-card for arbitrary completion: it exposes the name-space content to the completion mechanism (truncated according to `completion_limit`). This is occasionally useful to explore an unknown name-space, e.g. in some template.

File-system paths

Depending on prover markup about file-system paths in the source text, e.g. for the argument of a load command (§3.1.3), the completion mechanism explores the directory content and offers the result as completion popup. Relative path specifications are understood wrt. the *master directory* of the document node (§3.1.1) of the enclosing editor buffer; this requires a proper theory, not an auxiliary file.

A suffix of slashes may be used to continue the exploration of an already recognized directory name.

Spell-checking

The spell-checker combines semantic markup from the prover (regions of plain words) with static dictionaries (word lists) that are known to the editor.

Unknown words are underlined in the text, using `spell_checker_color` (blue by default). This is not an error, but a hint to the user that some action may be taken. The jEdit context menu provides various actions, as far as applicable:

```
isabelle.complete-word
isabelle.exclude-word
isabelle.exclude-word-permanently
isabelle.include-word
isabelle.include-word-permanently
```

Instead of the specific `isabelle.complete-word`, it is also possible to use the generic `isabelle.complete` with its default keyboard shortcut `C+b`.

Dictionary lookup uses some educated guesses about lower-case, upper-case, and capitalized words. This is oriented on common use in English, where this aspect is not decisive for proper spelling (in contrast to German, for example).

3.7.2 Semantic completion context

Completion depends on a semantic context that is provided by the prover, although with some delay, because at least a full PIDE protocol round-trip is required. Until that information becomes available in the PIDE document-model, the default context is given by the outer syntax of the editor mode (see also §3.1.1).

The semantic *language context* provides information about nested sub-languages of Isabelle: keywords are only completed for outer syntax, and antiquotations for languages that support them. Symbol abbreviations only work for specific sub-languages: e.g. "`=>`" is *not* completed in regular ML source, but is completed within ML strings, comments, antiquotations. Back-slash representations of symbols like "`\foobar`" or "`\<foobar>`" work in any context — after additional confirmation.

The prover may produce *no completion* markup in exceptional situations, to tell that some language keywords should be excluded from further completion attempts. For example, "`:`" within accepted Isar syntax looses its meaning as abbreviation for symbol "∈".

3.7.3 Input events

Completion is triggered by certain events produced by the user, with optional delay after keyboard input according to `jedit_completion_delay`.

Explicit completion works via action `isabelle.complete` with keyboard shortcut `C+b`. This overrides the shortcut for `complete-word` in jEdit, but it is possible to restore the original jEdit keyboard mapping of `complete-word` via *Global Options / Shortcuts* and invent a different one for `isabelle.complete`.

Explicit spell-checker completion works via `isabelle.complete-word`, which is exposed in the jEdit context menu, if the mouse points to a word that the spell-checker can complete.

Implicit completion works via regular keyboard input of the editor. It depends on further side-conditions:

1. The system option `jedit_completion` needs to be enabled (default).
2. Completion of syntax keywords requires at least 3 relevant characters in the text.
3. The system option `jedit_completion_delay` determines an additional delay (0.5 by default), before opening a completion popup. The delay gives the prover a chance to provide semantic completion information, notably the context (§3.7.2).
4. The system option `jedit_completion_immediate` (enabled by default) controls whether replacement text should be inserted immediately without popup, regardless of `jedit_completion_delay`. This aggressive mode of completion is restricted to symbol abbreviations that are not plain words (§2.2).
5. Completion of symbol abbreviations with only one relevant character in the text always enforces an explicit popup, regardless of `jedit_completion_immediate`.

3.7.4 Completion popup

A *completion popup* is a minimally invasive GUI component over the text area that offers a selection of completion items to be inserted into the text, e.g.

by mouse clicks. Items are sorted dynamically, according to the frequency of selection, with persistent history. The popup may interpret special keys `ENTER`, `TAB`, `ESCAPE`, `UP`, `DOWN`, `PAGE_UP`, `PAGE_DOWN`, but all other key events are passed to the underlying text area. This allows to ignore unwanted completions most of the time and continue typing quickly. Thus the popup serves as a mechanism of confirmation of proposed items, while the default is to continue without completion.

The meaning of special keys is as follows:

key	**action**
`ENTER`	select completion (if `jedit_completion_select_enter`)
`TAB`	select completion (if `jedit_completion_select_tab`)
`ESCAPE`	dismiss popup
`UP`	move up one item
`DOWN`	move down one item
`PAGE_UP`	move up one page of items
`PAGE_DOWN`	move down one page of items

Movement within the popup is only active for multiple items. Otherwise the corresponding key event retains its standard meaning within the underlying text area.

3.7.5 Insertion

Completion may first propose replacements to be selected (via a popup), or replace text immediately in certain situations and depending on certain options like `jedit_completion_immediate`. In any case, insertion works uniformly, by imitating normal jEdit text insertion, depending on the state of the *text selection*. Isabelle/jEdit tries to accommodate the most common forms of advanced selections in jEdit, but not all combinations make sense. At least the following important cases are well-defined:

No selection. The original is removed and the replacement inserted, depending on the caret position.

Rectangular selection of zero width. This special case is treated by jEdit as "tall caret" and insertion of completion imitates its normal behaviour: separate copies of the replacement are inserted for each line of the selection.

Other rectangular selection or multiple selections. Here the original is removed and the replacement is inserted for each line (or segment) of the selection.

Support for multiple selections is particularly useful for *HyperSearch*: clicking on one of the items in the *HyperSearch Results* window makes jEdit select all its occurrences in the corresponding line of text. Then explicit completion can be invoked via `C+b`, e.g. to replace occurrences of `-->` by ⟶.

Insertion works by removing and inserting pieces of text from the buffer. This counts as one atomic operation on the jEdit history. Thus unintended completions may be reverted by the regular `undo` action of jEdit. According to normal jEdit policies, the recovered text after `undo` is selected: `ESCAPE` is required to reset the selection and to continue typing more text.

3.7.6 Options

This is a summary of Isabelle/Scala system options that are relevant for completion. They may be configured in *Plugin Options / Isabelle / General* as usual.

- `completion_limit` specifies the maximum number of items for various semantic completion operations (name-space entries etc.)
- `jedit_completion` guards implicit completion via regular jEdit key events (§3.7.3): it allows to disable implicit completion altogether.
- `jedit_completion_select_enter` and `jedit_completion_select_tab` enable keys to select a completion item from the popup (§3.7.4). Note that a regular mouse click on the list of items is always possible.
- `jedit_completion_context` specifies whether the language context provided by the prover should be used at all. Disabling that option makes completion less "semantic". Note that incomplete or severely broken input may cause some disagreement of the prover and the user about the intended language context.
- `jedit_completion_delay` and `jedit_completion_immediate` determine the handling of keyboard events for implicit completion (§3.7.3).

  A `jedit_completion_delay > 0` postpones the processing of key events, until after the user has stopped typing for the given time span, but `jedit_completion_immediate = true` means that abbreviations of Isabelle symbols are handled nonetheless.
- `jedit_completion_path_ignore` specifies "glob" patterns to ignore in file-system path completion (separated by colons), e.g. backup files ending with tilde.

- `spell_checker` is a global guard for all spell-checker operations: it allows to disable that mechanism altogether.

- `spell_checker_dictionary` determines the current dictionary, taken from the colon-separated list in the settings variable `JORTHO_DICTIONARIES`. There are jEdit actions to specify local updates to a dictionary, by including or excluding words. The result of permanent dictionary updates is stored in the directory `$ISABELLE_HOME_USER/dictionaries`, in a separate file for each dictionary.

- `spell_checker_elements` specifies a comma-separated list of markup elements that delimit words in the source that is subject to spell-checking, including various forms of comments.

3.8 Automatically tried tools

Continuous document processing works asynchronously in the background. Visible document source that has been evaluated may get augmented by additional results of *asynchronous print functions*. An example for that is proof state output, if that is enabled in the Output panel (§3.2). More heavy-weight print functions may be applied as well, e.g. to prove or disprove parts of the formal text by other means.

Isabelle/HOL provides various automatically tried tools that operate on outermost goal statements (e.g. **lemma**, **theorem**), independently of the state of the current proof attempt. They work implicitly without any arguments. Results are output as *information messages*, which are indicated in the text area by blue squiggles and a blue information sign in the gutter (see figure 3.10). The message content may be shown as for other output (see also §3.2). Some tools produce output with *sendback* markup, which means that clicking on certain parts of the text inserts that into the source in the proper place.

The following Isabelle system options control the behavior of automatically tried tools (see also the jEdit dialog window *Plugin Options / Isabelle / General / Automatically tried tools*):

- `auto_methods` controls automatic use of a combination of standard proof methods (*auto*, *simp*, *blast*, etc.). This corresponds to the Isar command **try0** [5].

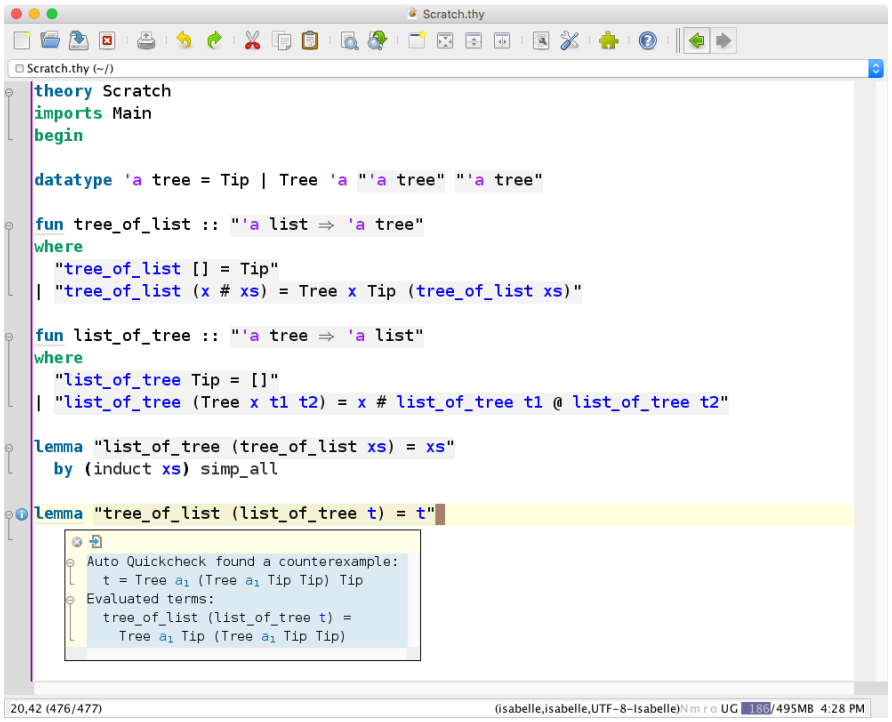


Figure 3.10: Result of automatically tried tools

The tool is disabled by default, since unparameterized invocation of standard proof methods often consumes substantial CPU resources without leading to success.

- `auto_nitpick` controls a slightly reduced version of **nitpick**, which tests for counterexamples using first-order relational logic. See also the Nitpick manual [2].

  This tool is disabled by default, due to the extra overhead of invoking an external Java process for each attempt to disprove a subgoal.

- `auto_quickcheck` controls automatic use of **quickcheck**, which tests for counterexamples using a series of assignments for free variables of a subgoal.

  This tool is *enabled* by default. It requires little overhead, but is a bit weaker than **nitpick**.

- `auto_sledgehammer` controls a significantly reduced version of **sledgehammer**, which attempts to prove a subgoal using external automatic provers. See also the Sledgehammer manual [1].

This tool is disabled by default, due to the relatively heavy nature of Sledgehammer.

- `auto_solve_direct` controls automatic use of **solve_direct**, which checks whether the current subgoals can be solved directly by an existing theorem. This also helps to detect duplicate lemmas.

  This tool is *enabled* by default.

Invocation of automatically tried tools is subject to some global policies of parallel execution, which may be configured as follows:

- `auto_time_limit` (default 2.0) determines the timeout (in seconds) for each tool execution.
- `auto_time_start` (default 1.0) determines the start delay (in seconds) for automatically tried tools, after the main command evaluation is finished.

Each tool is submitted independently to the pool of parallel execution tasks in Isabelle/ML, using hardwired priorities according to its relative "heaviness". The main stages of evaluation and printing of proof states take precedence, but an already running tool is not canceled and may thus reduce reactivity of proof document processing.

Users should experiment how the available CPU resources (number of cores) are best invested to get additional feedback from prover in the background, by using a selection of weaker or stronger tools.

3.9 Sledgehammer

The *Sledgehammer* panel (figure 3.11) provides a view on some independent execution of the Isar command **sledgehammer**, with process indicator (spinning wheel) and GUI elements for important Sledgehammer arguments and options. Any number of Sledgehammer panels may be active, according to the standard policies of Dockable Window Management in jEdit. Closing such windows also cancels the corresponding prover tasks.

The *Apply* button attaches a fresh invocation of **sledgehammer** to the command where the cursor is pointing in the text — this should be some pending proof problem. Further buttons like *Cancel* and *Locate* help to manage the running process.

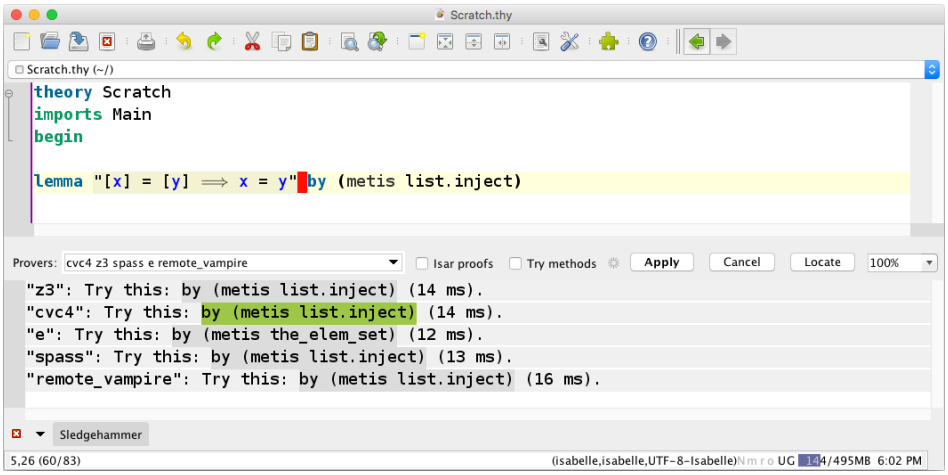


Figure 3.11: An instance of the Sledgehammer panel

Results appear incrementally in the output window of the panel. Proposed proof snippets are marked-up as *sendback*, which means a single mouse click inserts the text into a suitable place of the original source. Some manual editing may be required nonetheless, say to remove earlier proof attempts.

Chapter 4

Isabelle document preparation

The ultimate purpose of Isabelle is to produce nicely rendered documents with the Isabelle document preparation system, which is based on LaTeX; see also [3, 5]. Isabelle/jEdit provides some additional support for document editing.

4.1 Document outline

Theory sources may contain document markup commands, such as **chapter**, **section**, **subsection**. The Isabelle SideKick parser (§2.6) represents this document outline as structured tree view, with formal statements and proofs nested inside; see figure 4.1.

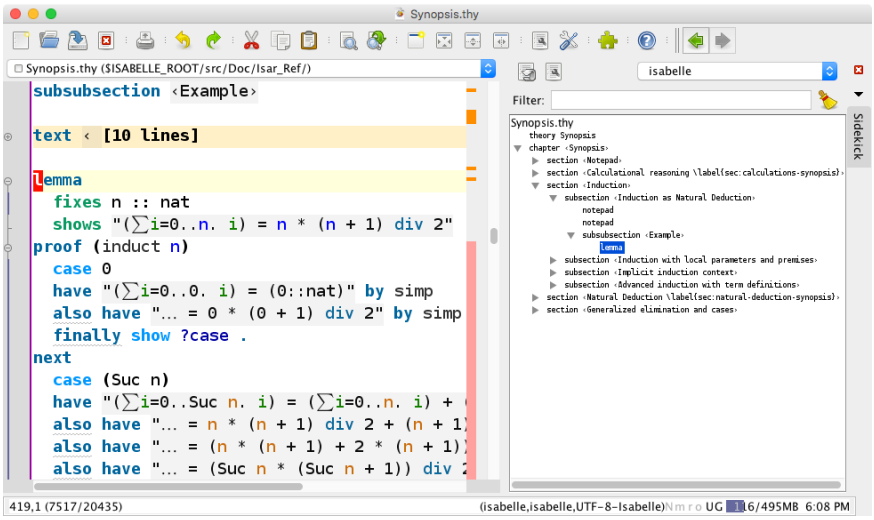


Figure 4.1: Isabelle document outline via SideKick tree view

It is also possible to use text folding according to this structure, by adjusting *Utilities / Buffer Options / Folding mode* of jEdit. The default mode

`isabelle` uses the structure of formal definitions, statements, and proofs. The alternative mode `sidekick` uses the document structure of the SideKick parser, as explained above.

4.2 Markdown structure

Document text is internally structured in paragraphs and nested lists, using notation that is similar to Markdown[1]. There are special control symbols for items of different kinds of lists, corresponding to `itemize`, `enumerate`, `description` in LaTeX. This is illustrated in for `itemize` in figure 4.2.

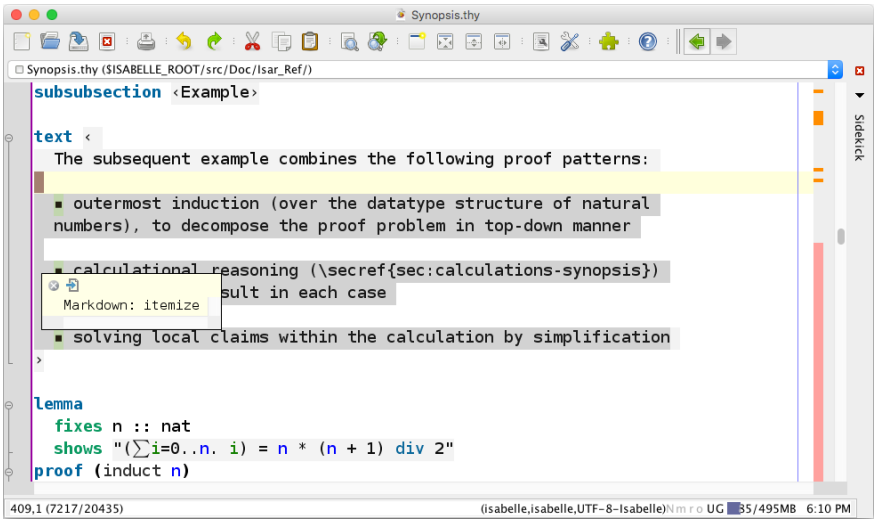


Figure 4.2: Markdown structure within document text

Items take colour according to the depth of nested lists. This helps to explore the implicit rules for list structure interactively. There is also markup for individual paragraphs in the text: it may be explored via mouse hovering with `CONTROL` / `COMMAND` as usual (§3.5).

4.3 Citations and BibTeX entries

Citations are managed by LaTeX and BibTeX in `.bib` files. The Isabelle session build process and the `isabelle latex` tool [3] are smart enough to assemble the result, based on the session directory layout.

[1] http://commonmark.org

The document antiquotation @{*cite*} is described in [5]. Within the Prover IDE it provides semantic markup for tooltips, hyperlinks, and completion for BibTEX database entries. Isabelle/jEdit does *not* know about the actual BibTEX environment used in LATEX batch-mode, but it can take citations from those `.bib` files that happen to be open in the editor; see figure 4.3.

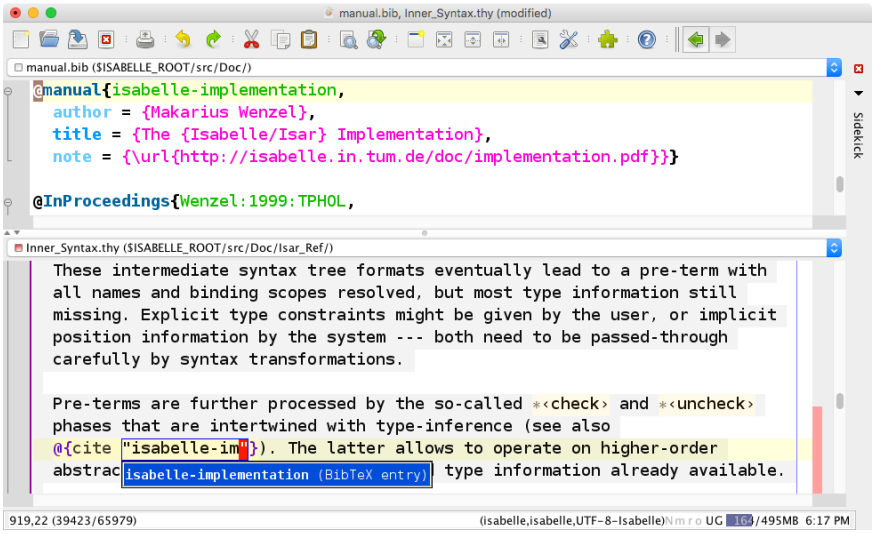


Figure 4.3: Semantic completion of citations from open BibTEX files

Isabelle/jEdit also provides some support for editing `.bib` files themselves. There is syntax highlighting based on entry types (according to standard BibTEX styles), a context-menu to compose entries systematically, and a SideKick tree view of the overall content; see figure 4.4.

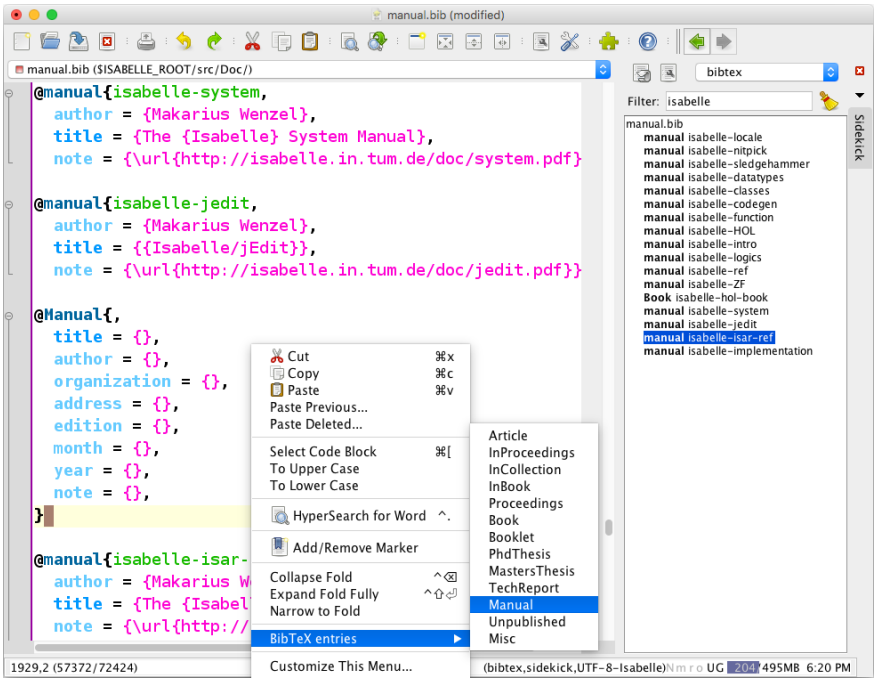


Figure 4.4: BibTEX mode with context menu and SideKick tree view

Chapter 5

ML debugging within the Prover IDE

Isabelle/ML is based on Poly/ML[1] and thus benefits from the source-level debugger of that implementation of Standard ML. The Prover IDE provides the *Debugger* dockable to connect to running ML threads, inspect the stack frame with local ML bindings, and evaluate ML expressions in a particular run-time context. A typical debugger session is shown in figure 5.1.

ML debugging depends on the following pre-requisites.

1. ML source needs to be compiled with debugging enabled. This may be controlled for particular chunks of ML sources using any of the subsequent facilities.

   (a) The system option `ML_debugger` as implicit state of the Isabelle process. It may be changed in the menu *Plugins / Plugin Options / Isabelle / General*. ML modules need to be reloaded and recompiled to pick up that option as intended.

   (b) The configuration option *ML_debugger*, with an attribute of the same name, to update a global or local context (e.g. with the **declare** command).

   (c) Commands that modify *ML_debugger* state for individual files: **ML_file_debug**, **ML_file_no_debug**, **SML_file_debug**, **SML_file_no_debug**.

   The instrumentation of ML code for debugging causes minor run-time overhead. ML modules that implement critical system infrastructure may lead to deadlocks or other undefined behaviour, when put under debugger control!

2. The *Debugger* panel needs to be active, otherwise the program ignores debugger instrumentation of the compiler and runs unmanaged. It is

[1] http://www.polyml.org

also possible to start debugging with the panel open, and later undock it, to let the program continue unhindered.

3. The ML program needs to be stopped at a suitable breakpoint, which may be activated individually or globally as follows.

   For ML sources that have been compiled with debugger support, the IDE visualizes possible breakpoints in the text. A breakpoint may be toggled by pointing accurately with the mouse, with a right-click to activate jEdit's context menu and its *Toggle Breakpoint* item. Alternatively, the *Break* checkbox in the *Debugger* panel may be enabled to stop ML threads always at the next possible breakpoint.

Note that the state of individual breakpoints *gets lost* when the coresponding ML source is re-compiled! This may happen unintentionally, e.g. when following hyperlinks into ML modules that have not been loaded into the IDE before.

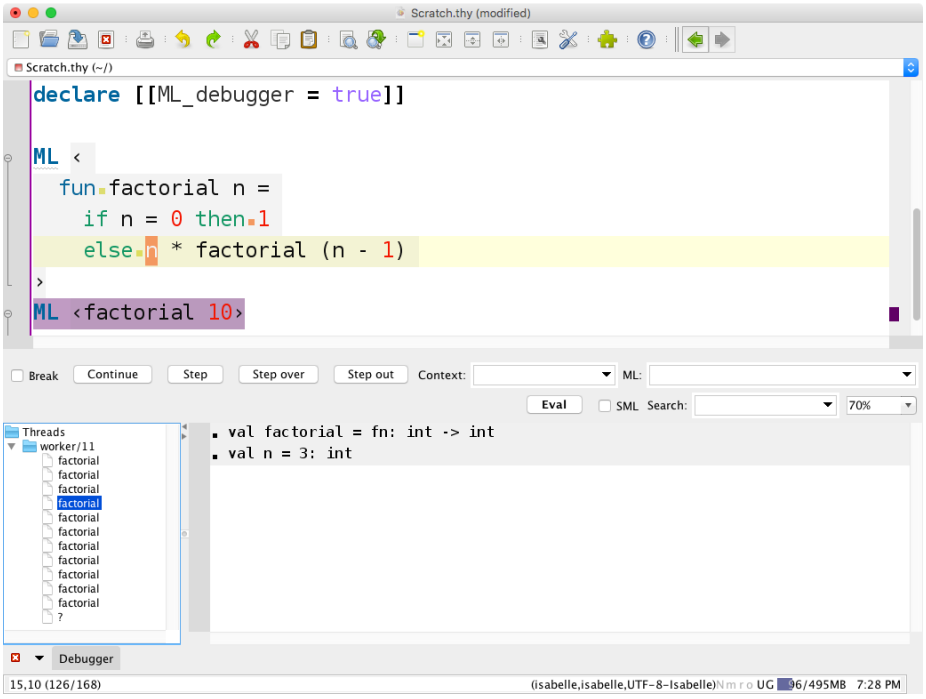


Figure 5.1: ML debugger session

The debugger panel (figure 5.1) shows a list of all threads that are presently stopped. Each thread shows a stack of all function invocations that lead to the current breakpoint at the top.

It is possible to jump between stack positions freely, by clicking on this list. The current situation is displayed in the big output window, as a local ML environment with names and printed values.

ML expressions may be evaluated in the current context by entering snippets of source into the text fields labeled *Context* and *ML*, and pushing the *Eval* button. By default, the source is interpreted as Isabelle/ML with the usual support for antiquotations (like **ML**, **ML_file**). Alternatively, strict Standard ML may be enforced via the *SML* checkbox (like **SML_file**).

The context for Isabelle/ML is optional, it may evaluate to a value of type `theory`, `Proof.context`, or `Context.generic`. Thus the given ML expression (with its antiquotations) may be subject to the intended dynamic runtime context, instead of the static compile-time context.

The buttons labeled *Continue*, *Step*, *Step over*, *Step out* recommence execution of the program, with different policies concerning nested function invocations. The debugger always moves the cursor within the ML source to the next breakpoint position, and offers new stack frames as before.

Chapter 6

Miscellaneous tools

6.1 Timing

Managed evaluation of commands within PIDE documents includes timing information, which consists of elapsed (wall-clock) time, CPU time, and GC (garbage collection) time. Note that in a multithreaded system it is difficult to measure execution time precisely: elapsed time is closer to the real requirements of runtime resources than CPU or GC time, which are both subject to influences from the parallel environment that are outside the scope of the current command transaction.

The *Timing* panel provides an overview of cumulative command timings for each document node. Commands with elapsed time below the given threshold are ignored in the grand total. Nodes are sorted according to their overall timing. For the document node that corresponds to the current buffer, individual command timings are shown as well. A double-click on a theory node or command moves the editor focus to that particular source position.

It is also possible to reveal individual timing information via some tooltip for the corresponding command keyword, using the technique of mouse hovering with `CONTROL` / `COMMAND` modifier (§3.5). Actual display of timing depends on the global option `jedit_timing_threshold`, which can be configured in *Plugin Options / Isabelle / General*.

The *Monitor* panel visualizes various data collections about recent activity of the Isabelle/ML task farm and the underlying ML runtime system. The display is continuously updated according to `editor_chart_delay`. Note that the painting of the chart takes considerable runtime itself — on the Java Virtual Machine that runs Isabelle/Scala, not Isabelle/ML. Internally, the Isabelle/Scala module `isabelle.ML_Statistics` provides further access to statistics of Isabelle/ML.

6.2 Low-level output

Prover output is normally shown directly in the main text area or specific panels like *Output* (§3.2) or *State* (§3.3). Beyond this, it is occasionally useful to inspect low-level output channels via some of the following additional panels:

- *Protocol* shows internal messages between the Isabelle/Scala and Isabelle/ML side of the PIDE document editing protocol. Recording of messages starts with the first activation of the corresponding dockable window; earlier messages are lost.

  Actual display of protocol messages causes considerable slowdown, so it is important to undock all *Protocol* panels for production work.

- *Raw Output* shows chunks of text from the `stdout` and `stderr` channels of the prover process. Recording of output starts with the first activation of the corresponding dockable window; earlier output is lost.

  The implicit stateful nature of physical I/O channels makes it difficult to relate raw output to the actual command from where it was originating. Parallel execution may add to the confusion. Peeking at physical process I/O is only the last resort to diagnose problems with tools that are not PIDE compliant.

  Under normal circumstances, prover output always works via managed message channels (corresponding to `writeln`, `warning`, `Output.error_message` in Isabelle/ML), which are displayed by regular means within the document model (§3.2). Unhandled Isabelle/ML exceptions are printed by the system via `Output.error_message`.

- *Syslog* shows system messages that might be relevant to diagnose problems with the startup or shutdown phase of the prover process; this also includes raw output on `stderr`. Isabelle/ML also provides an explicit `Output.system_message` operation, which is occasionally useful for diagnostic purposes within the system infrastructure itself.

  A limited amount of syslog messages are buffered, independently of the docking state of the *Syslog* panel. This allows to diagnose serious problems with Isabelle/PIDE process management, outside of the actual protocol layer.

  Under normal situations, such low-level system output can be ignored.

Chapter 7

Known problems and workarounds

- **Problem:** Odd behavior of some diagnostic commands with global side-effects, like writing a physical file.

  Workaround: Copy/paste complete command text from elsewhere, or disable continuous checking temporarily.

- **Problem:** No direct support to remove document nodes from the collection of theories.

  Workaround: Clear the buffer content of unused files and close *without* saving changes.

- **Problem:** Keyboard shortcuts `C+PLUS` and `C+MINUS` for adjusting the editor font size depend on platform details and national keyboards.

  Workaround: Rebind keys via *Global Options / Shortcuts*.

- **Problem:** The Mac OS X key sequence `COMMAND+COMMA` for application *Preferences* is in conflict with the jEdit default keyboard shortcut for *Incremental Search Bar* (action `quick-search`).

  Workaround: Rebind key via *Global Options / Shortcuts* according to national keyboard, e.g. `COMMAND+SLASH` on English ones.

- **Problem:** On Mac OS X with native Apple look-and-feel, some exotic national keyboards may cause a conflict of menu accelerator keys with regular jEdit key bindings. This leads to duplicate execution of the corresponding jEdit action.

  Workaround: Disable the native Apple menu bar via Java runtime option `-Dapple.laf.useScreenMenuBar=false`.

- **Problem:** Mac OS X system fonts sometimes lead to character drop-outs in the main text area.

  Workaround: Use the default `IsabelleText` font.

- **Problem:** Mac OS X with Retina display has problems to determine the font metrics of `IsabelleText` accurately, notably in plain Swing text fields (e.g. in the *Search and Replace* dialog).

  Workaround: Install `IsabelleText` and `IsabelleTextBold` on the system with *Font Book*, despite the warnings in §2.2 against that! The `.ttf` font files reside in some directory `$ISABELLE_HOME/contrib/isabelle_fonts-XYZ`.

- **Problem:** Some Linux/X11 input methods such as IBus tend to disrupt key event handling of Java/AWT/Swing.

  Workaround: Do not use X11 input methods. Note that environment variable `XMODIFIERS` is reset by default within Isabelle settings.

- **Problem:** Some Linux/X11 window managers that are not "re-parenting" cause problems with additional windows opened by Java. This affects either historic or neo-minimalistic window managers like `awesome` or `xmonad`.

  Workaround: Use a regular re-parenting X11 window manager.

- **Problem:** Various forks of Linux/X11 window managers and desktop environments (like Gnome) disrupt the handling of menu popups and mouse positions of Java/AWT/Swing.

  Workaround: Use suitable version of Linux desktops.

- **Problem:** Full-screen mode via jEdit action `toggle-full-screen` (default keyboard shortcut `F11`) works on Windows, but not on Mac OS X or various Linux/X11 window managers.

  Workaround: Use native full-screen control of the window manager (notably on Mac OS X).

- **Problem:** Heap space of the JVM may fill up and render the Prover IDE unresponsive, e.g. when editing big Isabelle sessions with many theories.

  Workaround: On a 64bit platform, ensure that the JVM runs in 64bit mode, but the Isabelle/ML process remains in 32bit mode! Do not switch Isabelle/ML into 64bit mode in the expectation to be "more efficient" — this requires approx. 32 GB to make sense.

  For the JVM, always use the 64bit version. That is the default on all platforms, except for Windows: the standard download is for win32,

but there is a separate download for win64. This implicitly provides a larger default heap for the JVM.

Moreover, it is possible to increase JVM heap parameters explicitly, by editing platform-specific files (for "properties" or "options") that are associated with the main app bundle.

Also note that jEdit provides a heap space monitor in the status line (bottom-right). Double-clicking on that causes full garbage-collection, which sometimes helps in low-memory situations.

Bibliography

[1] J. C. Blanchette. *Hammering Away: A User's Guide to Sledgehammer for Isabelle/HOL.* http://isabelle.in.tum.de/doc/sledgehammer.pdf.

[2] J. C. Blanchette. *Picking Nits: A User's Guide to Nitpick for Isabelle/HOL.* http://isabelle.in.tum.de/doc/nitpick.pdf.

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

[4] M. Wenzel. *The Isabelle/Isar Implementation.* http://isabelle.in.tum.de/doc/implementation.pdf.

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

[6] M. Wenzel. Parallel proof checking in Isabelle/Isar. In G. Dos Reis and L. Théry, editors, *ACM SIGSAM Workshop on Programming Languages for Mechanized Mathematics Systems (PLMMS 2009).* ACM Digital Library, 2009.

[7] M. Wenzel. Asynchronous proof processing with Isabelle/Scala and Isabelle/jEdit. In C. S. Coen and D. Aspinall, editors, *User Interfaces for Theorem Provers (UITP 2010), FLOC 2010 Satellite Workshop*, ENTCS. Elsevier, July 2010.

[8] M. Wenzel. Isabelle as document-oriented proof assistant. In J. H. Davenport, W. M. Farmer, F. Rabe, and J. Urban, editors, *Conference on Intelligent Computer Mathematics / Mathematical Knowledge Management (CICM/MKM 2011)*, volume 6824 of *LNAI*. Springer, 2011.

[9] M. Wenzel. Isabelle/jEdit — a Prover IDE within the PIDE framework. In J. Jeuring et al., editors, *Conference on Intelligent Computer Mathematics (CICM 2012)*, volume 7362 of *LNAI*. Springer, 2012.

[10] M. Wenzel. READ-EVAL-PRINT in parallel and asynchronous proof-checking. In *User Interfaces for Theorem Provers (UITP 2012)*, EPTCS, 2013.

[11] M. Wenzel. Shared-memory multiprocessing for interactive theorem proving. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving — 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *Lecture Notes in Computer Science*. Springer, 2013.

[12] 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.

[13] M. Wenzel. System description: Isabelle/jEdit in 2014. In C. Benzmüller and B. Woltzenlogel Paleo, editors, *User Interfaces for Theorem Provers (UITP 2014)*, EPTCS, July 2014. http://eptcs.web.cse.unsw.edu.au/paper.cgi?UITP2014:11.

Index