

The Isabelle System Manual

Makarius Wenzel

9 June 2019

Contents

1	The Isabelle system environment	1
1.1	Isabelle settings	1
1.1.1	Bootstrapping the environment	1
1.1.2	Common variables	2
1.1.3	Additional components	5
1.2	The Isabelle tool wrapper	6
1.3	The raw Isabelle ML process	8
1.3.1	Batch mode	8
1.3.2	Interactive mode	9
1.4	The raw Isabelle Java process	10
1.5	YXML versus XML	10
2	Isabelle sessions and build management	12
2.1	Session ROOT specifications	12
2.2	System build options	17
2.3	Invoking the build process	19
2.4	Maintain theory imports wrt. session structure	23
2.5	Retrieve theory exports	24
2.6	Dump PIDE session database	25
2.7	Update theory sources based on PIDE markup	27
3	Presenting theories	30
3.1	Generating HTML browser information	30
3.2	Preparing session root directories	31
3.3	Preparing Isabelle session documents	32
3.4	Running L ^A T _E X within the Isabelle environment	34
4	The Isabelle server	36
4.1	Command-line tools	36

4.1.1	Server	36
4.1.2	Client	37
4.1.3	Examples	38
4.2	Protocol messages	39
4.2.1	Byte messages	39
4.2.2	Text messages	40
4.2.3	Input and output messages	40
4.2.4	Initial password exchange	41
4.2.5	Synchronous commands	41
4.2.6	Asynchronous commands	42
4.3	Types for JSON values	42
4.4	Server commands and results	47
4.4.1	Command <code>help</code>	47
4.4.2	Command <code>echo</code>	48
4.4.3	Command <code>shutdown</code>	48
4.4.4	Command <code>cancel</code>	48
4.4.5	Command <code>session_build</code>	49
4.4.6	Command <code>session_start</code>	51
4.4.7	Command <code>session_stop</code>	52
4.4.8	Command <code>use_theories</code>	53
4.4.9	Command <code>purge_theories</code>	56
5	Isabelle/Scala development tools	58
5.1	Java Runtime Environment within Isabelle	58
5.2	Scala toplevel	58
5.3	Scala compiler	59
5.4	Scala script wrapper	59
6	Miscellaneous tools	60
6.1	Resolving Isabelle components	60
6.2	Displaying documents	61
6.3	Viewing documentation	61
6.4	Shell commands within the settings environment	61
6.5	Inspecting the settings environment	62
6.6	Installing standalone Isabelle executables	63

<i>CONTENTS</i>	iii
6.7 Creating instances of the Isabelle logo	63
6.8 Output the version identifier of the Isabelle distribution	64
Bibliography	65
Index	66

The Isabelle system environment

This manual describes Isabelle together with related tools as seen from a system oriented view. See also the *Isabelle/Isar Reference Manual* [2] for the actual Isabelle input language and related concepts, and *The Isabelle/Isar Implementation Manual* [1] for the main concepts of the underlying implementation in Isabelle/ML.

1.1 Isabelle settings

Isabelle executables may depend on the *Isabelle settings* within the process environment. This is a statically scoped collection of environment variables, such as `ISABELLE_HOME`, `ML_SYSTEM`, `ML_HOME`. These variables are *not* intended to be set directly from the shell, but are provided by Isabelle *components* their *settings files* as explained below.

1.1.1 Bootstrapping the environment

Isabelle executables need to be run within a proper settings environment. This is bootstrapped as described below, on the first invocation of one of the outer wrapper scripts (such as `isabelle`). This happens only once for each process tree, i.e. the environment is passed to subprocesses according to regular Unix conventions.

1. The special variable `ISABELLE_HOME` is determined automatically from the location of the binary that has been run.

You should not try to set `ISABELLE_HOME` manually. Also note that the Isabelle executables either have to be run from their original location in the distribution directory, or via the executable objects created by the `isabelle install` tool. Symbolic links are admissible, but a plain copy of the `$ISABELLE_HOME/bin` files will not work!

2. The file `$ISABELLE_HOME/etc/settings` is run as a `bash` shell script with the `auto-export` option for variables enabled.

This file holds a rather long list of shell variable assignments, thus providing the site-wide default settings. The Isabelle distribution already contains a global settings file with sensible defaults for most variables. When installing the system, only a few of these may have to be adapted (probably `ML_SYSTEM` etc.).

3. The file `$ISABELLE_HOME_USER/etc/settings` (if it exists) is run in the same way as the site default settings. Note that the variable `ISABELLE_HOME_USER` has already been set before — usually to something like `$USER_HOME/.isabelle/Isabelle2018`.

Thus individual users may override the site-wide defaults. Typically, a user settings file contains only a few lines, with some assignments that are actually changed. Never copy the central `$ISABELLE_HOME/etc/settings` file!

Since settings files are regular GNU `bash` scripts, one may use complex shell commands, such as `if` or `case` statements to set variables depending on the system architecture or other environment variables. Such advanced features should be added only with great care, though. In particular, external environment references should be kept at a minimum.

A few variables are somewhat special, e.g. `ISABELLE_TOOL` is set automatically to the absolute path name of the `isabelle` executables.

Note that the settings environment may be inspected with the `isabelle getenv` tool. This might help to figure out the effect of complex settings scripts.

1.1.2 Common variables

This is a reference of common Isabelle settings variables. Note that the list is somewhat open-ended. Third-party utilities or interfaces may add their own selection. Variables that are special in some sense are marked with `*`.

`USER_HOME*` Is the cross-platform user home directory. On Unix systems this is usually the same as `HOME`, but on Windows it is the regular home directory of the user, not the one of within the Cygwin root file-system.¹

¹Cygwin itself offers another choice whether its `HOME` should point to the `/home` directory tree or the Windows user home.

`ISABELLE_HOME*` is the location of the top-level Isabelle distribution directory. This is automatically determined from the Isabelle executable that has been invoked. Do not attempt to set `ISABELLE_HOME` yourself from the shell!

`ISABELLE_HOME_USER` is the user-specific counterpart of `ISABELLE_HOME`. The default value is relative to `$USER_HOME/.isabelle`, under rare circumstances this may be changed in the global setting file. Typically, the `ISABELLE_HOME_USER` directory mimics `ISABELLE_HOME` to some extent. In particular, site-wide defaults may be overridden by a private `$ISABELLE_HOME_USER/etc/settings`.

`ISABELLE_PLATFORM_FAMILY*` is automatically set to the general platform family: `linux`, `macos`, `windows`. Note that platform-dependent tools usually need to refer to the more specific identification according to `ISABELLE_PLATFORM64`, `ISABELLE_PLATFORM32`, `ISABELLE_WINDOWS_PLATFORM64`, `ISABELLE_WINDOWS_PLATFORM32`.

`ISABELLE_PLATFORM64*`, `ISABELLE_PLATFORM32*` indicate the standard Posix platform: `x86_64` for 64 bit and `x86` for 32 bit, together with a symbolic name for the operating system (`linux`, `darwin`, `cygwin`). All platforms support 64 bit executables, some platforms also support 32 bit executables.

In GNU bash scripts, it is possible to use the following expressions (with quotes) to specify a preference of 64 bit over 32 bit:

```
"${ISABELLE_PLATFORM64:-$ISABELLE_PLATFORM32}"
```

In contrast, the subsequent expression prefers the old 32 bit variant (which is only relevant for unusual applications):

```
"${ISABELLE_PLATFORM32:-$ISABELLE_PLATFORM64}"
```

`ISABELLE_WINDOWS_PLATFORM64*`, `ISABELLE_WINDOWS_PLATFORM32*` indicate the native Windows platform. These settings are analogous (but independent) of those for the standard Posix subsystem: `ISABELLE_PLATFORM64`, `ISABELLE_PLATFORM32`.

In GNU bash scripts, a preference for native Windows platform variants may be specified like this (first 64 bit, second 32 bit):

```
"${ISABELLE_WINDOWS_PLATFORM64:-${ISABELLE_WINDOWS_PLATFORM32:-
  ${ISABELLE_PLATFORM64:-$ISABELLE_PLATFORM32}}}"
```


`ISABELLE_TOOL*` is automatically set to the full path name of the `isabelle` executable.

`ISABELLE_IDENTIFIER*` refers to the name of this Isabelle distribution, e.g. “Isabelle2018”.

`ML_SYSTEM`, `ML_HOME`, `ML_OPTIONS`, `ML_PLATFORM`, `ML_IDENTIFIER*` specify the underlying ML system to be used for Isabelle. There is only a fixed set of admissible `ML_SYSTEM` names (see the `$ISABELLE_HOME/etc/settings` file of the distribution).

The actual compiler binary will be run from the directory `ML_HOME`, with `ML_OPTIONS` as first arguments on the command line. The optional `ML_PLATFORM` may specify the binary format of ML heap images, which is useful for cross-platform installations. The value of `ML_IDENTIFIER` is automatically obtained by composing the values of `ML_SYSTEM`, `ML_PLATFORM` and the Isabelle version values.

`ISABELLE_JDK_HOME` points to a full JDK (Java Development Kit) installation with `javac` and `jar` executables. Note that conventional `JAVA_HOME` points to the JRE (Java Runtime Environment), not the JDK.

`ISABELLE_JAVA_PLATFORM` identifies the hardware and operating system platform for the Java installation of Isabelle. That is always the (native) 64 bit variant: `x86_64-linux`, `x86_64-darwin`, `x86_64-windows`.

`ISABELLE_BROWSER_INFO` is the directory where HTML and PDF browser information is stored (see also §3.1); its default is `$ISABELLE_HOME_USER/browser_info`. For “system build mode” (see §2.3), `ISABELLE_BROWSER_INFO_SYSTEM` is used instead; its default is `$ISABELLE_HOME/browser_info`.

`ISABELLE_HEAPS` is the directory where session heap images, log files, and build databases are stored; its default is `$ISABELLE_HOME_USER/heaps`. If `system_heaps` is true, `ISABELLE_HEAPS_SYSTEM` is used instead; its default is `$ISABELLE_HOME/heaps`. See also §2.3.

`ISABELLE_LOGIC` specifies the default logic to load if none is given explicitly by the user. The default value is `HOL`.

`ISABELLE_LINE_EDITOR` specifies the line editor for the `isabelle` console interface.

ISABELLE_LATEX, ISABELLE_PDFLATEX, ISABELLE_BIBTEX refer to \LaTeX related tools for Isabelle document preparation (see also §3.4).

ISABELLE_TOOLS is a colon separated list of directories that are scanned by `isabelle` for external utility programs (see also §1.2).

ISABELLE_DOCS is a colon separated list of directories with documentation files.

PDF_VIEWER specifies the program to be used for displaying `pdf` files.

DVI_VIEWER specifies the program to be used for displaying `dvi` files.

ISABELLE_TMP_PREFIX* is the prefix from which any running Isabelle ML process derives an individual directory for temporary files.

ISABELLE_TOOL_JAVA_OPTIONS is passed to the `java` executable when running Isabelle tools (e.g. `isabelle build`). This is occasionally helpful to provide more heap space, via additional options like `-Xms1g -Xmx4g`.

1.1.3 Additional components

Any directory may be registered as an explicit *Isabelle component*. The general layout conventions are that of the main Isabelle distribution itself, and the following two files (both optional) have a special meaning:

- `etc/settings` holds additional settings that are initialized when bootstrapping the overall Isabelle environment, cf. §1.1.1. As usual, the content is interpreted as a GNU bash script. It may refer to the component's enclosing directory via the `COMPONENT` shell variable.

For example, the following setting allows to refer to files within the component later on, without having to hardwire absolute paths:

```
MY_COMPONENT_HOME="$COMPONENT"
```

Components can also add to existing Isabelle settings such as `ISABELLE_TOOLS`, in order to provide component-specific tools that can be invoked by end-users. For example:

```
ISABELLE_TOOLS="$ISABELLE_TOOLS:$COMPONENT/lib/Tools"
```

- `etc/components` holds a list of further sub-components of the same structure. The directory specifications given here can be either absolute (with leading `/`) or relative to the component's main directory.

The root of component initialization is `ISABELLE_HOME` itself. After initializing all of its sub-components recursively, `ISABELLE_HOME_USER` is included in the same manner (if that directory exists). This allows to install private components via `$ISABELLE_HOME_USER/etc/components`, although it is often more convenient to do that programmatically via the `init_component` shell function in the `etc/settings` script of `$ISABELLE_HOME_USER` (or any other component directory). For example:

```
init_component "$HOME/screwdriver-2.0"
```

This is tolerant wrt. missing component directories, but might produce a warning.

More complex situations may be addressed by initializing components listed in a given catalog file, relatively to some base directory:

```
init_components "$HOME/my_component_store" "some_catalog_file"
```

The component directories listed in the catalog file are treated as relative to the given base directory.

See also §6.1 for some tool-support for resolving components that are formally initialized but not installed yet.

1.2 The Isabelle tool wrapper

The main *Isabelle tool wrapper* provides a generic startup environment for Isabelle-related utilities, user interfaces, add-on applications etc. Such tools automatically benefit from the settings mechanism (§1.1). Moreover, this is the standard way to invoke Isabelle/Scala functionality as a separate operating-system process. Isabelle command-line tools are run uniformly via a common wrapper — `isabelle`:

```
Usage: isabelle TOOL [ARGS ...]
```

Start Isabelle TOOL with ARGS; pass `"-?"` for tool-specific help.

Available tools:

...

Tools may be implemented in Isabelle/Scala or as stand-alone executables (usually as GNU bash scripts). In the invocation of “`isabelle tool`”, the named *tool* is resolved as follows (and in the given order).

1. An external tool found on the directories listed in the `ISABELLE_TOOLS` settings variable (colon-separated list in standard POSIX notation).
 - (a) If a file “`tool.scala`” is found, the source needs to define some object that extends the class `Isabelle_Tool.Body`. The Scala compiler is invoked on the spot (which may take some time), and the body function is run with the command-line arguments as `List[String]`.
 - (b) If an executable file “`tool`” is found, it is invoked as stand-alone program with the command-line arguments provided as `argv` array.
2. An internal tool that is registered in `Isabelle_Tool.internal_tools` within the Isabelle/Scala namespace of `Pure.jar`. This is the preferred entry-point for high-end tools implemented in Isabelle/Scala — compiled once when the Isabelle distribution is built. These tools are provided by Isabelle/Pure and cannot be augmented in user-space.

There are also some administrative tools that are available from a bare repository clone of Isabelle, but not in regular distributions.

Examples

Show the list of available documentation of the Isabelle distribution:

```
isabelle doc
```

View a certain document as follows:

```
isabelle doc system
```

Query the Isabelle settings environment:

```
isabelle getenv ISABELLE_HOME_USER
```

1.3 The raw Isabelle ML process

1.3.1 Batch mode

The `isabelle process` tool runs the raw ML process in batch mode:

Usage: `isabelle process [OPTIONS]`

Options are:

```
-T THEORY    load theory
-d DIR       include session directory
-e ML_EXPR   evaluate ML expression on startup
-f ML_FILE   evaluate ML file on startup
-l NAME      logic session name (default ISABELLE_LOGIC="HOL")
-m MODE      add print mode for output
-o OPTION    override Isabelle system OPTION (via NAME=VAL or NAME)
```

Run the raw Isabelle ML process in batch mode.

Options `-e` and `-f` allow to evaluate ML code, before the ML process is started. The source is either given literally or taken from a file. Multiple `-e` and `-f` options are evaluated in the given order. Errors lead to premature exit of the ML process with return code 1.

Option `-T` loads a specified theory file. This is a wrapper for `-e` with a suitable `use_thy` invocation.

Option `-l` specifies the logic session name. Option `-d` specifies additional directories for session roots, see also §2.3.

The `-m` option adds identifiers of print modes to be made active for this session. For example, `-m ASCII` prefers ASCII replacement syntax over mathematical Isabelle symbols.

Option `-o` allows to override Isabelle system options for this process, see also §2.2.

Examples

The subsequent example retrieves the `Main` theory value from the theory loader within ML:

```
isabelle process -e 'Thy_Info.get_theory "Main"'
```

Observe the delicate quoting rules for the GNU bash shell vs. ML. The Isabelle/ML and Scala libraries provide functions for that, but here we need to do it manually.

This is how to invoke a function body with proper return code and printing of errors, and without printing of a redundant `val it = (): unit` result:

```
isabelle process -e 'Command_Line.tool0 (fn () => writeln "OK")'
```

```
isabelle process -e 'Command_Line.tool0 (fn () => error "Bad")'
```

1.3.2 Interactive mode

The `isabelle console` tool runs the raw ML process with interactive console and line editor:

Usage: `isabelle console [OPTIONS]`

Options are:

```
-d DIR          include session directory
-i NAME        include session in name-space of theories
-l NAME        logic session name (default ISABELLE_LOGIC)
-m MODE        add print mode for output
-n            no build of session image on startup
-o OPTION      override Isabelle system OPTION (via NAME=VAL or NAME)
-r            bootstrap from raw Poly/ML
```

Build a logic session image and run the raw Isabelle ML process in interactive mode, with line editor `ISABELLE_LINE_EDITOR`.

Option `-l` specifies the logic session name. By default, its heap image is checked and built on demand, but the option `-n` skips that.

Option `-i` includes additional sessions into the name-space of theories: multiple occurrences are possible.

Option `-r` indicates a bootstrap from the raw Poly/ML system, which is relevant for Isabelle/Pure development.

Options `-d`, `-m`, `-o` have the same meaning as for `isabelle process` (§1.3.1).

The Isabelle/ML process is run through the line editor that is specified via the settings variable `ISABELLE_LINE_EDITOR` (e.g. `rlwrap` for GNU readline); the fall-back is to use plain standard input/output.

The user is connected to the raw ML toplevel loop: this is neither Isabelle/Isar nor Isabelle/ML within the usual formal context. The most relevant ML commands at this stage are `use` (for ML files) and `use_thy` (for theory files).

1.4 The raw Isabelle Java process

The `isabelle_java` executable allows to run a Java process within the name space of Java and Scala components that are bundled with Isabelle, but *without* the Isabelle settings environment (§1.1).

After such a JVM cold-start, the Isabelle environment can be accessed via `Isabelle_System.getenv` as usual, but the underlying process environment remains clean. This is e.g. relevant when invoking other processes that should remain separate from the current Isabelle installation.

Note that under normal circumstances, Isabelle command-line tools are run *within* the settings environment, as provided by the `isabelle` wrapper (§1.2 and §5.1).

Example

The subsequent example creates a raw Java process on the command-line and invokes the main Isabelle application entry point:

```
isabelle_java isabelle.Main
```

1.5 YXML versus XML

Isabelle tools often use YXML, which is a simple and efficient syntax for untyped XML trees. The YXML format is defined as follows.

1. The encoding is always UTF-8.
2. Body text is represented verbatim (no escaping, no special treatment of white space, no named entities, no CDATA chunks, no comments).
3. Markup elements are represented via ASCII control characters **X** = 5 and **Y** = 6 as follows:

XML	YXML
<code><name attribute=value ...></code>	<code>XYnameYattribute=value...X</code>
<code></name></code>	<code>XYX</code>

There is no special case for empty body text, i.e. `<foo/>` is treated like `<foo></foo>`. Also note that **X** and **Y** may never occur in well-formed XML documents.

Parsing YXML is pretty straight-forward: split the text into chunks separated by **X**, then split each chunk into sub-chunks separated by **Y**. Markup chunks start with an empty sub-chunk, and a second empty sub-chunk indicates close of an element. Any other non-empty chunk consists of plain text. For example, see `~/src/Pure/PIDE/yxml.ML` or `~/src/Pure/PIDE/yxml.scala`.

YXML documents may be detected quickly by checking that the first two characters are **XY**.

Isabelle sessions and build management

An Isabelle *session* consists of a collection of related theories that may be associated with formal documents (chapter 3). There is also a notion of *persistent heap* image to capture the state of a session, similar to object-code in compiled programming languages. Thus the concept of session resembles that of a “project” in common IDE environments, but the specific name emphasizes the connection to interactive theorem proving: the session wraps-up the results of user-interaction with the prover in a persistent form.

Application sessions are built on a given parent session, which may be built recursively on other parents. Following this path in the hierarchy eventually leads to some major object-logic session like *HOL*, which itself is based on *Pure* as the common root of all sessions.

Processing sessions may take considerable time. Isabelle build management helps to organize this efficiently. This includes support for parallel build jobs, in addition to the multithreaded theory and proof checking that is already provided by the prover process itself.

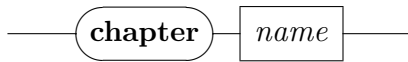
2.1 Session ROOT specifications

Session specifications reside in files called `ROOT` within certain directories, such as the home locations of registered Isabelle components or additional project directories given by the user.

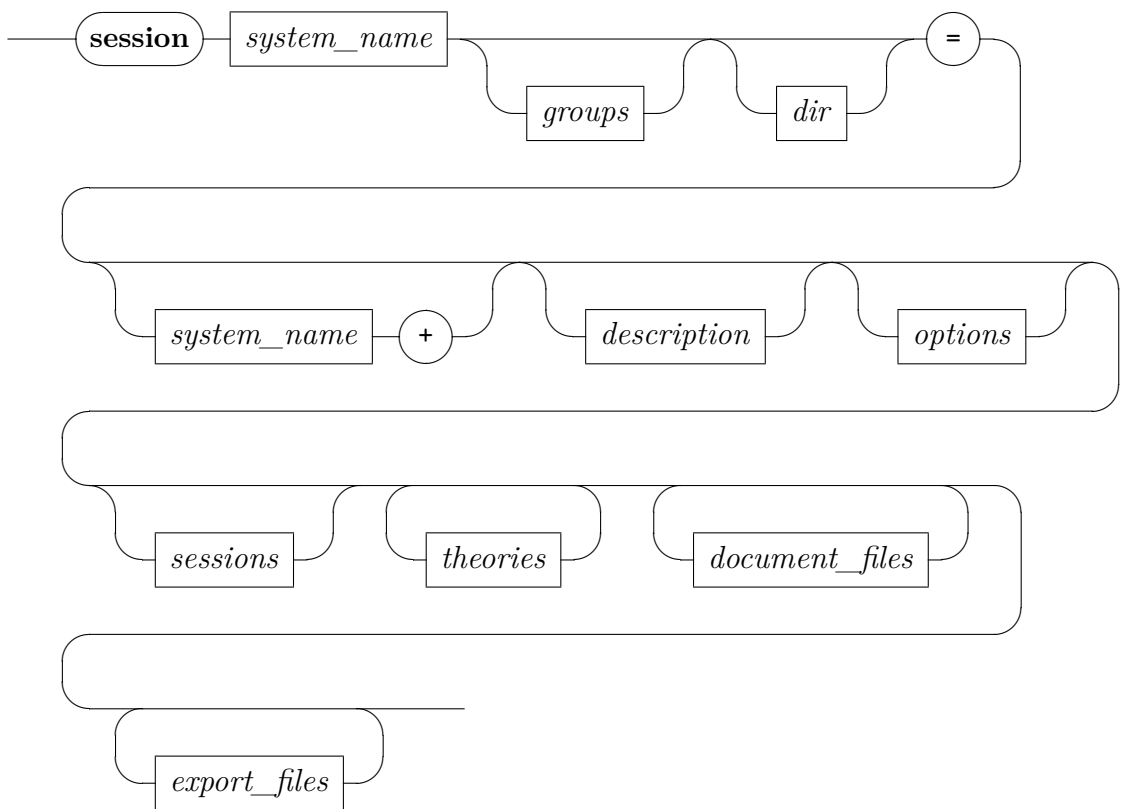
The `ROOT` file format follows the lexical conventions of the *outer syntax* of Isabelle/Isar, see also [2]. This defines common forms like identifiers, names, quoted strings, verbatim text, nested comments etc. The grammar for *session_chapter* and *session_entry* is given as syntax diagram below; each `ROOT` file may contain multiple specifications like this. Chapters help to organize browser info (§3.1), but have no formal meaning. The default chapter is “*Unsorted*”.

Isabelle/jEdit [3] includes a simple editing mode `isabelle-root` for session ROOT files, which is enabled by default for any file of that name.

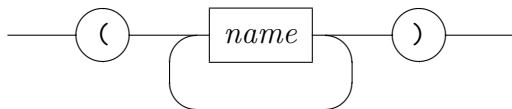
session_chapter



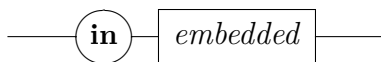
session_entry



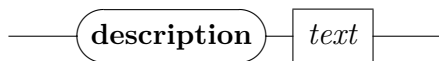
groups



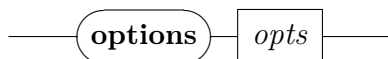
dir



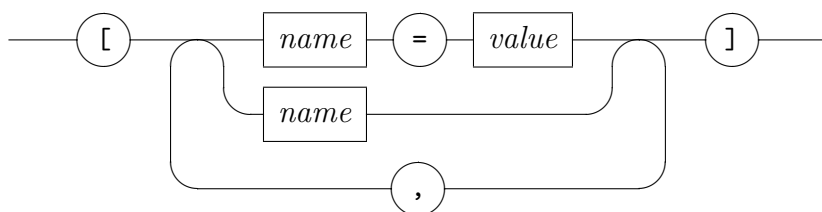
description



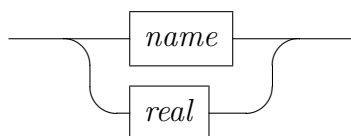
options



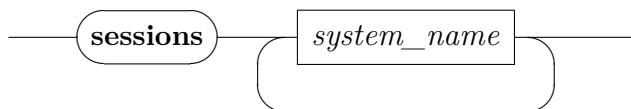
opts



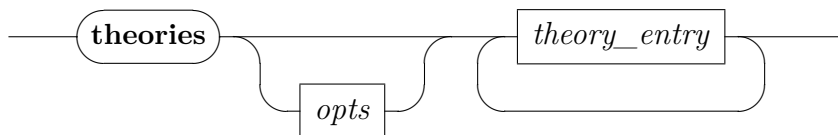
value



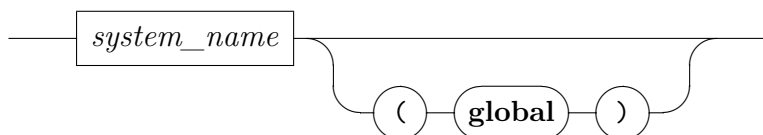
sessions



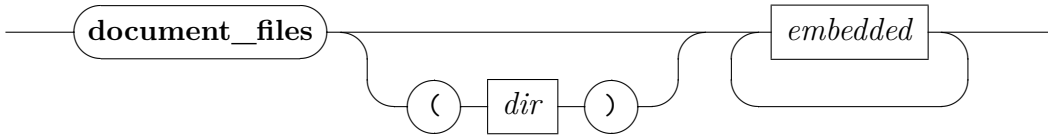
theories



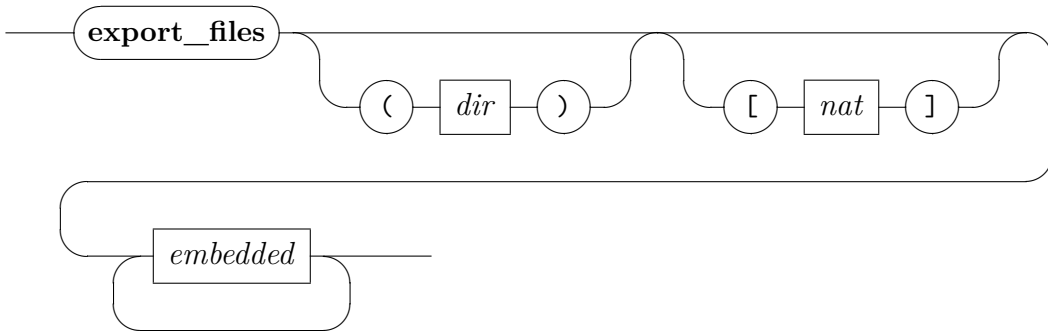
theory_entry



document_files



export_files



session $A = B + body$ defines a new session A based on parent session B , with its content given in *body* (imported sessions and theories). Note that a parent (like *HOL*) is mandatory in practical applications: only Isabelle/Pure can bootstrap itself from nothing.

All such session specifications together describe a hierarchy (graph) of sessions, with globally unique names. The new session name A should be sufficiently long and descriptive to stand on its own in a potentially large library.

session A (*groups*) indicates a collection of groups where the new session is a member. Group names are uninterpreted and merely follow certain conventions. For example, the Isabelle distribution tags some important sessions by the group name called “*main*”. Other projects may invent their own conventions, but this requires some care to avoid clashes within this unchecked name space.

session A **in** *dir* specifies an explicit directory for this session; by default this is the current directory of the **ROOT** file.

All theory files are located relatively to the session directory. The prover process is run within the same as its current working directory.

description *text* is a free-form annotation for this session.

options $[x = a, y = b, z]$ defines separate options (§2.2) that are used when processing this session, but *without* propagation to child sessions. Note that z abbreviates $z = true$ for Boolean options.

sessions *names* specifies sessions that are *imported* into the current name space of theories. This allows to refer to a theory A from session B by the qualified name $B.A$ — although it is loaded again into the current ML process, which is in contrast to a theory that is already present in the *parent* session.

Theories that are imported from other sessions are excluded from the current session document.

theories *options names* specifies a block of theories that are processed within an environment that is augmented by the given options, in addition to the global session options given before. Any number of blocks of **theories** may be given. Options are only active for each **theories** block separately.

A theory name that is followed by (**global**) is treated literally in other session specifications or theory imports — the normal situation is to qualify theory names by the session name; this ensures globally unique names in big session graphs. Global theories are usually the entry points to major logic sessions: *Pure*, *Main*, *Complex_Main*, *HOLCF*, *IFOL*, *FOL*, *ZF*, *ZFC* etc. Regular Isabelle applications should not claim any global theory names.

document_files (**in** *base_dir*) *files* lists source files for document preparation, typically *.tex* and *.sty* for L^AT_EX. Only these explicitly given files are copied from the base directory to the document output directory, before formal document processing is started (see also §3.3). The local path structure of the *files* is preserved, which allows to reconstruct the original directory hierarchy of *base_dir*. The default *base_dir* is **document** within the session root directory.

export_files (**in** *target_dir*) [*number*] *patterns* specifies theory exports that may get written to the file-system, e.g. via **isabelle build** with option **-e** (§2.3). The *target_dir* specification is relative to the session root directory; its default is **export**. Exports are selected via *patterns* as in **isabelle export** (§2.5). The number given in brackets (default: 0) specifies elements that should be pruned from each name: it allows to reduce the resulting directory hierarchy at the danger of overwriting files due to loss of uniqueness.

Examples

See `~/src/HOL/ROOT` for a diversity of practically relevant situations, although it uses relatively complex quasi-hierarchic naming conventions like `HOL-SPARK`, `HOL-SPARK-Examples`. An alternative is to use unqualified names that are relatively long and descriptive, as in the Archive of Formal Proofs (<https://isa-afp.org>), for example.

2.2 System build options

See `~/etc/options` for the main defaults provided by the Isabelle distribution. Isabelle/jEdit [3] includes a simple editing mode `isabelle-options` for this file-format.

The following options are particularly relevant to build Isabelle sessions, in particular with document preparation (chapter 3).

- `browser_info` controls output of HTML browser info, see also §3.1.
- `document` specifies the document output format, see `isabelle document` option `-o` in §3.3. In practice, the most relevant values are `document=false` or `document=pdf`.
- `document_output` specifies an alternative directory for generated output of the document preparation system; the default is within the `ISABELLE_BROWSER_INFO` hierarchy as explained in §3.1. See also `isabelle mkroot`, which generates a default configuration with output readily available to the author of the document.
- `document_variants` specifies document variants as a colon-separated list of `name=tags` entries, corresponding to `isabelle document` options `-n` and `-t`.

For example, `document_variants=document:outline=/proof,/ML` indicates two documents: the one called `document` with default tags, and the other called `outline` where proofs and ML sections are folded.

Document variant names are just a matter of conventions. It is also possible to use different document variant names (without tags) for different document root entries, see also §3.3.

- `document_tags` specifies alternative command tags as a comma-separated list of items: either `“command%tag”` for a specific command,

or “%tag” as default for all other commands. This is occasionally useful to control the global visibility of commands via session options (e.g. in ROOT).

- **threads** determines the number of worker threads for parallel checking of theories and proofs. The default 0 means that a sensible maximum value is determined by the underlying hardware. For machines with many cores or with hyperthreading, this is often requires manual adjustment (on the command-line or within personal settings or preferences, not within a session ROOT).
- **condition** specifies a comma-separated list of process environment variables (or Isabelle settings) that are required for the subsequent theories to be processed. Conditions are considered “true” if the corresponding environment value is defined and non-empty.
- **timeout** and **timeout_scale** specify a real wall-clock timeout for the session as a whole: the two values are multiplied and taken as the number of seconds. Typically, **timeout** is given for individual sessions, and **timeout_scale** as global adjustment to overall hardware performance. The timer is controlled outside the ML process by the JVM that runs Isabelle/Scala. Thus it is relatively reliable in canceling processes that get out of control, even if there is a deadlock without CPU time usage.
- **profiling** specifies a mode for global ML profiling. Possible values are the empty string (disabled), **time** for **profile_time** and **allocations** for **profile_allocations**. Results appear near the bottom of the session log file.
- **system_heaps** determines the directories for session heap images: **\$ISABELLE_HEAPS** is the user directory and **\$ISABELLE_HEAPS_SYSTEM** the system directory (usually within the Isabelle application). For **system_heaps=false**, heaps are stored in the user directory and may be loaded from both directories. For **system_heaps=true**, store and load happens only in the system directory.

The `isabelle options` tool prints Isabelle system options. Its command-line usage is:

```
Usage: isabelle options [OPTIONS] [MORE_OPTIONS ...]
```

Options are:

```
-b          include $ISABELLE_BUILD_OPTIONS
```

```

-g OPTION    get value of OPTION
-l           list options
-x FILE      export to FILE in YXML format

```

Report Isabelle system options, augmented by MORE_OPTIONS given as arguments NAME=VAL or NAME.

The command line arguments provide additional system options of the form *name=value* or *name* for Boolean options.

Option `-b` augments the implicit environment of system options by the ones of `ISABELLE_BUILD_OPTIONS`, cf. §2.3.

Option `-g` prints the value of the given option. Option `-l` lists all options with their declaration and current value.

Option `-x` specifies a file to export the result in YXML format, instead of printing it in human-readable form.

2.3 Invoking the build process

The `isabelle build` tool invokes the build process for Isabelle sessions. It manages dependencies between sessions, related sources of theories and auxiliary files, and target heap images. Accordingly, it runs instances of the prover process with optional document preparation. Its command-line usage is:¹

```
Usage: isabelle build [OPTIONS] [SESSIONS ...]
```

```

Options are:
-B NAME      include session NAME and all descendants
-D DIR       include session directory and select its sessions
-N           cyclic shuffling of NUMA CPU nodes (performance tuning)
-R           operate on requirements of selected sessions
-S           soft build: only observe changes of sources, not heap
images
-X NAME      exclude sessions from group NAME and all descendants
-a           select all sessions
-b           build heap images
-c           clean build
-d DIR       include session directory
-e           export files from session specification into file-system
-f           fresh build
-g NAME      select session group NAME
-j INT       maximum number of parallel jobs (default 1)

```

¹Isabelle/Scala provides the same functionality via `isabelle.Build.build`.


```

-k KEYWORD    check theory sources for conflicts with proposed keywords
-l            list session source files
-n            no build -- test dependencies only
-o OPTION     override Isabelle system OPTION (via NAME=VAL or NAME)
-v            verbose
-x NAME       exclude session NAME and all descendants

```

Build and manage Isabelle sessions, depending on implicit settings:

```
ISABELLE_BUILD_OPTIONS="..."
```

```
ML_PLATFORM="..."
```

```
ML_HOME="..."
```

```
ML_SYSTEM="..."
```

```
ML_OPTIONS="..."
```

Isabelle sessions are defined via session *ROOT* files as described in (§2.1). The totality of sessions is determined by collecting such specifications from all Isabelle component directories (§1.1.3), augmented by more directories given via options *-d DIR* on the command line. Each such directory may contain a session *ROOT* file with several session specifications.

Any session root directory may refer recursively to further directories of the same kind, by listing them in a catalog file *ROOTS* line-by-line. This helps to organize large collections of session specifications, or to make *-d* command line options persistent (e.g. in *\$ISABELLE_HOME_USER/ROOTS*).

The subset of sessions to be managed is determined via individual *SESSIONS* given as command-line arguments, or session groups that are given via one or more options *-g NAME*. Option *-a* selects all sessions. The build tool takes session dependencies into account: the set of selected sessions is completed by including all ancestors.

One or more options *-B NAME* specify base sessions to be included (all descendants wrt. the session parent or import graph).

One or more options *-x NAME* specify sessions to be excluded (all descendants wrt. the session parent or import graph). Option *-X* is analogous to this, but excluded sessions are specified by session group membership.

Option *-R* reverses the selection in the sense that it refers to its requirements: all ancestor sessions excluding the original selection. This allows to prepare the stage for some build process with different options, before running the main build itself (without option *-R*).

Option *-D* is similar to *-d*, but selects all sessions that are defined in the given directories.

Option `-S` indicates a “soft build”: the selection is restricted to those sessions that have changed sources (according to actually imported theories). The status of heap images is ignored.

The build process depends on additional options (§2.2) that are passed to the prover eventually. The settings variable `ISABELLE_BUILD_OPTIONS` allows to provide additional defaults, e.g. `ISABELLE_BUILD_OPTIONS="document=pdf threads=4"`. Moreover, the environment of system build options may be augmented on the command line via `-o name=value` or `-o name`, which abbreviates `-o name=true` for Boolean options. Multiple occurrences of `-o` on the command-line are applied in the given order.

Option `-b` ensures that heap images are produced for all selected sessions. By default, images are only saved for inner nodes of the hierarchy of sessions, as required for other sessions to continue later on.

Option `-c` cleans the selected sessions (all descendants wrt. the session parent or import graph) before performing the specified build operation.

Option `-e` executes the `export_files` directives from the ROOT specification of all explicitly selected sessions: the status of the session build database needs to be OK, but the session could have been built earlier. Using `export_files`, a session may serve as abstract interface for add-on build artefacts, but these are only materialized on explicit request: without option `-e` there is no effect on the physical file-system yet.

Option `-f` forces a fresh build of all selected sessions and their requirements.

Option `-n` omits the actual build process after the preparatory stage (including optional cleanup). Note that the return code always indicates the status of the set of selected sessions.

Option `-j` specifies the maximum number of parallel build jobs (prover processes). Each prover process is subject to a separate limit of parallel worker threads, cf. system option `threads`.

Option `-N` enables cyclic shuffling of NUMA CPU nodes. This may help performance tuning on Linux servers with separate CPU/memory modules.

Option `-v` increases the general level of verbosity. Option `-l` lists the source files that contribute to a session.

Option `-k` specifies a newly proposed keyword for outer syntax (multiple uses allowed). The theory sources are checked for conflicts wrt. this hypothetical change of syntax, e.g. to reveal occurrences of identifiers that need to be quoted.

Examples

Build a specific logic image:

```
isabelle build -b HOLCF
```

Build the main group of logic images:

```
isabelle build -b -g main
```

Build all descendants (and requirements) of FOL and ZF:

```
isabelle build -B FOL -B ZF
```

Build all sessions where sources have changed (ignoring heaps):

```
isabelle build -a -S
```

Provide a general overview of the status of all Isabelle sessions, without building anything:

```
isabelle build -a -n -v
```

Build all sessions with HTML browser info and PDF document preparation:

```
isabelle build -a -o browser_info -o document=pdf
```

Build all sessions with a maximum of 8 parallel prover processes and 4 worker threads each (on a machine with many cores):

```
isabelle build -a -j8 -o threads=4
```

Build some session images with cleanup of their descendants, while retaining their ancestry:

```
isabelle build -b -c HOL-Algebra HOL-Word
```

Clean all sessions without building anything:

```
isabelle build -a -n -c
```

Build all sessions from some other directory hierarchy, according to the settings variable AFP that happens to be defined inside the Isabelle environment:

```
isabelle build -D '$AFP'
```

Inform about the status of all sessions required for AFP, without building anything yet:

```
isabelle build -D '$AFP' -R -v -n
```

2.4 Maintain theory imports wrt. session structure

The `isabelle imports` tool helps to maintain theory imports wrt. session structure. It supports three main operations via options `-I`, `-M`, `-U`. Its command-line usage is:

Usage: `isabelle imports [OPTIONS] [SESSIONS ...]`

Options are:

<code>-B NAME</code>	include session NAME and all descendants
<code>-D DIR</code>	include session directory and select its sessions
<code>-I</code>	operation: report session imports
<code>-M</code>	operation: Mercurial repository check for theory files
<code>-R</code>	operate on requirements of selected sessions
<code>-U</code>	operation: update theory imports to use session qualifiers
<code>-X NAME</code>	exclude sessions from group NAME and all descendants
<code>-a</code>	select all sessions
<code>-d DIR</code>	include session directory
<code>-g NAME</code>	select session group NAME
<code>-i</code>	incremental update according to session graph structure
<code>-o OPTION</code>	override Isabelle system OPTION (via NAME=VAL or NAME)
<code>-v</code>	verbose
<code>-x NAME</code>	exclude session NAME and all descendants

Maintain theory imports wrt. session structure. At least one operation needs to be specified (see options `-I -M -U`).

The selection of sessions and session directories works as for `isabelle build` via options `-B`, `-D`, `-R`, `-X`, `-a`, `-d`, `-g`, `-x` (see §2.3).

Option `-o` overrides Isabelle system options as for `isabelle build` (see §2.3).

Option `-v` increases the general level of verbosity.

Option `-I` determines reports session imports:

Potential session imports are derived from old-style use of theory files from other sessions via the directory structure. After declaring those as **sessions** in the corresponding `ROOT` file entry, a proper session-qualified theory name can be used (cf. option `-U`). For example, `adhoc imports "~/src/HOL/Library/Multiset"` becomes formal `imports "HOL-Library.Multiset"` after adding **sessions** `"HOL-Library"` to the `ROOT` entry.

Actual session imports are derived from the session qualifiers of all currently imported theories. This helps to minimize dependencies in the session import structure to what is actually required.

Option `-M` checks imported theories against the Mercurial repositories of the underlying session directories; non-repository directories are ignored. This helps to find files that are accidentally ignored, e.g. due to rearrangements of the session structure.

Option `-U` updates theory imports with old-style directory specifications to canonical session-qualified theory names, according to the theory name space imported via **sessions** within the `ROOT` specification.

Option `-i` modifies the meaning of option `-U` to proceed incrementally, following to the session graph structure in bottom-up order. This may lead to more accurate results in complex session hierarchies.

Examples

Determine potential session imports for some project directory:

```
isabelle imports -I -D 'some/where/My_Project'
```

Mercurial repository check for some project directory:

```
isabelle imports -M -D 'some/where/My_Project'
```

Incremental update of theory imports for some project directory:

```
isabelle imports -U -i -D 'some/where/My_Project'
```

2.5 Retrieve theory exports

The `isabelle export` tool retrieves theory exports from the session database. Its command-line usage is:

```
Usage: isabelle export [OPTIONS] SESSION
```

Options are:

```
-O DIR      output directory for exported files (default: "export")
-d DIR      include session directory
-l          list exports
```

```

-n          no build of session
-o OPTION   override Isabelle system OPTION (via NAME=VAL or NAME)
-p NUM      prune path of exported files by NUM elements
-x PATTERN  extract files matching pattern (e.g.\ "*:**" for all)

```

List or export theory exports for SESSION: named blobs produced by `isabelle build`. Option `-l` or `-x` is required; option `-x` may be repeated.

The PATTERN language resembles glob patterns in the shell, with `?` and `*` (both excluding `:` and `/`), `**` (excluding `:`), and `[abc]` or `[^abc]`, and variants `{pattern1,pattern2,pattern3}`.

The specified session is updated via `isabelle build` (§2.3), with the same options `-d`, `-o`. The option `-n` suppresses the implicit build process: it means that a potentially outdated session database is used!

Option `-l` lists all stored exports, with compound names *theory:name*.

Option `-x` extracts stored exports whose compound name matches the given pattern. Note that wild cards `?` and `*` do not match the separators `:` and `/`; the wild card `**` matches over directory name hierarchies separated by `/`. Thus the pattern `*:**` matches *all* theory exports. Multiple options `-x` refer to the union of all specified patterns.

Option `-O` specifies an alternative output directory for option `-x`: the default is `export` within the current directory. Each theory creates its own sub-directory hierarchy, using the session-qualified theory name.

Option `-p` specifies the number of elements that should be pruned from each name: it allows to reduce the resulting directory hierarchy at the danger of overwriting files due to loss of uniqueness.

2.6 Dump PIDE session database

The `isabelle dump` tool dumps information from the cumulative PIDE session database (which is processed on the spot). Its command-line usage is:

```
Usage: isabelle dump [OPTIONS] [SESSIONS ...]
```

Options are:

```

-A NAMES    dump named aspects (default: ...)
-B NAME     include session NAME and all descendants
-D DIR      include session directory and select its sessions
-O DIR      output directory for dumped files (default: "dump")
-R          operate on requirements of selected sessions
-X NAME     exclude sessions from group NAME and all descendants

```

```

-a          select all sessions
-d DIR     include session directory
-g NAME    select session group NAME
-l NAME    logic session name (default ISABELLE_LOGIC="HOL")
-o OPTION  override Isabelle system OPTION (via NAME=VAL or NAME)
-v        verbose
-x NAME    exclude session NAME and all descendants

```

Dump cumulative PIDE session database, with the following aspects:

...

Options `-B`, `-D`, `-R`, `-X`, `-a`, `-d`, `-g`, `-x` and the remaining command-line arguments specify sessions as in `isabelle build` (§2.3): the cumulative PIDE database of all their loaded theories is dumped to the output directory of option `-O` (default: `dump` in the current directory).

Option `-l` specifies a logic image for the underlying prover process: its theories are not processed again, and their PIDE session database is excluded from the dump.

Option `-o` overrides Isabelle system options as for `isabelle build` (§2.3).

Option `-v` increases the general level of verbosity.

Option `-A` specifies named aspects of the dump, as a comma-separated list. The default is to dump all known aspects, as given in the command-line usage of the tool. The underlying Isabelle/Scala function `isabelle.Dump.dump()` takes aspects as user-defined operations on the final PIDE state and document version. This allows to imitate Prover IDE rendering under program control.

Examples

Dump all Isabelle/ZF sessions (which are rather small):

```
isabelle dump -v -B ZF
```

Dump the quite substantial `HOL-Analysis` session, using main Isabelle/HOL as starting point:

```
isabelle dump -v -l HOL HOL-Analysis
```

Dump all sessions connected to `HOL-Analysis`, including a full bootstrap of Isabelle/HOL from Isabelle/Pure:

```
isabelle dump -v -l Pure -B HOL-Analysis
```

This results in uniform PIDE markup for everything, except for the Isabelle/Pure bootstrap process itself. Producing that on the spot requires several GB of heap space, both for the Isabelle/Scala and Isabelle/ML process (in 64bit mode). Here are some relevant settings (§1.1.1) for such ambitious applications:

```
ISABELLE_TOOL_JAVA_OPTIONS="-Xms4g -Xmx32g -Xss16m"
ML_OPTIONS="--minheap 4G --maxheap 32G"
```

2.7 Update theory sources based on PIDE markup

The `isabelle update` tool updates theory sources based on markup that is produced from a running PIDE session (similar to `isabelle dump` §2.6). Its command-line usage is:

```
Usage: isabelle update [OPTIONS] [SESSIONS ...]
```

Options are:

```
-B NAME      include session NAME and all descendants
-D DIR      include session directory and select its sessions
-R          operate on requirements of selected sessions
-X NAME     exclude sessions from group NAME and all descendants
-a         select all sessions
-d DIR     include session directory
-g NAME    select session group NAME
-l NAME    logic session name (default ISABELLE_LOGIC="HOL")
-o OPTION  override Isabelle system OPTION (via NAME=VAL or NAME)
-u OPT     override update option: shortcut for "-o update_OPT"
-v         verbose
-x NAME    exclude session NAME and all descendants
```

Update theory sources based on PIDE markup.

Options `-B`, `-D`, `-R`, `-X`, `-a`, `-d`, `-g`, `-x` and the remaining command-line arguments specify sessions as in `isabelle build` (§2.3) or `isabelle dump` (§2.6).

Option `-l` specifies the underlying logic image.

Option `-v` increases the general level of verbosity.

Option `-o` overrides Isabelle system options as for `isabelle build` (§2.3). Option `-u` refers to specific `update` options, by relying on naming convention: “`-u OPT`” is a shortcut for “`-o update_ OPT`”.

The following update options are supported:

- `update_inner_syntax_cartouches` to update inner syntax (types, terms, etc.) to use cartouches, instead of double-quoted strings or atomic identifiers. For example, “`lemma "x = x"`” is replaced by “`lemma <x = x>`”, and “`assume A`” is replaced by “`assume <A>`”.
- `update_mixfix_cartouches` to update mixfix templates to use cartouches instead of double-quoted strings. For example, “`(infixl "+" 65)`” is replaced by “`(infixl <+> 65)`”.
- `update_control_cartouches` to update antiquotations to use the compact form with control symbol and cartouche argument. For example, “`@{term "x + y"}`” is replaced by “`term <x + y>`” (the control symbol is literally `\<^term>`.)
- `update_path_cartouches` to update file-system paths to use cartouches: this depends on language markup provided by semantic processing of parsed input.

It is also possible to produce custom updates in Isabelle/ML, by reporting `Markup.update` with the precise source position and a replacement text. This operation should be made conditional on specific system options, similar to the ones above. Searching the above option names in ML sources of `$ISABELLE_HOME/src/Pure` provides some examples.

Updates can be in conflict by producing nested or overlapping edits: this may require to run `isabelle update` multiple times.

Examples

Update some cartouche notation in all theory sources required for session `HOL-Analysis`:

```
isabelle update -u mixfix_cartouches -l Pure HOL-Analysis
```

Update the same for all application sessions based on `HOL-Analysis` — its image is taken as starting point and its sources are not touched:

```
isabelle update -u mixfix_cartouches -l HOL-Analysis -B HOL-Analysis
```

This two-stage approach reduces resource requirements of the running PIDE session: a base image like `HOL-Analysis` (or `HOL`, `HOL-Library`) is more compact than importing all required theory (and ML) source files from `Pure`.

Update sessions that build on `HOL-Proofs`, which need to be run separately with special options as follows:

```
isabelle update -u mixfix_cartouches -l HOL-Proofs -B HOL-Proofs  
-o record_proofs=2
```

See also the end of §2.6 for hints on increasing Isabelle/ML heap sizes for very big PIDE processes that include many sessions, notably from the Archive of Formal Proofs.

Presenting theories

Isabelle provides several ways to present the outcome of formal developments, including WWW-based browsable libraries or actual printable documents. Presentation is centered around the concept of *sessions* (chapter 2). The global session structure is that of a tree, with Isabelle Pure at its root, further object-logics derived (e.g. HOLCF from HOL, and HOL from Pure), and application sessions further on in the hierarchy.

The tools `isabelle mkroot` and `isabelle build` provide the primary means for managing Isabelle sessions, including proper setup for presentation; `isabelle build` tells the Isabelle process to run any additional stages required for document preparation, notably the `isabelle document` and `isabelle latex`. The complete tool chain for managing batch-mode Isabelle sessions is illustrated in figure 3.1.

<code>isabelle mkroot</code>	invoked once by the user to initialize the session <code>ROOT</code> with optional <code>document</code> directory;
<code>isabelle build</code>	invoked repeatedly by the user to keep session output up-to-date (HTML, documents etc.);
<code>isabelle process</code>	run through <code>isabelle build</code> ;
<code>isabelle document</code>	run by the Isabelle process if document preparation is enabled;
<code>isabelle latex</code>	universal \LaTeX tool wrapper invoked multiple times by <code>isabelle document</code> ; also useful for manual experiments;

Figure 3.1: The tool chain of Isabelle session presentation

3.1 Generating HTML browser information

As a side-effect of building sessions, Isabelle is able to generate theory browsing information, including HTML documents that show the theory sources

and the relationship with its ancestors and descendants. Besides the HTML file that is generated for every theory, Isabelle stores links to all theories of a session in an index file. As a second hierarchy, groups of sessions are organized as *chapters*, with a separate index. Note that the implicit tree structure of the session build hierarchy is *not* relevant for the presentation.

To generate theory browsing information for an existing session, just invoke `isabelle build` with suitable options:

```
isabelle build -o browser_info -v -c FOL
```

The presentation output will appear in `$ISABELLE_BROWSER_INFO/FOL/FOL` as reported by the above verbose invocation of the build process.

Many Isabelle sessions (such as `HOL-Library` in `~/src/HOL/Library`) also provide printable documents in PDF. These are prepared automatically as well if enabled like this:

```
isabelle build -o browser_info -o document=pdf -v -c HOL-Library
```

Enabling both browser info and document preparation simultaneously causes an appropriate “document” link to be included in the HTML index. Documents may be generated independently of browser information as well, see §3.3 for further details.

The theory browsing information is stored in a sub-directory directory determined by the `ISABELLE_BROWSER_INFO` setting plus a prefix corresponding to the session chapter and identifier. In order to present Isabelle applications on the web, the corresponding subdirectory from `ISABELLE_BROWSER_INFO` can be put on a WWW server.

3.2 Preparing session root directories

The `isabelle mkroot` tool configures a given directory as session root, with some `ROOT` file and optional document source directory. Its usage is:

```
Usage: isabelle mkroot [OPTIONS] [DIRECTORY]
```

Options are:

<code>-A LATEX</code>	provide author in LaTeX notation (default: user name)
<code>-I</code>	init Mercurial repository and add generated files
<code>-T LATEX</code>	provide title in LaTeX notation (default: session name)
<code>-n NAME</code>	alternative session name (default: directory base name)

Prepare session root directory (default: current directory).

The results are placed in the given directory *dir*, which refers to the current directory by default. The `isabelle mkroot` tool is conservative in the sense that it does not overwrite existing files or directories. Earlier attempts to generate a session root need to be deleted manually.

The generated session template will be accompanied by a formal document, with `DIRECTORY/document/root.tex` as its \LaTeX entry point (see also chapter 3).

Options `-T` and `-A` specify the document title and author explicitly, using \LaTeX source notation.

Option `-I` initializes a Mercurial repository in the target directory, and adds all generated files (without commit).

Option `-n` specifies an alternative session name; otherwise the base name of the given directory is used.

The implicit Isabelle settings variable `ISABELLE_LOGIC` specifies the parent session.

Examples

Produce session `Test` within a separate directory of the same name:

```
isabelle mkroot Test && isabelle build -D Test
```

Upgrade the current directory into a session `ROOT` with document preparation, and build it:

```
isabelle mkroot && isabelle build -D .
```

3.3 Preparing Isabelle session documents

The `isabelle document` tool prepares logic session documents, processing the sources as provided by the user and generated by Isabelle. Its usage is:

Usage: `isabelle document [OPTIONS]`

Options are:

<code>-d DIR</code>	document output directory (default "output/document")
<code>-n NAME</code>	document name (default "document")
<code>-o FORMAT</code>	document format: pdf (default), dvi
<code>-t TAGS</code>	markup for tagged regions

Prepare the theory session document in document directory, producing the specified output format.

This tool is usually run automatically as part of the Isabelle build process, provided document preparation has been enabled via suitable options. It may be manually invoked on the generated browser information document output as well, e.g. in case of errors encountered in the batch run.

Option `-d` specifies an alternative document output directory. The default is `output/document` (derived from the document name). Note that the result will appear in the parent of this directory.

The `-n` and `-o` option specify the final output file name and format, the default is `document.pdf`.

The `-t` option tells \LaTeX how to interpret tagged Isabelle command regions. Tags are specified as a comma separated list of modifier/name pairs: `+foo` (or just `foo`) means to keep, `-foo` to drop, and `/foo` to fold text tagged as `foo`. The builtin default is equivalent to the tag specification `+document,+theory,+proof,+ML,+visible,-invisible,+important,+unimportant`; see also the \LaTeX macros `\isakeeptag`, `\isadroptag`, and `\isafoldtag`, in `~/lib/texinputs/isabelle.sty`.

Document preparation requires a `document` directory within the session sources. This directory is supposed to contain all the files needed to produce the final document — apart from the actual theories which are generated by Isabelle.

For most practical purposes, `isabelle document` is smart enough to create any of the specified output formats, taking `root.tex` supplied by the user as a starting point. This even includes multiple runs of \LaTeX to accommodate references and bibliographies (the latter assumes `root.bib` within the same directory).

In more complex situations, a separate `build` script for the document sources may be given. It is invoked with command-line arguments for the document format and the document variant name. The script needs to produce corresponding output files, e.g. `root.pdf` for target format `pdf` (and default variants). The main work can be again delegated to `isabelle latex`, but it is also possible to harvest generated \LaTeX sources and copy them elsewhere.

When running the session, Isabelle copies the content of the original `document` directory into its proper place within `ISABELLE_BROWSER_INFO`, according to the session path and document variant. Then, for any processed

theory A some \LaTeX source is generated and put there as $A.tex$. Furthermore, a list of all generated theory files is put into $session.tex$. Typically, the root \LaTeX file provided by the user would include $session.tex$ to get a document containing all the theories.

The \LaTeX versions of the theories require some macros defined in $~/lib/texinputs/isabelle.sty$. Doing $\usepackage{isabelle}$ in $root.tex$ should be fine; the underlying $isabelle\ latex$ already includes an appropriate path specification for \TeX inputs.

If the text contains any references to Isabelle symbols (such as $\langle forall \rangle$) then $isabellesym.sty$ should be included as well. This package contains a standard set of \LaTeX macro definitions \isasympfoo corresponding to $\langle foo \rangle$, see [1] for a complete list of predefined Isabelle symbols. Users may invent further symbols as well, just by providing \LaTeX macros in a similar fashion as in $~/lib/texinputs/isabellesym.sty$ of the Isabelle distribution.

For proper setup of DVI and PDF documents (with hyperlinks and bookmarks), we recommend to include $~/lib/texinputs/pdfsetup.sty$ as well.

As a final step of Isabelle document preparation, $isabelle\ document$ is run on the generated $output/document$ directory. Thus the actual output document is built and installed in its proper place. The generated sources are deleted after successful run of \LaTeX and friends.

Some care is needed if the document output location is configured differently, say within a directory whose content is still required afterwards!

3.4 Running \LaTeX within the Isabelle environment

The $isabelle\ latex$ tool provides the basic interface for Isabelle document preparation. Its usage is:

```
Usage: isabelle latex [OPTIONS] [FILE]
```

Options are:

```
-o FORMAT    specify output format: pdf (default), dvi, bbl, idx, sty
```

```
Run LaTeX (and related tools) on FILE (default root.tex),
producing the specified output format.
```

Appropriate \LaTeX -related programs are run on the input file, according to the given output format: $latex$, $pdflatex$, $dvips$, $bibtex$ (for bbl), and

`makeindex` (for `idx`). The actual commands are determined from the settings environment (`ISABELLE_PDFLATEX` etc.).

The `sty` output format causes the Isabelle style files to be updated from the distribution. This is useful in special situations where the document sources are to be processed another time by separate tools.

Examples

Invoking `isabelle latex` by hand may be occasionally useful when debugging failed attempts of the automatic document preparation stage of batch-mode Isabelle. The abortive process leaves the sources at a certain place within `ISABELLE_BROWSER_INFO`, see the runtime error message for details. This enables users to inspect \LaTeX runs in further detail, e.g. like this:

```
cd "$(isabelle getenv -b ISABELLE_BROWSER_INFO)/Unsorted/Test/document"  
isabelle latex -o pdf
```

The Isabelle server

An Isabelle session requires at least two processes, which are both rather heavy: Isabelle/Scala for the system infrastructure and Isabelle/ML for the logic session (e.g. HOL). In principle, these processes can be invoked directly on the command-line, e.g. via `isabelle java`, `isabelle scala`, `isabelle process`, `isabelle console`, but this approach is inadequate for reactive applications that require quick responses from the prover.

In contrast, the Isabelle server exposes Isabelle/Scala as a “terminate-stay-resident” application that manages multiple logic *sessions* and concurrent tasks to use *theories*. This provides an analogous to `Thy_Info.use_theories` in Isabelle/ML, but with full concurrency and Isabelle/PIDE markup.

The client/server arrangement via TCP sockets also opens possibilities for remote Isabelle services that are accessed by local applications, e.g. via an SSH tunnel.

4.1 Command-line tools

4.1.1 Server

The `isabelle server` tool manages resident server processes:

Usage: `isabelle server [OPTIONS]`

Options are:

<code>-L FILE</code>	logging on FILE
<code>-c</code>	console interaction with specified server
<code>-l</code>	list servers (alternative operation)
<code>-n NAME</code>	explicit server name (default: <code>isabelle</code>)
<code>-p PORT</code>	explicit server port
<code>-s</code>	assume existing server, no implicit startup
<code>-x</code>	exit specified server (alternative operation)

Manage resident Isabelle servers.

The main operation of `isabelle server` is to ensure that a named server

is running, either by finding an already running process (according to the central database file `$ISABELLE_HOME_USER/servers.db`) or by becoming itself a new server that accepts connections on a particular TCP socket. The server name and its address are printed as initial output line. If another server instance is already running, the current `isabelle server` process will terminate; otherwise, it keeps running as a new server process until an explicit `shutdown` command is received. Further details of the server socket protocol are explained in §4.2.

Other server management operations are invoked via options `-l` and `-x` (see below).

Option `-n` specifies an alternative server name: at most one process for each name may run, but each server instance supports multiple connections and logic sessions.

Option `-p` specifies an explicit TCP port for the server socket (which is always on `localhost`): the default is to let the operating system assign a free port number.

Option `-s` strictly assumes that the specified server process is already running, skipping the optional server startup phase.

Option `-c` connects the console in/out channels after the initial check for a suitable server process. Also note that the `isabelle client` tool (§4.1.2) provides a command-line editor to interact with the server.

Option `-L` specifies a log file for exceptional output of internal server and session operations.

Operation `-l` lists all active server processes with their connection details.

Operation `-x` exits the specified server process by sending it a `shutdown` command.

4.1.2 Client

The `isabelle client` tool provides console interaction for Isabelle servers:

```
Usage: isabelle client [OPTIONS]
```

```
Options are:
```

```
  -n NAME      explicit server name
  -p PORT      explicit server port
```

```
Console interaction for Isabelle server (with line-editor).
```

This is a wrapper to `isabelle server -s -c` for interactive experimentation, which uses `ISABELLE_LINE_EDITOR` if available. The server name is sufficient for identification, as the client can determine the connection details from the local database of active servers.

Option `-n` specifies an explicit server name as in `isabelle server`.

Option `-p` specifies an explicit server port as in `isabelle server`.

4.1.3 Examples

Ensure that a particular server instance is running in the background:

```
isabelle server -n test &
```

The first line of output presents the connection details:¹

```
server "test" = 127.0.0.1:4711 (password "XYZ")
```

List available server processes:

```
isabelle server -l
```

Connect the command-line client to the above test server:

```
isabelle client -n test
```

Interaction now works on a line-by-line basis, with commands like `help` or `echo`. For example, some JSON values may be echoed like this:

```
echo 42
echo [1, 2, 3]
echo {"a": "text", "b": true, "c": 42}
```

Closing the connection (via CTRL-D) leaves the server running: it is possible to reconnect again, and have multiple connections at the same time.

Exit the named server on the command-line:

```
isabelle server -n test -x
```

¹This information may be used in other TCP clients, without access to Isabelle/Scala and the underlying database of running servers.

4.2 Protocol messages

The Isabelle server listens on a regular TCP socket, using a line-oriented protocol of structured messages. Input *commands* and output *results* (via OK or ERROR) are strictly alternating on the toplevel, but commands may also return a *task* identifier to indicate an ongoing asynchronous process that is joined later (via FINISHED or FAILED). Asynchronous NOTE messages may occur at any time: they are independent of the main command-result protocol.

For example, the synchronous `echo` command immediately returns its argument as OK result. In contrast, the asynchronous `session_build` command returns OK {"task":*id*} and continues in the background. It will eventually produce FINISHED {"task":*id*,...} or FAILED {"task":*id*,...} with the final result. Intermediately, it may emit asynchronous messages of the form NOTE {"task":*id*,...} to inform about its progress. Due to the explicit task identifier, the client can show these messages in the proper context, e.g. a GUI window for this particular session build job.

Subsequently, the protocol message formats are described in further detail.

4.2.1 Byte messages

The client-server connection is a raw byte-channel for bidirectional communication, but the Isabelle server always works with messages of a particular length. Messages are written as a single chunk that is flushed immediately.

Message boundaries are determined as follows:

- A *short message* consists of a single line: it is a sequence of arbitrary bytes excluding CR (13) and LF (10), and terminated by CR-LF or just LF.
- A *long message* starts with a single that consists only of decimal digits: these are interpreted as length of the subsequent block of arbitrary bytes. A final line-terminator (as above) may be included here, but is not required.

Messages in JSON format (see below) always fit on a single line, due to escaping of newline characters within string literals. This is convenient for interactive experimentation, but it can impact performance for very long messages. If the message byte-length is given on the preceding line, the server can read the message more efficiently as a single block.

4.2.2 Text messages

Messages are read and written as byte streams (with byte lengths), but the content is always interpreted as plain text in terms of the UTF-8 encoding.² Note that line-endings and other formatting characters are invariant wrt. UTF-8 representation of text: thus implementations are free to determine the overall message structure before or after applying the text encoding.

4.2.3 Input and output messages

Server input and output messages have a uniform format as follows:

- *name argument* such that:
- *name* is the longest prefix consisting of ASCII letters, digits, “_”, “.”,
- the separator between *name* and *argument* is the longest possible sequence of ASCII blanks (it could be empty, e.g. when the argument starts with a quote or bracket),
- *argument* is the rest of the message without line terminator.

Input messages are sent from the client to the server. Here the *name* specifies a *server command*: the list of known commands may be retrieved via the `help` command.

Output messages are sent from the server to the client. Here the *name* specifies the *server reply*, which always has a specific meaning as follows:

- synchronous results: `OK` or `ERROR`
- asynchronous results: `FINISHED` or `FAILED`
- intermediate notifications: `NOTE`

The *argument* format is uniform for both input and output messages:

- empty argument (Scala type `Unit`)
- XML element in YXML notation (Scala type `XML.Elem`)

²See also the “UTF-8 Everywhere Manifesto” <https://utf8everywhere.org>.

- JSON value (Scala type `JSON.T`)

JSON values may consist of objects (records), arrays (lists), strings, numbers, booleans, null.³ Since JSON requires explicit quotes and backslash-escapes to represent arbitrary text, the YXML notation for XML trees (§1.5) works better for large messages with a lot of PIDE markup.

Nonetheless, the most common commands use JSON by default: big chunks of text (theory sources etc.) are taken from the underlying file-system and results are pre-formatted for plain-text output, without PIDE markup information. This is a concession to simplicity: the server imitates the appearance of command-line tools on top of the Isabelle/PIDE infrastructure.

4.2.4 Initial password exchange

Whenever a new client opens the server socket, the initial message needs to be its unique password as a single line, without length indication (i.e. a “short message” in the sense of §4.2.1).

The server replies either with `OK` (and some information about the Isabelle version) or by silent disconnection of what is considered an illegal connection attempt. Note that `isabelle client` already presents the correct password internally.

Server passwords are created as Universally Unique Identifier (UUID) in Isabelle/Scala and stored in a per-user database, with restricted file-system access only for the current user. The Isabelle/Scala server implementation is careful to expose the password only on private output channels, and not on a process command-line (which is accessible to other users, e.g. via the `ps` command).

4.2.5 Synchronous commands

A *synchronous command* corresponds to regular function application in Isabelle/Scala, with single argument and result (regular or error). Both the argument and the result may consist of type `Unit`, `XML.Elem`, `JSON.T`. An error result typically consists of a JSON object with error message and potentially further result fields (this resembles exceptions in Scala).

These are the protocol exchanges for both cases of command execution:

³See also the official specification <https://www.json.org> and unofficial explorations “Parsing JSON is a Minefield” http://seriot.ch/parsing_json.php.

- input:** *command argument*
- (a) regular **output:** *OK result*
- (b) error **output:** *ERROR result*

4.2.6 Asynchronous commands

An *asynchronous command* corresponds to an ongoing process that finishes or fails eventually, while emitting arbitrary notifications in between. Formally, it starts as synchronous command with immediate result **OK** giving the **task** identifier, or an immediate **ERROR** that indicates bad command syntax. For a running task, the termination is indicated later by **FINISHED** or **FAILED**, together with its ultimate result value.

These are the protocol exchanges for various cases of command task execution:

- input:** *command argument*
- immediate **output:** *OK {"task": id}*
- intermediate **output:** *NOTE {"task": id, ...}*
- (a) regular **output:** *FINISHED {"task": id, ...}*
- (b) error **output:** *FAILED {"task": id, ...}*

- input:** *command argument*
- immediate **output:** *ERROR ...*

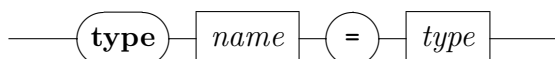
All asynchronous messages are decorated with the task identifier that was revealed in the immediate (synchronous) result. Thus the client can invoke further asynchronous commands and still dispatch the resulting stream of asynchronous messages properly.

The synchronous command `cancel {"task": id}` tells the specified task to terminate prematurely: usually causing a **FAILED** result, but this is not guaranteed: the cancel event may come too late or the running process may just ignore it.

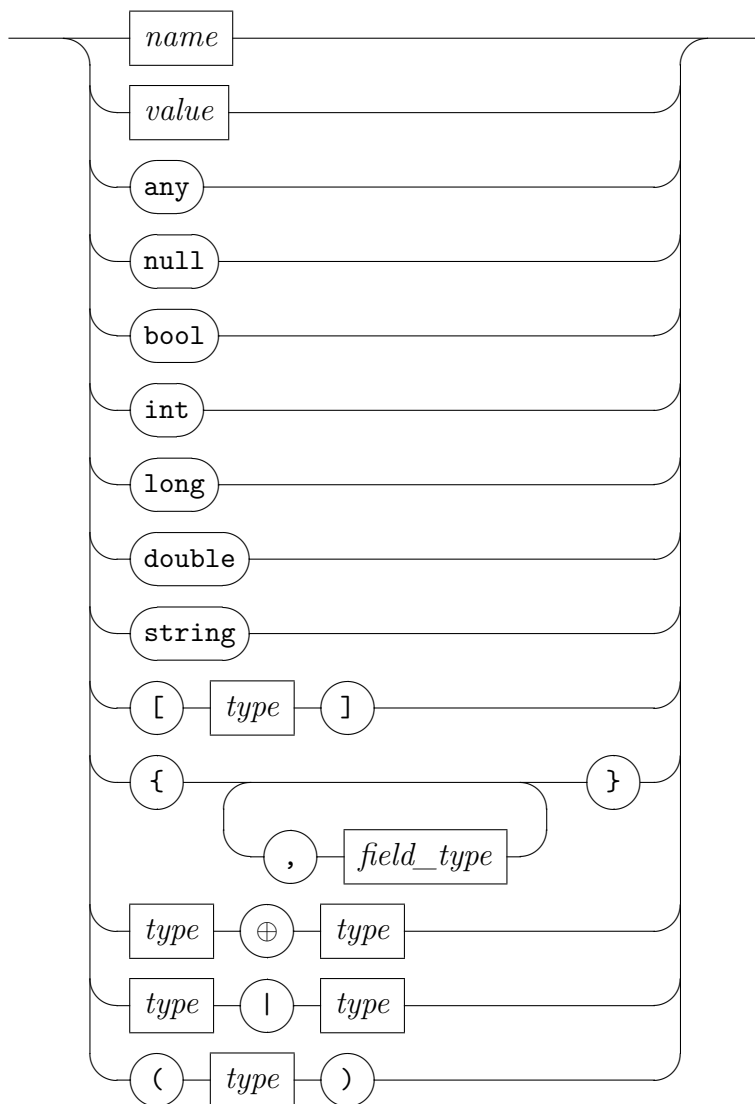
4.3 Types for JSON values

In order to specify concrete JSON types for command arguments and result messages, the following type definition language shall be used:

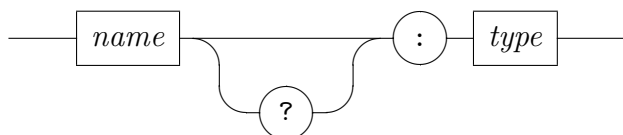
type_def



type



field_type



This is a simplified variation of TypeScript interfaces.⁴ The meaning of these types is specified wrt. the Isabelle/Scala implementation as follows.

⁴<https://www.typescriptlang.org/docs/handbook/interfaces.html>

- A *name* refers to a type defined elsewhere. The environment of type definitions is given informally: put into proper foundational order, it needs to specify a strongly normalizing system of syntactic abbreviations; type definitions may not be recursive.
- A *value* in JSON notation represents the singleton type of the given item. For example, the string "error" can be used as type for a slot that is guaranteed to contain that constant.
- Type *any* is the super type of all other types: it is an untyped slot in the specification and corresponds to `Any` or `JSON.T` in Isabelle/Scala.
- Type *null* is the type of the improper value *null*; it corresponds to type `Null` in Scala and is normally not used in Isabelle/Scala.⁵
- Type *bool* is the type of the truth values `true` and `false`; it corresponds to `Boolean` in Scala.
- Types *int*, *long*, *double* are specific versions of the generic *number* type, corresponding to `Int`, `Long`, `Double` in Scala, but `Long` is limited to 53 bit precision.⁶
- Type *string* represents Unicode text; it corresponds to type `String` in Scala.
- Type `[t]` is the array (or list) type over *t*; it corresponds to `List[t]` in Scala. The list type is co-variant as usual (i.e. monotonic wrt. the subtype relation).
- Object types describe the possible content of JSON records, with field names and types. A question mark after a field name means that it is optional. In Scala this could refer to an explicit type `Option[t]`, e.g. `{a: int, b?: string}` corresponding to a Scala case class with arguments `a: Int, b: Option[String]`.

Alternatively, optional fields can have a default value. If nothing else is specified, a standard “empty value” is used for each type, i.e. `0` for the number types, `false` for *bool*, or the empty string, array, object etc.

Object types are *permissive* in the sense that only the specified field names need to conform to the given types, but unspecified fields may be present as well.

⁵See also “Null References: The Billion Dollar Mistake” by Tony Hoare <https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare>.

⁶Implementations of JSON typically standardize *number* to `Double`, which can absorb `Int` faithfully, but not all of `Long`.

- The type expression $t_1 \oplus t_2$ only works for two object types with disjoint field names: it is the concatenation of the respective *field_type* specifications taken together. For example: $\{task: string\} \oplus \{ok: bool\}$ is the equivalent to $\{task: string, ok: bool\}$.
- The type expression $t_1 | t_2$ is the disjoint union of two types, either one of the two cases may occur.
- Parentheses (t) merely group type expressions syntactically.

These types correspond to JSON values in an obvious manner, which is not further described here. For example, the JSON array `[1, 2, 3]` conforms to types `[int]`, `[long]`, `[double]`, `[any]`, `any`.

Note that JSON objects require field names to be quoted, but the type language omits quotes for clarity. Thus the object `{"a": 42, "b": "xyz"}` conforms to the type $\{a: int, b: string\}$, for example.

The absence of an argument or result is represented by the Scala type `Unit`: it is written as empty text in the message *argument* (§4.2.3). This is not part of the JSON language.

Server replies have name tags like `OK`, `ERROR`: these are used literally together with type specifications to indicate the particular name with the type of its argument, e.g. `OK [string]` for a regular result that is a list (JSON array) of strings.

Here are some common type definitions, for use in particular specifications of command arguments and results.

- **type** `position` = $\{line?: int, offset?: int, end_offset?: int, file?: string, id?: long\}$ describes a source position within Isabelle text. Only the *line* and *file* fields make immediate sense to external programs. Detailed *offset* and *end_offset* positions are counted according to Isabelle symbols, see `Symbol.symbol` in Isabelle/ML [1]. The position *id* belongs to the representation of command transactions in the Isabelle/PIDE protocol: it normally does not occur in externalized positions.
- **type** `message` = $\{kind: string, message: string, pos?: position\}$ where the *kind* provides some hint about the role and importance of the message. The main message kinds are `writeln` (for regular output), `warning`, `error`.

- **type** *error_message* = {*kind*: "error", *message*: string} refers to error messages in particular. These occur routinely with **ERROR** or **FAILED** replies, but also as initial command syntax errors (which are omitted in the command specifications below).
- **type** *theory_progress* = {*kind*: "writeln", *message*: string, *theory*: string, *session*: string, *percentage?*: int} reports formal progress in loading theories (e.g. when building a session image). Apart from a regular output message, it also reveals the formal theory name (e.g. "HOL.Nat") and session name (e.g. "HOL"). Note that some rare theory names lack a proper session prefix, e.g. theory "Main" in session "HOL". The optional percentage has the same meaning as in **type** *node_status* below.
- **type** *timing* = {*elapsed*: double, *cpu*: double, *gc*: double} refers to common Isabelle timing information in seconds, usually with a precision of three digits after the point (whole milliseconds).
- **type** *uuid* = string refers to a Universally Unique Identifier (UUID) as plain text.⁷ Such identifiers are created as private random numbers of the server and only revealed to the client that creates a certain resource (e.g. task or session). A client may disclose this information for use in a different client connection: this allows to share sessions between multiple connections.

Client commands need to provide syntactically wellformed UUIDs: this is trivial to achieve by using only identifiers that have been produced by the server beforehand.
- **type** *task* = {*task*: uuid} identifies a newly created asynchronous task and thus allows the client to control it by the **cancel** command. The same task identification is included in all messages produced by this task.
- **type** *session_id* = {*session_id*: uuid} identifies a newly created PIDE session managed by the server. Sessions are independent of client connections and may be shared by different clients, as long as the internal session identifier is known.
- **type** *node* = {*node_name*: string, *theory_name*: string} represents the internal node name of a theory. The *node_name* is derived from

⁷See <https://www.ietf.org/rfc/rfc4122.txt> and <https://docs.oracle.com/javase/8/docs/api/java/util/UUID.html>.

the canonical theory file-name (e.g. "`~/src/HOL/ex/Seq.thy`" after normalization within the file-system). The *theory_name* is the session-qualified theory name (e.g. `HOL-ex.Seq`).

- **type** *node_status* = {*ok*: *bool*, *total*: *int*, *unprocessed*: *int*, *running*: *int*, *warned*: *int*, *failed*: *int*, *finished*: *int*, *canceled*: *bool*, *consolidated*: *bool*, *percentage*: *int*} represents a formal theory node status of the PIDE document model as follows.
 - Fields *total*, *unprocessed*, *running*, *warned*, *failed*, *finished* account for individual commands within a theory node; *ok* is an abstraction for *failed* = 0.
 - The *canceled* flag tells if some command in the theory has been spontaneously canceled (by an Interrupt exception that could also indicate resource problems).
 - The *consolidated* flag indicates whether the outermost theory command structure has finished (or failed) and the final **end** command has been checked.
 - The *percentage* field tells how far the node has been processed. It ranges between 0 and 99 in normal operation, and reaches 100 when the node has been formally consolidated as described above.

4.4 Server commands and results

Here follows an overview of particular Isabelle server commands with their results, which are usually represented as JSON values with types according to §4.3. The general format of input and output messages is described in §4.2.3. The relevant Isabelle/Scala source files are:

```
$ISABELLE_HOME/src/Pure/Tools/server_commands.scala
$ISABELLE_HOME/src/Pure/Tools/server.scala
$ISABELLE_HOME/src/Pure/General/json.scala
```

4.4.1 Command help

regular result: `OK [string]`

The `help` command has no argument and returns the list of server command names. This is occasionally useful for interactive experimentation (see also `isabelle client` in §4.1.2).

4.4.2 Command echo

argument: *any*
regular result: OK *any*

The `echo` command is the identity function: it returns its argument as regular result. This is occasionally useful for testing and interactive experimentation (see also `isabelle client` in §4.1.2).

The Scala type of `echo` is actually more general than given above: `Unit`, `XML.Elem`, `JSON.T` work uniformly. Note that `XML.Elem` might be difficult to type on the console in its YXML syntax (§1.5).

4.4.3 Command shutdown

regular result: OK

The `shutdown` command has no argument and result value. It forces a shutdown of the connected server process, stopping all open sessions and closing the server socket. This may disrupt pending commands on other connections!

The command-line invocation `isabelle server -x` opens a server connection and issues a `shutdown` command (see also §4.1.1).

4.4.4 Command cancel

argument: *task*
regular result: OK

The command `cancel {"task": id}` attempts to cancel the specified task. Cancellation is merely a hint that the client prefers an ongoing process to be stopped. The command always succeeds formally, but it may get ignored by a task that is still running; it might also refer to a non-existing or no-longer existing task (without producing an error).

Successful cancellation typically leads to an asynchronous failure of type `FAILED {task: uuid, message: "Interrupt"}`. A different message is also possible, depending how the task handles the event.

4.4.5 Command `session_build`

argument: `session_build_args`
 immediate result: `OK task`
 notifications: `NOTE task` \oplus (`theory_progress` | `message`)
 regular result: `FINISHED task` \oplus `session_build_results`
 error result: `FAILED task` \oplus `error_message` \oplus `session_build_results`

```

type session_build_args =
  {session: string,
   preferences?: string,           default: server preferences
   options?: [string],
   dirs?: [string],
   include_sessions: [string],
   verbose?: bool}

```

```

type session_build_result =
  {session: string,
   ok: bool,
   return_code: int,
   timeout: bool,
   timing: timing}

```

```

type session_build_results =
  {ok: bool,
   return_code: int,
   sessions: [session_build_result]}

```

The `session_build` command prepares a session image for interactive use of theories. This is a limited version of command-line tool `isabelle build` (§2.3), with specific options to request a formal context for an interactive PIDE session.

The build process is asynchronous, with notifications that inform about the progress of loaded theories. Some further informative messages are output as well.

Coordination of independent build processes is at the discretion of the client (or end-user), just as for `isabelle build` and `isabelle jedit`. There is no built-in coordination of conflicting builds with overlapping hierarchies of session images. In the worst case, a session image produced by one task may get overwritten by another task!

Arguments

The *session* field specifies the target session name. The build process will produce all required ancestor images according to the overall session graph.

The environment of Isabelle system options is determined from *preferences* that are augmented by *options*, which is a list individual updates of the form the *name=value* or *name* (the latter abbreviates *name=true*); see also command-line option `-o` for `isabelle build`. The preferences are loaded from the file `$ISABELLE_HOME_USER/etc/preferences` by default, but the client may provide alternative contents for it (as text, not a file-name). This could be relevant in situations where client and server run in different operating-system contexts.

The *dirs* field specifies additional directories for session ROOT and ROOTS files (§2.1). This augments the name space of available sessions; see also option `-d` in `isabelle build`.

The *include_sessions* field specifies sessions whose theories should be included in the overall name space of session-qualified theory names. This corresponds to a **sessions** specification in ROOT files (§2.1). It enables the `use_theories` command (§4.4.8) to refer to sources from other sessions in a robust manner, instead of relying on directory locations.

The *verbose* field set to `true` yields extra verbosity. The effect is similar to option `-v` in `isabelle build`.

Intermediate output

The asynchronous notifications of command `session_build` mainly serve as progress indicator: the output resembles that of the session build window of Isabelle/jEdit after startup [3].

For the client it is usually sufficient to print the messages in plain text, but note that *theory_progress* also reveals formal *theory* and *session* names directly.

Results

The overall *session_build_results* contain both a summary and an entry *session_build_result* for each session in the build hierarchy. The result is always provided, independently of overall success (**FINISHED** task) or failure (**FAILED** task).

The *ok* field tells abstractly, whether all required session builds came out as *ok*, i.e. with zero *return_code*. A non-zero *return_code* indicates an error according to usual POSIX conventions for process exit.

The individual *session_build_result* entries provide extra fields:

- *timeout* tells if the build process was aborted after running too long,
- *timing* gives the overall process timing in the usual Isabelle format with elapsed, CPU, GC time.

Examples

Build of a session image from the Isabelle distribution:

```
session_build {"session": "HOL-Word"}
```

Build a session image from the Archive of Formal Proofs:

```
session_build {"session": "Coinductive", "dirs": ["$AFP_BASE/thys"]}
```

4.4.6 Command `session_start`

argument:	$session_build_args \oplus \{print_mode?: [string]\}$
immediate result:	OK <i>task</i>
notifications:	NOTE <i>task</i> \oplus (<i>theory_progress</i> <i>message</i>)
regular result:	FINISHED <i>task</i> \oplus <i>session_id</i> \oplus { <i>tmp_dir</i> : <i>string</i> }
error result:	FAILED <i>task</i> \oplus <i>error_message</i>

The `session_start` command starts a new Isabelle/PIDE session with underlying Isabelle/ML process, based on a session image that it produces on demand using `session_build`. Thus it accepts all *session_build_args* and produces similar notifications, but the detailed *session_build_results* are omitted.

The session build and startup process is asynchronous: when the task is finished, the session remains active for commands, until a `session_stop` or `shutdown` command is sent to the server.

Sessions are independent of client connections: it is possible to start a session and later apply `use_theories` on different connections, as long as the internal session identifier is known: shared theory imports will be used only once (and persist until purged explicitly).

Arguments

Most arguments are shared with `session_build` (§4.4.5).

The `print_mode` field adds identifiers of print modes to be made active for this session. For example, `"print_mode": ["ASCII"]` prefers ASCII replacement syntax over mathematical Isabelle symbols. See also option `-m` in `isabelle process` (§1.3.1).

Results

The `session_id` provides the internal identification of the session object within the sever process. It can remain active as long as the server is running, independently of the current client connection.

The `tmp_dir` field refers to a temporary directory that is specifically created for this session and deleted after it has been stopped. This may serve as auxiliary file-space for the `use_theories` command, but concurrent use requires some care in naming temporary files, e.g. by using sub-directories with globally unique names.

As `tmp_dir` is the default `master_dir` for commands `use_theories` and `purge_theories`, theory files copied there may be used without further path specification.

Examples

Start a default Isabelle/HOL session:

```
session_start {"session": "HOL"}
```

Start a session from the Archive of Formal Proofs:

```
session_start {"session": "Coinductive", "dirs": ["$AFP_BASE/thys"]}
```

4.4.7 Command `session_stop`

argument:	<code>session_id</code>
immediate result:	<code>OK task</code>
regular result:	<code>FINISHED task ⊕ session_stop_result</code>
error result:	<code>FAILED task ⊕ error_message ⊕ session_stop_result</code>

```
type session_stop_result = {ok: bool, return_code: int}
```

The `session_stop` command forces a shutdown of the identified PIDE session. This asynchronous task usually finishes quickly. Failure only happens in unusual situations, according to the return code of the underlying Isabelle/ML process.

Arguments

The `session_id` provides the UUID originally created by the server for this session.

Results

The `ok` field tells abstractly, whether the Isabelle/ML process has terminated properly.

The `return_code` field expresses this information according to usual POSIX conventions for process exit.

4.4.8 Command `use_theories`

argument: `use_theories_arguments`

immediate result: `OK task`

regular result: `FINISHED use_theories_results`

type `use_theories_arguments =`

```
{session_id: uuid,
  theories: [string],
  master_dir?: string,           default: session tmp_dir
  pretty_margin?: double,       default: 76
  unicode_symbols?: bool,
  export_pattern?: string,
  check_delay?: double,         default: 0.5
  check_limit?: int,
  watchdog_timeout?: double,    default: 600.0
  nodes_status_delay?: double} default: -1.0
```

```

type export =
  {name: string, base64: bool, body: string}
type node_results =
  {status: node_status, messages: [message], exports: [export]}
type nodes_status =
  [node  $\oplus$  {status: node_status}]
type use_theories_results =
  {ok: bool,
   errors: [message],
   nodes: [node  $\oplus$  node_results]}

```

The `use_theories` command updates the identified session by adding the current version of theory files to it, while dependencies are resolved implicitly. The command succeeds eventually, when all theories have status *terminated* or *consolidated* in the sense of *node_status* (§4.3).

Already used theories persist in the session until purged explicitly (§4.4.9). This also means that repeated invocations of `use_theories` are idempotent: it could make sense to do that with different values for *pretty_margin* or *unicode_symbols* to get different formatting for *errors* or *messages*.

A non-empty *export_pattern* means that theory *exports* are retrieved (see §2.5). An *export name* roughly follows file-system standards: “/” separated list of base names (excluding special names like “.” or “..”). The *base64* field specifies the format of the *body* string: it is true for a byte vector that cannot be represented as plain text in UTF-8 encoding, which means the string needs to be decoded as in `java.util.Base64.getDecoder.decode(String)`.

The status of PIDE processing is checked every *check_delay* seconds, and bounded by *check_limit* attempts (default: 0, i.e. unbounded). A *check_limit* > 0 effectively specifies a global timeout of *check_delay* × *check_limit* seconds.

If *watchdog_timeout* is greater than 0, it specifies the timespan (in seconds) after the last command status change of Isabelle/PIDE, before finishing with a potentially non-terminating or deadlocked execution.

A non-negative *nodes_status_delay* enables continuous notifications of kind *nodes_status*, with a field of name and type *nodes_status*. The time interval is specified in seconds; by default it is negative and thus disabled.

Arguments

The *session_id* is the identifier provided by the server, when the session was created (possibly on a different client connection).

The *theories* field specifies theory names as in theory **imports** or in **ROOT theories**.

The *master_dir* field specifies the master directory of imported theories: it acts like the “current working directory” for locating theory files. This is irrelevant for *theories* with an absolute path name (e.g. "`~/src/HOL/ex/Seq.thy`") or session-qualified theory name (e.g. "`HOL-ex/Seq`").

The *pretty_margin* field specifies the line width for pretty-printing. The default is suitable for classic console output. Formatting happens at the end of **use_theories**, when all prover messages are exported to the client.

The *unicode_symbols* field set to **true** renders message output for direct output on a Unicode capable channel, ideally with the Isabelle fonts as in Isabelle/jEdit. The default is to keep the symbolic representation of Isabelle text, e.g. `\<forall>` instead of its rendering as \forall . This means the client needs to perform its own rendering before presenting it to the end-user.

Results

The *ok* field indicates overall success of processing the specified theories with all their dependencies.

When *ok* is **false**, the *errors* field lists all errors cumulatively (including imported theories). The messages contain position information for the original theory nodes.

The *nodes* field provides detailed information about each imported theory node. The individual fields are as follows:

- *node_name*: the canonical name for the theory node, based on its file-system location;
- *theory_name*: the logical theory name;
- *status*: the overall node status, e.g. see the visualization in the *Theories* panel of Isabelle/jEdit [3];
- *messages*: the main bulk of prover messages produced in this theory (with kind `writeln`, `warning`, `error`).

Examples

Process some example theory from the Isabelle distribution, within the context of an already started session for Isabelle/HOL (see also §4.4.6):

```
use_theories {"session_id": ..., "theories": ["~/src/HOL/ex/Seq"]}
```

Process some example theories in the context of their (single) parent session:

```
session_start {"session": "HOL-Library"}
use_theories {"session_id": ..., "theories": ["~/src/HOL/Unix/Unix"]}
session_stop {"session_id": ...}
```

Process some example theories that import other theories via session-qualified theory names:

```
session_start {"session": "HOL", "include_sessions": ["HOL-Unix"]}
use_theories {"session_id": ..., "theories": ["HOL-Unix.Unix"]}
session_stop {"session_id": ...}
```

4.4.9 Command `purge_theories`

argument: `purge_theories_arguments`

regular result: `OK purge_theories_result`

```
type purge_theories_arguments =
  {session_id: uuid,
   theories: [string],
   master_dir?: string,           default: session tmp_dir
   all?: bool}
```

```
type purge_theories_result = {purged: [string]}
```

The `purge_theories` command updates the identified session by removing theories that are no longer required: theories that are used in pending `use_theories` tasks or imported by other theories are retained.

Arguments

The `session_id` is the identifier provided by the server, when the session was created (possibly on a different client connection).

The *theories* field specifies theory names to be purged: imported dependencies are *not* completed. Instead it is possible to provide the already completed import graph returned by `use_theories` as *nodes / node_name*.

The *master_dir* field specifies the master directory as in `use_theories`. This is irrelevant, when passing fully-qualified theory node names (e.g. *node_name* from *nodes* in *use_theories_results*).

The *all* field set to `true` attempts to purge all presently loaded theories.

Results

The *purged* field gives the theory nodes that were actually removed.

The *retained* field gives the remaining theory nodes, i.e. the complement of *purged*.

Isabelle/Scala development tools

Isabelle/ML and Isabelle/Scala are the two main language environments for Isabelle tool implementations. There are some basic command-line tools to work with the underlying Java Virtual Machine, the Scala toplevel and compiler. Note that Isabelle/jEdit [3] provides a Scala Console for interactive experimentation within the running application.

5.1 Java Runtime Environment within Isabelle

The `isabelle java` tool is a direct wrapper for the Java Runtime Environment, within the regular Isabelle settings environment (§1.1). The command line arguments are that of the underlying Java version. It is run in `-server` mode if possible, to improve performance (at the cost of extra startup time). The `java` executable is the one within `ISABELLE_JDK_HOME`, according to the standard directory layout for official JDK distributions. The class loader is augmented such that the name space of `Isabelle/Pure.jar` is available, which is the main Isabelle/Scala module.

For example, the following command-line invokes the main method of class `isabelle.GUI_Setup`, which opens a windows with some diagnostic information about the Isabelle environment:

```
isabelle java isabelle.GUI_Setup
```

5.2 Scala toplevel

The `isabelle scala` tool is a direct wrapper for the Scala toplevel; see also `isabelle java` above. The command line arguments are that of the underlying Scala version.

This allows to interact with Isabelle/Scala in TTY mode like this:

```
isabelle scala
scala> isabelle.Isabelle_System.getenv("ISABELLE_HOME")
scala> val options = isabelle.Options.init()
scala> options.bool("browser_info")
scala> options.string("document")
```

5.3 Scala compiler

The `isabelle scalac` tool is a direct wrapper for the Scala compiler; see also `isabelle scala` above. The command line arguments are that of the underlying Scala version.

This allows to compile further Scala modules, depending on existing Isabelle/Scala functionality. The resulting class or jar files can be added to the Java classpath using the `classpath` Bash function that is provided by the Isabelle process environment. Thus add-on components can register themselves in a modular manner, see also §1.1.3.

Note that jEdit [3] has its own mechanisms for adding plugin components, which needs special attention since it overrides the standard Java class loader.

5.4 Scala script wrapper

The executable `$ISABELLE_HOME/bin/isabelle_scala_script` allows to run Isabelle/Scala source files stand-alone programs, by using a suitable “hash-bang” line and executable file permissions. For example:

```
#!/usr/bin/env isabelle_scala_script

val options = isabelle.Options.init()
Console.println("browser_info = " + options.bool("browser_info"))
Console.println("document = " + options.string("document"))
```

This assumes that the executable may be found via the `PATH` from the process environment: this is the case when Isabelle settings are active, e.g. in the context of the main Isabelle tool wrapper §1.2. Alternatively, the full `$ISABELLE_HOME/bin/isabelle_scala_script` may be specified in expanded form.

Miscellaneous tools

Subsequently we describe various Isabelle related utilities, given in alphabetical order.

6.1 Resolving Isabelle components

The `isabelle components` tool resolves Isabelle components:

```
Usage: isabelle components [OPTIONS] [COMPONENTS ...]
```

Options are:

```
-I          init user settings
-R URL      component repository
             (default $ISABELLE_COMPONENT_REPOSITORY)
-a          resolve all missing components
-l          list status
```

Resolve Isabelle components via download and installation.
COMPONENTS are identified via base name.

```
ISABELLE_COMPONENT_REPOSITORY="https://isabelle.in.tum.de/components"
```

Components are initialized as described in §1.1.3 in a permissive manner, which can mark components as “missing”. This state is amended by letting `isabelle components` download and unpack components that are published on the default component repository <https://isabelle.in.tum.de/components> in particular.

Option `-R` specifies an alternative component repository. Note that `file:///` URLs can be used for local directories.

Option `-a` selects all missing components to be resolved. Explicit components may be named as command line-arguments as well. Note that components are uniquely identified by their base name, while the installation takes place in the location that was specified in the attempt to initialize the component before.

Option `-l` lists the current state of available and missing components with their location (full name) within the file-system.

Option `-I` initializes the user settings file to subscribe to the standard components specified in the Isabelle repository clone — this does not make any sense for regular Isabelle releases. If the file already exists, it needs to be edited manually according to the printed explanation.

6.2 Displaying documents

The `isabelle display` tool displays documents in DVI or PDF format:

Usage: `isabelle display DOCUMENT`

Display `DOCUMENT` (in DVI or PDF format).

The settings `DVI_VIEWER` and `PDF_VIEWER` determine the programs for viewing the corresponding file formats. Normally this opens the document via the desktop environment, potentially in an asynchronous manner with re-use of previews views.

6.3 Viewing documentation

The `isabelle doc` tool displays Isabelle documentation:

Usage: `isabelle doc [DOC ...]`

View Isabelle documentation.

If called without arguments, it lists all available documents. Each line starts with an identifier, followed by a short description. Any of these identifiers may be specified as arguments, in order to display the corresponding document (see also §6.2).

The `ISABELLE_DOCS` setting specifies the list of directories (separated by colons) to be scanned for documentations.

6.4 Shell commands within the settings environment

The `isabelle env` tool is a direct wrapper for the standard `/usr/bin/env` command on POSIX systems, running within the Isabelle settings environ-

ment (§1.1).

The command-line arguments are that of the underlying version of `env`. For example, the following invokes an instance of the GNU Bash shell within the Isabelle environment:

```
isabelle env bash
```

6.5 Inspecting the settings environment

The Isabelle settings environment — as provided by the site-default and user-specific settings files — can be inspected with the `isabelle getenv` tool:

Usage: `isabelle getenv [OPTIONS] [VARNAMES ...]`

Options are:

<code>-a</code>	display complete environment
<code>-b</code>	print values only (doesn't work for <code>-a</code>)
<code>-d FILE</code>	dump complete environment to FILE (null terminated entries)

Get value of `VARNAMES` from the Isabelle settings.

With the `-a` option, one may inspect the full process environment that Isabelle related programs are run in. This usually contains much more variables than are actually Isabelle settings. Normally, output is a list of lines of the form `name=value`. The `-b` option causes only the values to be printed.

Option `-d` produces a dump of the complete environment to the specified file. Entries are terminated by the ASCII null character, i.e. the C string terminator.

Examples

Get the location of `ISABELLE_HOME_USER` where user-specific information is stored:

```
isabelle getenv ISABELLE_HOME_USER
```

Get the value only of the same settings variable, which is particularly useful in shell scripts:

```
isabelle getenv -b ISABELLE_HOME_USER
```

6.6 Installing standalone Isabelle executables

By default, the main Isabelle binaries (`isabelle` etc.) are just run from their location within the distribution directory, probably indirectly by the shell through its `PATH`. Other schemes of installation are supported by the `isabelle install` tool:

Usage: `isabelle install [OPTIONS] BINDIR`

Options are:

`-d DISTDIR` refer to `DISTDIR` as Isabelle distribution
(default `ISABELLE_HOME`)

Install Isabelle executables with absolute references to the distribution directory.

The `-d` option overrides the current Isabelle distribution directory as determined by `ISABELLE_HOME`.

The `BINDIR` argument tells where executable wrapper scripts for `isabelle` and `isabelle_scala_script` should be placed, which is typically a directory in the shell's `PATH`, such as `$HOME/bin`.

It is also possible to make symbolic links of the main Isabelle executables manually, but making separate copies outside the Isabelle distribution directory will not work!

6.7 Creating instances of the Isabelle logo

The `isabelle logo` tool creates instances of the generic Isabelle logo as EPS and PDF, for inclusion in \LaTeX documents.

Usage: `isabelle logo [OPTIONS] XYZ`

Create instance `XYZ` of the Isabelle logo (as EPS and PDF).

Options are:

`-n NAME` alternative output base name (default `"isabelle_xyz"`)
`-q` quiet mode

Option `-n` specifies an alternative (base) name for the generated files. The default is `isabelle_xyz` in lower-case.

Option `-q` omits printing of the result file name.

Implementors of Isabelle tools and applications are encouraged to make derived Isabelle logos for their own projects using this template.

6.8 Output the version identifier of the Isabelle distribution

The `isabelle version` tool displays Isabelle version information:

```
Usage: isabelle version [OPTIONS]
```

```
Options are:
```

```
-i          short identification (derived from Mercurial id)
```

```
Display Isabelle version information.
```

The default is to output the full version string of the Isabelle distribution, e.g. “Isabelle2018: August 2018.”

The `-i` option produces a short identification derived from the Mercurial id of the `ISABELLE_HOME` directory.

Bibliography

- [1] M. Wenzel. *The Isabelle/Isar Implementation*.
<https://isabelle.in.tum.de/doc/implementation.pdf>.
- [2] M. Wenzel. *The Isabelle/Isar Reference Manual*.
<https://isabelle.in.tum.de/doc/isar-ref.pdf>.
- [3] M. Wenzel. *Isabelle/jEdit*. <https://isabelle.in.tum.de/doc/jedit.pdf>.

Index

- bash (executable), **2**, **2**
- browser_info (system option), **17**
- build (tool), **16**, **19**, **30**

- client (tool), **37**
- components (tool), **60**
- condition (system option), **18**
- console (tool), **4**, **9**

- display (tool), **61**
- doc (tool), **61**
- document (system option), **17**
- document (tool), **30**, **32**
- document_output (system option), **17**
- document_tags (system option), **17**
- document_variants (system option), **17**
- dump (tool), **25**
- DVI_VIEWER (setting), **5**

- env (tool), **61**
- export (tool), **16**, **24**

- getenv (tool), **62**

- imports (tool), **23**
- install (tool), **63**
- isabelle (executable), **1**, **6**
- ISABELLE_BIBTEX (setting), **5**
- ISABELLE_BROWSER_INFO (setting), **4**, **31**
- ISABELLE_BROWSER_INFO_SYSTEM (setting), **4**
- ISABELLE_BUILD_OPTIONS (setting), **21**
- ISABELLE_DOCS (setting), **5**
- ISABELLE_HEAPS (setting), **4**
- ISABELLE_HEAPS_SYSTEM (setting), **4**
- ISABELLE_HOME (setting), **1**, **3**
- ISABELLE_HOME_USER (setting), **3**
- ISABELLE_IDENTIFIER (setting), **4**
- isabelle_java (executable), **10**
- ISABELLE_JAVA_PLATFORM (setting), **4**
- ISABELLE_JDK_HOME (setting), **4**
- ISABELLE_LATEX (setting), **5**
- ISABELLE_LINE_EDITOR (setting), **4**
- ISABELLE_LOGIC (setting), **4**
- ISABELLE_PDFLATEX (setting), **5**
- ISABELLE_PLATFORM32 (setting), **3**
- ISABELLE_PLATFORM64 (setting), **3**
- ISABELLE_PLATFORM_FAMILY (setting), **3**
- ISABELLE_TMP_PREFIX (setting), **5**
- ISABELLE_TOOL (setting), **2**
- ISABELLE_TOOL_JAVA_OPTIONS (setting), **5**
- ISABELLE_TOOLS (setting), **5**
- ISABELLE_WINDOWS_PLATFORM32 (setting), **3**

ISABELLE_WINDOWS_PLATFORM64 (setting), **3**

java (tool), **58**

latex (tool), 30, **34**

logo (tool), **63**

mkroot (tool), 30, **31**

ML_HOME (setting), **4**

ML_IDENTIFIER (setting), **4**

ML_OPTIONS (setting), **4**

ML_PLATFORM (setting), **4**

ML_SYSTEM (setting), **4**

options (tool), **18**

PDF_VIEWER (setting), **5**

process (tool), **8**, 30

profiling (system option), **18**

rlwrap (executable), **9**

scala (tool), **58**

scalac (tool), **59**

server (tool), **36**

session_chapter (syntax), **13**

session_entry (syntax), **13**

system_heaps (system option), **18**

threads (system option), **18**, 21

timeout (system option), **18**

timeout_scale (system option), **18**

update (tool), **27**

USER_HOME (setting), **2**

version (tool), **64**