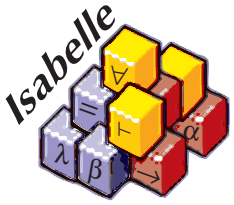


The Isabelle System Manual

Makarius Wenzel

12 December 2021

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 . . . 7
  - 1.3.1 Batch mode . . . 7
  - 1.3.2 Interactive mode . . . 9
- 1.4 The raw Isabelle Java process . . . 9
- 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 . . . 20
- 2.4 Print messages from build database . . . 24
- 2.5 Retrieve theory exports . . . 25
- 2.6 Dump PIDE session database . . . 26
- 2.7 Update theory sources based on PIDE markup . . . 28
- 2.8 Explore sessions structure . . . 30

3 Presenting theories **32**
- 3.1 Generating HTML browser information . . . 32
- 3.2 Preparing session root directories . . . 33
- 3.3 Preparing Isabelle session documents . . . 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 `help` . . . 47
4.4.2 Command `echo` . . . 48
4.4.3 Command `shutdown` . . . 48
4.4.4 Command `cancel` . . . 48
4.4.5 Command `session_build` . . . 49
4.4.6 Command `session_start` . . . 51
4.4.7 Command `session_stop` . . . 52
4.4.8 Command `use_theories` . . . 53
4.4.9 Command `purge_theories` . . . 56

5 Isabelle/Scala systems programming 58
5.1 Command-line tools . . . 59
5.1.1 Java Runtime Environment . . . 59
5.1.2 Scala toplevel . . . 59
5.1.3 Scala script wrapper . . . 59
5.1.4 Scala compiler . . . 60
5.2 Isabelle/Scala/Java modules . . . 60
5.2.1 Component configuration via `etc/build.props` . . . 60
5.2.2 Explicit Isabelle/Scala/Java build . . . 62
5.2.3 Project setup for common Scala IDEs . . . 63
5.3 Registered Isabelle/Scala functions . . . 64
5.3.1 Defining functions in Isabelle/Scala . . . 64
5.3.2 Invoking functions in Isabelle/ML . . . 64

5.4 Documenting Isabelle/Scala entities 66

6 Phabricator server setup **69**
6.1 Quick start 70
6.1.1 Initial setup 70
6.1.2 Mailer configuration 72
6.1.3 SSH configuration 72
6.1.4 Internet domain name and HTTPS configuration . . . 73
6.2 Global data storage and backups 74
6.3 Upgrading Phabricator installations 75
6.4 Reference of command-line tools 76
6.4.1 `isabelle phabricator` 76
6.4.2 `isabelle phabricator_setup` 77
6.4.3 `isabelle phabricator_setup_mail` 78
6.4.4 `isabelle phabricator_setup_ssh` 79

7 Miscellaneous tools **81**
7.1 Building Isabelle docker images 81
7.2 Managing Isabelle components 83
7.3 Viewing documentation 84
7.4 Shell commands within the settings environment 85
7.5 Inspecting the settings environment 85
7.6 Mercurial repository setup 86
7.7 Installing standalone Isabelle executables 87
7.8 Creating instances of the Isabelle logo 87
7.9 Output the version identifier of the Isabelle distribution 88

Bibliography **89**

Index **90**

Chapter 1

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/Isabelle2021-1`.

   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]

[1]Cygwin 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 `$USER_HOME/.isabelle`, under rare circumstances this may be changed in the global setting file. Typically, the `ISABELLE_HOME_USER` directory mimics `ISABELLE_HOME` to some extend. In particular, site-wide defaults may be overridden by a private `$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_WINDOWS_PLATFORM64`, `ISABELLE_APPLE_PLATFORM64`.

`ISABELLE_PLATFORM64`* indicates the standard Posix platform (`x86_64`, `arm64`), together with a symbolic name for the operating system (`linux`, `darwin`, `cygwin`).

`ISABELLE_WINDOWS_PLATFORM64`*, `ISABELLE_WINDOWS_PLATFORM32`* indicate the native Windows platform: both 64 bit and 32 bit executables are supported here.

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_APPLE_PLATFORM64`* indicates the native Apple Silicon platform (`arm64-darwin` if available), instead of Intel emulation via Rosetta (`ISABELLE_PLATFORM64=x86_64-darwin`).

`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. "`Isabelle2021-1`".

`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 admissable `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 explicitely by the user. The default value is `HOL`.

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

`ISABELLE_PDFLATEX`, `ISABELLE_LUALATEX`, `ISABELLE_BIBTEX`, `ISABELLE_MAKEINDEX` refer to LaTeX-related tools for Isabelle document preparation (see also §3.3).

`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.

`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 §7.2 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 `etc/settings` via the shell function `isabelle_scala_service`, referring to a suitable instance of class `isabelle.Isabelle_Scala_Tools`. This is the preferred approach for non-trivial systems programming in Isabelle/Scala: instead of adhoc interpretation of `scala` scripts, which is somewhat slow and only type-checked at runtime, there are properly compiled `jar` modules (see also the shell function `classpath` in §5).

There are also various 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.tool (fn () => writeln "OK")'
```

```
isabelle process -e 'Command_Line.tool (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 §5.1.1).

Example

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

```
isabelle_java -Djava.awt.headless=false isabelle.jedit.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 $\mathbf{X} = 5$ and $\mathbf{Y} = 6$ as follows:

XML	YXML
<*name attribute*=*value* ...>	**XY***name***Y***attribute*=*value*...**X**
</*name*>	**XYX**

   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**.

Chapter 2

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.

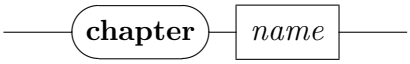
2.1 Session ROOT specifications

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.

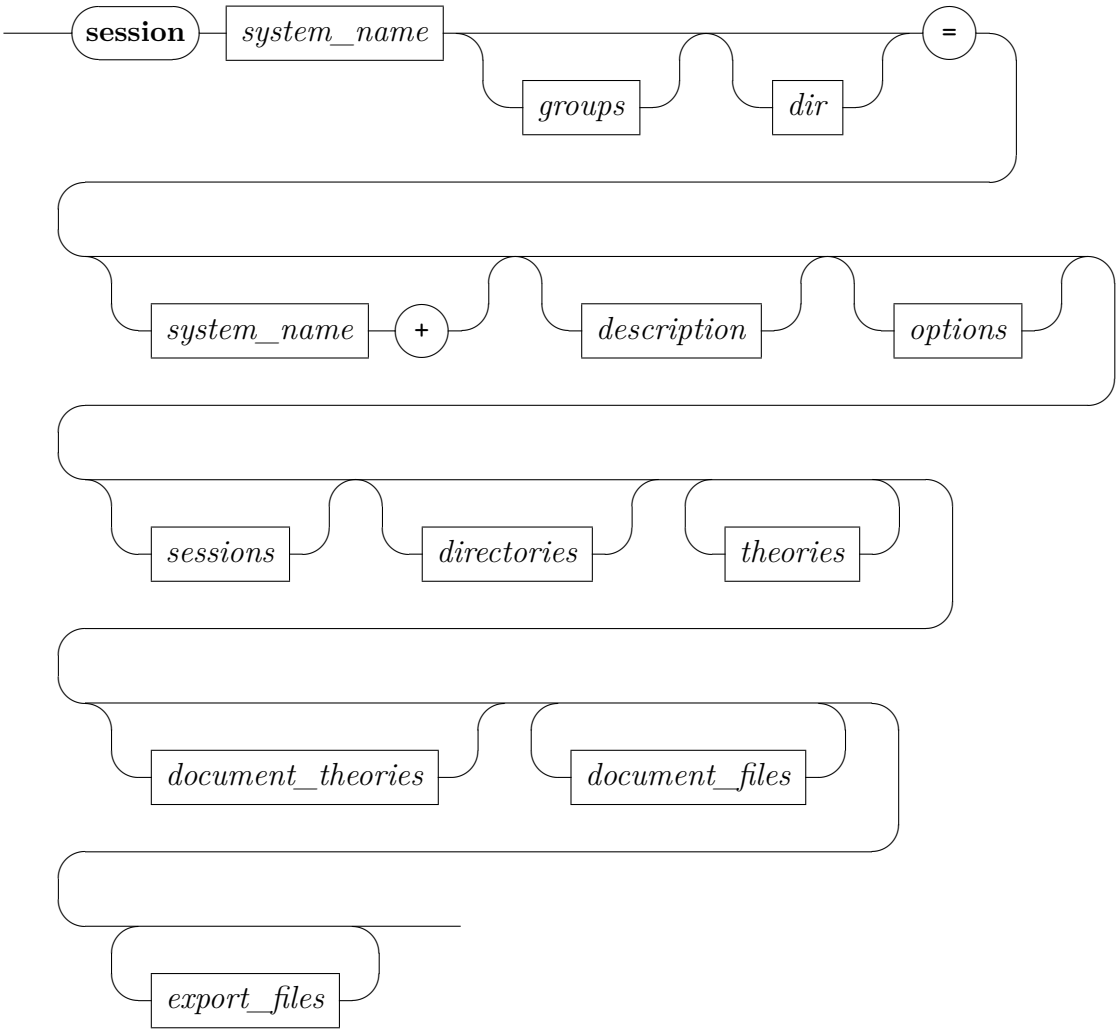
The ROOT file format follows the lexical conventions of the *outer syntax* of Isabelle/Isar, see also [2]. This defines common forms like identifiers, names, quoted strings, verbatim text, nested comments etc. The grammar for *session_chapter* and *session_entry* is given as syntax diagram below; each 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.

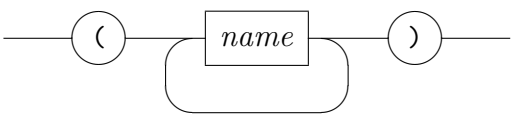
session_chapter



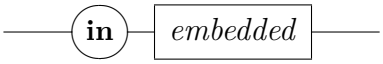
session_entry



groups

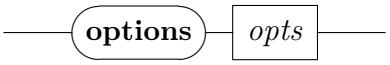


dir

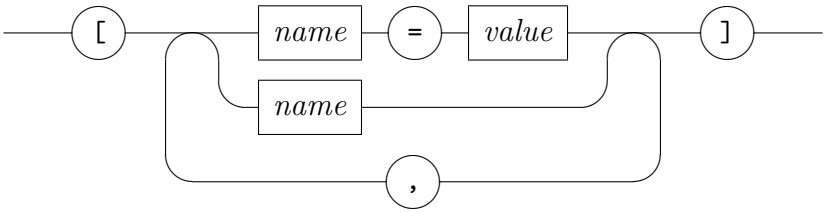


description

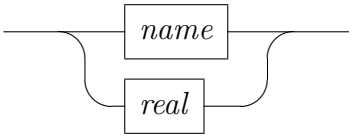
options



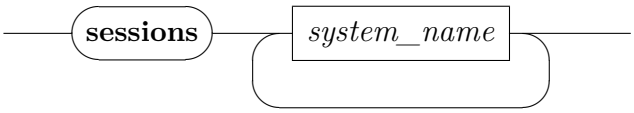
opts



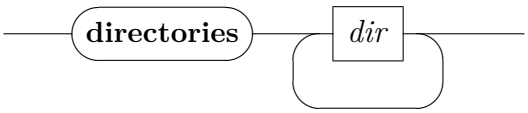
value



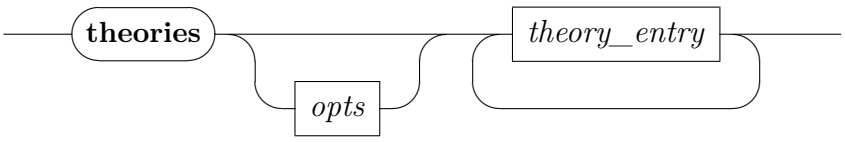
sessions



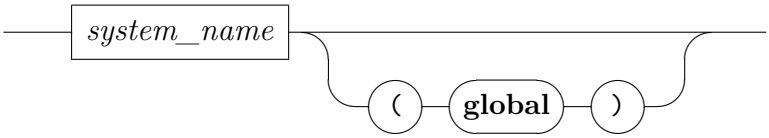
directories



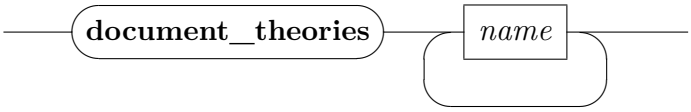
theories



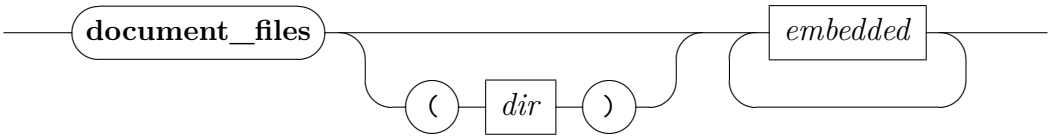
theory_entry



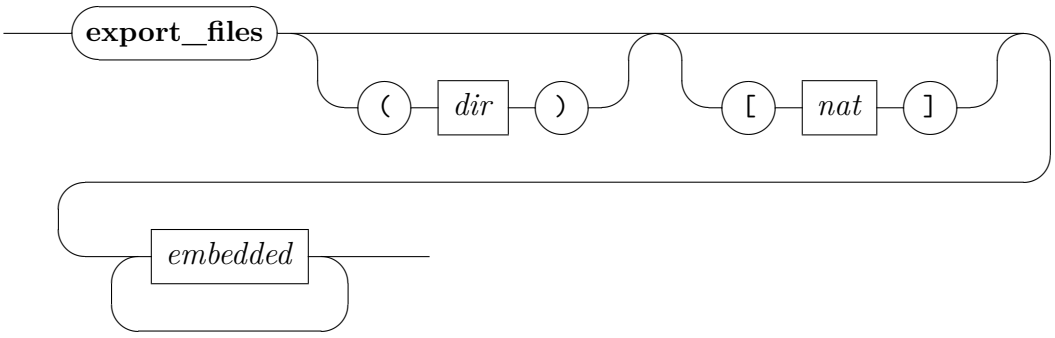
document_theories



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 $z = \mathit{true}$ 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.

directories *dirs* specifies additional directories for import of theory files via **theories** within `ROOT` or **imports** within a theory; *dirs* are relative to the main session directory (cf. **session** ... **in** *dir*). These directories need to be exclusively assigned to a unique session, without implicit sharing of file-system locations.

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_theories *names* specifies theories from other sessions that should be included in the generated document source directory. These theories need to be explicit imports in the current session, or implicit imports from the underlying hierarchy of parent sessions. The generated `session.tex` file is not affected: the session's LaTeX setup needs to `\input{...}` generated `.tex` files separately.

document_files (**in** *base_dir*) *files* lists source files for document preparation, typically `.tex` and `.sty` for LaTeX. 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` controls document output for a particular session or theory; `document=pdf` or `document=true` means enabled, `document=""` or `document=false` means disabled (especially for particular theories).
- `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_echo` informs about document file names during session presentation.
- `document_variants` specifies document variants as a colon-separated list of *name*=*tags* entries. The default name `document`, without additional tags.

  Tags are specified as a comma separated list of modifier/name pairs and tell LaTeX how to interpret certain Isabelle command regions: "`+`*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`.

  In contrast, `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`).
- `document_comment_latex` enables regular LaTeX `comment.sty`, instead of the historic version for plain TeX (default). The latter is much faster, but in conflict with LaTeX classes like Dagstuhl LIPIcs[1].

[1] https://github.com/dagstuhl-publishing/styles

- `document_bibliography` explicitly enables the use of `bibtex`; the default is to check the presence of `root.bib`, but it could have a different name.

- `document_heading_prefix` specifies a prefix for the LaTeX macro names generated from Isar commands like **chapter**, **section** etc. The default is `isamarkup`, e.g. **section** becomes `\isamarkupsection`.

- `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_log` specifies an optional log file for low-level messages produced by `Output.system_message` in Isabelle/ML; the standard value "`-`" refers to console progress of the build job.

- `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:[2]

```
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)
    -P DIR       enable HTML/PDF presentation in directory (":" for
default)
    -R           refer to requirements of selected sessions
    -S           soft build: only observe changes of sources, not heap
images
```

[2]Isabelle/Scala provides the same functionality via `isabelle.Build.build`.

```
  -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)
  -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_TOOL_JAVA_OPTIONS="..."
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 descen-

dants 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 or string options. Multiple occurrences of `-o` on the command-line are applied in the given order.

Option `-P` enables PDF/HTML presentation in the given directory, where "`-P:`" refers to the default `ISABELLE_BROWSER_INFO` (or `ISABELLE_BROWSER_INFO_SYSTEM`). This applies only to explicitly selected sessions; note that option $-R$ allows to select all requirements separately.

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
```

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-Library HOL-Algebra
```

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 Print messages from build database

The `isabelle log` tool prints prover messages from the build database of the given session. Its command-line usage is:

```
Usage: isabelle log [OPTIONS] SESSION

  Options are:
    -T NAME      restrict to given theories (multiple options possible)
    -U           output Unicode symbols
    -m MARGIN    margin for pretty printing (default: 76.0)
    -o OPTION    override Isabelle system OPTION (via NAME=VAL or NAME)
    -v           print all messages, including information etc.

  Print messages from the build database of the given session, without any
  checks against current sources: results from a failed build can be
  printed as well.
```

The specified session database is taken as is, independently of the current session structure and theories sources. The order of messages follows the source positions of source files; thus the erratic evaluation of parallel processing rarely matters. There is *no* implicit build process involved, so it is possible to retrieve error messages from a failed session as well.

Option `-o` allows to change system options, as in `isabelle build` (§2.3). This may affect the storage space for the build database, notably via `system_heaps`, or `build_database_server` and its relatives.

Option `-T` restricts output to given theories: multiple entries are possible by repeating this option on the command-line. The default is to refer to *all* theories that were used in original session build process.

Options `-m` and `-U` modify pretty printing and output of Isabelle symbols. The default is for an old-fashioned ASCII terminal at 80 characters per line (76 + 4 characters to prefix warnings or errors).

Option `-v` prints all messages from the session database that are normally inlined into the source text, including information messages etc.

Examples

Print messages from theory `HOL.Nat` of session `HOL`, using Unicode rendering of Isabelle symbols and a margin of 100 characters:

```
isabelle log -T HOL.Nat -U -m 100 HOL
```

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 subdirectory 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           refer to requirements of selected sessions
    -X NAME      exclude sessions from group NAME and all descendants
    -a           select all sessions
    -b NAME      base logic image (default "Pure")
    -d DIR       include session directory
    -g NAME      select session group NAME
```

```
  -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 `-b` specifies an optional base logic image, for improved scalability of the PIDE session. Its theories are only processed if it is included in the overall session selection.

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 operation `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, with full bootstrap from Isabelle/Pure:

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

Dump all sessions connected to HOL-Analysis, using main Isabelle/HOL as basis:

```
isabelle dump -v -b HOL -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           refer to requirements of selected sessions
    -X NAME      exclude sessions from group NAME and all descendants
    -a           select all sessions
    -b NAME      base logic image (default "Pure")
    -d DIR       include session directory
    -g NAME      select session group NAME
    -o OPTION    override Isabelle system OPTION (via NAME=VAL or NAME)
    -u OPT       overide 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 `-b` specifies an optional base logic image, for improved scalability of the PIDE session. Its theories are only processed if it is included in the overall session selection.

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` (and ancestors):

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

Update the same for all application sessions based on `HOL-Analysis` — using its image is taken starting point (for reduced resource requirements):

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

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.

2.8 Explore sessions structure

The `isabelle sessions` tool explores the sessions structure. Its command-line usage is:

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

  Options are:
    -B NAME      include session NAME and all descendants
    -D DIR       include session directory and select its sessions
    -R           refer to 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
    -x NAME      exclude session NAME and all descendants

  Explore the structure of Isabelle sessions and print result names in
  topological order (on stdout).
```

Arguments and options for session selection resemble `isabelle build` (§2.3).

Examples

All sessions of the Isabelle distribution:

```
isabelle sessions -a
```

Sessions that are based on `ZF` (and required by it):

```
isabelle sessions -B ZF
```

All sessions of Isabelle/AFP (based in directory `AFP`):

```
isabelle sessions -D AFP/thys
```

Sessions required by Isabelle/AFP (based in directory `AFP`):

```
isabelle sessions -R -D AFP/thys
```

Chapter 3

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 command-line tools `isabelle mkroot` and `isabelle build` provide the primary means for managing Isabelle sessions, including options for presentation: "`document=pdf`" generates PDF output from the theory session, and "`document_output=dir`" emits a copy of the document sources with the PDF into the given directory (relative to the session directory).

Alternatively, `isabelle document` may be used to turn the generated LaTeX sources of a session (exports from its build database) into PDF.

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 theory documents in PDF. These are prepared automatically as well if enabled like this:

```
isabelle build -o browser_info -o document -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:
    -A LATEX     provide author in LaTeX notation (default: user name)
    -I           init Mercurial repository and add generated files
    -T LATEX     provide title in LaTeX notation (default: session name)
    -n NAME      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. Its usage is:

```
Usage: isabelle document [OPTIONS] SESSION

  Options are:
    -O DIR       output directory for LaTeX sources and resulting PDF
    -P DIR       output directory for resulting PDF
    -S DIR       output directory for LaTeX sources
    -V           verbose latex
    -d DIR       include session directory
    -o OPTION    override Isabelle system OPTION (via NAME=VAL or NAME)
    -v           verbose build

  Prepare the theory document of a session.
```

Generated LaTeX sources are taken from the session build database: `isabelle build` is invoked beforehand to ensure that it is up-to-date. Further files are generated on the spot, notably essential Isabelle style files, and `session.tex` to input all theory sources from the session (excluding imports from other sessions).

Options `-d`, `-o`, `-v` have the same meaning as for `isabelle build`.

Option `-V` prints full output of LaTeX tools.

Option `-O` *dir* specifies the output directory for generated LaTeX sources and the result PDF file. Options `-P` and `-S` only refer to the PDF and sources, respectively.

For example, for output directory "`output`" and the default document variant "`document`", the generated document sources are placed into the subdirectory `output/document/` and the resulting PDF into `output/document.pdf`.

Isabelle is usually smart enough to create the PDF from the given `root.tex` and optional `root.bib` (bibliography) and `root.idx` (index) using standard LaTeX tools. Actual command-lines are given by settings `ISABELLE_LUALATEX` (or `ISABELLE_PDFLATEX`), `ISABELLE_BIBTEX`, `ISABELLE_MAKEINDEX`: these variables are used without quoting in shell scripts, and thus may contain additional options.

The system option `document_build` specifies an alternative build engine, e.g. within the session `ROOT` file as "`options [document_build = pdflatex]`". The following standard engines are available:

- `lualatex` (default) uses the shell command `$ISABELLE_LUALATEX` on the main `root.tex` file, with further runs of `$ISABELLE_BIBTEX` and `$ISABELLE_MAKEINDEX` as required.
- `pdflatex` uses `$ISABELLE_PDFLATEX` instead of `$ISABELLE_LUALATEX`, and the other tools as above.
- `build` invokes an executable script of the same name in a private directory containing all **document_files** and other generated document sources. The script is invoked as "`./build pdf` *name*" for the document variant name; it needs to produce a corresponding *name*`.pdf` file by arbitrary means on its own.

Further engines can be defined by add-on components in Isabelle/Scala (§5.2), providing a service class derived from `isabelle.Document_Build.Engine`. Available classes are listed in `isabelle.Document_Build.engines`.

Examples

Produce the document from session `FOL` with full verbosity, and a copy in the current directory (subdirectory `document` and file `document.pdf`):

```
isabelle document -v -V -O. FOL
```

Chapter 4

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

[1] This 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`)

[2]See 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, bools, 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:

[3] See 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` ...

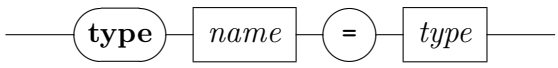
All asynchronous messages are decorated with the task identifier that was revealed in the immediate (synchronous) result. Thus the client can invoke further asynchronous commands and still dispatch the resulting stream of asynchronous messages properly.

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.

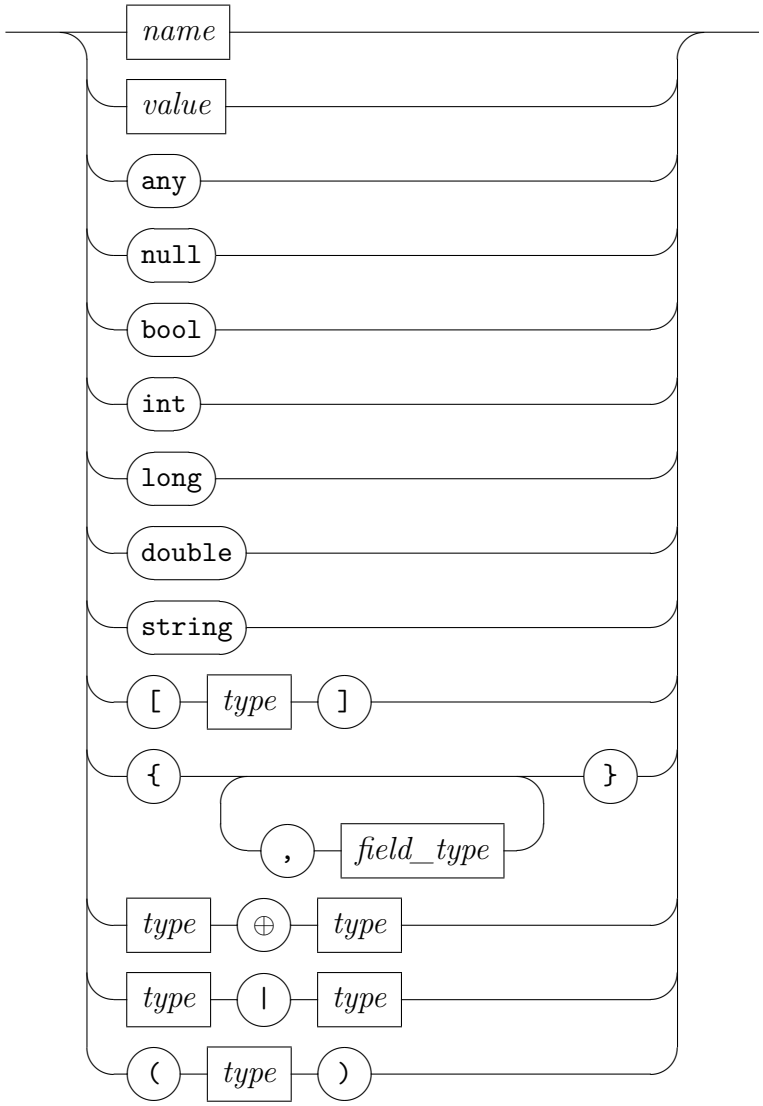
4.3 Types for JSON values

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

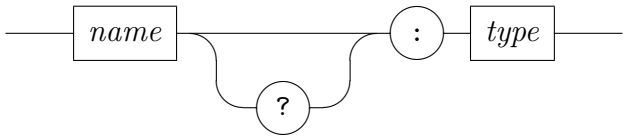
type_def



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.

[5] See also "Null References: The Billion Dollar Mistake" by Tony Hoare https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare.

[6] Implementations 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 $t_1 \mid t_2$ 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

[7] See https://www.ietf.org/rfc/rfc4122.txt and https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/UUID.html.

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

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` *task*
notifications:	`NOTE` *task* $\oplus$ (*theory_progress* | *message*)
regular result:	`FINISHED` *task* $\oplus$ *session_id* $\oplus$ {*tmp_dir*: *string*}
error result:	`FAILED` *task* $\oplus$ *error_message*

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:	*session_id*
immediate result:	`OK` *task*
regular result:	`FINISHED` *task* $\oplus$ *session_stop_result*
error result:	`FAILED` *task* $\oplus$ *error_message* $\oplus$ *session_stop_result*

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

The `session_stop` command forces a shutdown of the identified PIDE session. This asynchronous tasks 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*, *base*64: *bool*, *body*: *string*}
type *node_results* =
  {*status*: *node_status*, *messages*: [*message*], *exports*: [*export*]}
type *nodes_status* =
  [*node* ⊕ {*status*: *node_status*}]
type *use_theories_results* =
  {*ok*: *bool*,
   errors: [*message*],
   nodes: [*node* ⊕ *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 *base*64 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* $\times$ *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/Examples/Seq.thy"`) or session-qualified theory name (e.g. `"HOL-Examples.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 ∀. 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/Examples/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*.

Chapter 5

Isabelle/Scala systems programming

Isabelle/ML and Isabelle/Scala are the two main implementation languages of the Isabelle environment:

- Isabelle/ML is for *mathematics*, to develop tools within the context of symbolic logic, e.g. for constructing proofs or defining domain-specific formal languages. See the *Isabelle/Isar implementation manual* [1] for more details.
- Isabelle/Scala is for *physics*, to connect with the world of systems and services, including editors and IDE frameworks.

There are various ways to access Isabelle/Scala modules and operations:

- Isabelle command-line tools (§5.1) run in a separate Java process.
- Isabelle/ML antiquotations access Isabelle/Scala functions (§5.3) via the PIDE protocol: execution happens within the running Java process underlying Isabelle/Scala.
- The `Console/Scala` plugin of Isabelle/jEdit [3] operates on the running Java application, using the Scala read-eval-print-loop (REPL).

The main Isabelle/Scala/jEdit functionality is provided by `$ISABELLE_HOME/lib/classes/isabelle.jar`. Further underlying Scala and Java libraries are bundled with Isabelle, e.g. to access SQLite or PostgreSQL via JDBC.

Add-on Isabelle components may augment the system environment by providing suitable configuration in `etc/settings` (GNU bash script). The shell function `classpath` helps to write `etc/settings` in a portable manner: it refers to library `jar` files in standard POSIX path notation. On Windows, this is converted to native platform format, before invoking Java (§5.1).

There is also an implicit build process for Isabelle/Scala/Java modules, based on `etc/build.props` within the component directory (see also §5.2).

5.1 Command-line tools

5.1.1 Java Runtime Environment

The `isabelle java` tool is a direct wrapper for the Java Runtime Environment, within the regular Isabelle settings environment (§1.1) and Isabelle classpath. The command line arguments are that of the bundled Java distribution: see option `-help` in particular.

The `java` executable is taken from `ISABELLE_JDK_HOME`, according to the standard directory layout for regular distributions of OpenJDK.

The shell function `isabelle_jdk` allows shell scripts to invoke other Java tools robustly (e.g. `isabelle_jdk jar`), without depending on accidental operating system installations.

5.1.2 Scala toplevel

The `isabelle scala` tool is a direct wrapper for the Scala toplevel, similar to `isabelle java` above. The command line arguments are that of the bundled Scala distribution: see option `-help` in particular. This allows to interact with Isabelle/Scala interactively.

Example

Explore the Isabelle system environment in Scala:

```
$ isabelle scala

import isabelle._

val isabelle_home = Isabelle_System.getenv("ISABELLE_HOME")

val options = Options.init()
options.bool("browser_info")
options.string("document")
```

5.1.3 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.

5.1.4 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 bundled Scala distribution.

This provides a low-level mechanism 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 shell function `classpath`.

A more convenient high-level approach works via `etc/build.props` (see §5.2).

5.2 Isabelle/Scala/Java modules

5.2.1 Component configuration via `etc/build.props`

Isabelle components may augment the Isabelle/Scala/Java environment declaratively via properties given in `etc/build.props` (within the component directory). This specifies an output `jar` *module*, based on Scala or Java *sources*, and arbitrary *resources*. Moreover, a module can specify *services* that are subclasses of `isabelle.Isabelle_System.Service`; these have a particular meaning to Isabelle/Scala tools.

Before running a Scala or Java process, the Isabelle system implicitly ensures that all provided modules are compiled and packaged (as jars). It is also possible to invoke `isabelle scala_build` explicitly, with extra options.

The syntax of in `etc/build.props` follows a regular Java properties file[1],

[1] https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/Properties.html#load(java.io.Reader)

but the encoding is `UTF-8`, instead of historic `ISO 8859-1` from the API documentation.

The subsequent properties are relevant for the Scala/Java build process. Most properties are optional: the default is an empty string (or list). File names are relative to the main component directory and may refer to Isabelle settings variables (e.g. `$ISABELLE_HOME`).

- `title` (required) is a human-readable description of the module, used in printed messages.
- `module` specifies a `jar` file name for the output module, as result of the specified sources (and resources). If this is absent (or `no_build` is set, as described below), there is no implicit build process. The contributing sources might be given nonetheless, notably for `isabelle scala_project` (§5.2.3), which includes Scala/Java sources of components, while suppressing `jar` modules (to avoid duplication of program content).
- `no_build` is a Boolean property, with default `false`. If set to `true`, the implicit build process for the given `module` is *omitted* — it is assumed to be provided by other means.
- `scalac_options` and `javac_options` augment the default settings `ISABELLE_SCALAC_OPTIONS` and `ISABELLE_JAVAC_OPTIONS` for this component; option syntax follows the regular command-line tools `scalac` and `javac`, respectively.
- `main` specifies the main entry point for the `jar` module. This is only relevant for direct invocation like "`java -jar test.jar`".
- `requirements` is a list of `jar` modules that are needed in the compilation process, but not provided by the regular classpath (notably `ISABELLE_CLASSPATH`).

  A *normal entry* refers to a single `jar` file name, possibly with settings variables as usual. E.g. `$ISABELLE_SCALA_JAR` for the main `$ISABELLE_HOME/lib/classes/isabelle.jar` (especially relevant for add-on modules).

  A *special entry* is of of the form `env:`*variable* and refers to a settings variable from the Isabelle environment: its value may consist of multiple `jar` entries (separated by colons). Environment variables are not expanded recursively.

- `resources` is a list of files that should be included in the resulting `jar` file. Each item consists of a pair separated by colon: *source*`:`*target* means to copy an existing source file (relative to the component directory) to the given target file or directory (relative to the `jar` name space). A *file* specification without colon abbreviates *file*`:`*file*, i.e. the file is copied while retaining its relative path name.

- `sources` is a list of `.scala` or `.java` files that contribute to the specified module. It is possible to use both languages simultaneously: the Scala and Java compiler will be invoked consecutively to make this work.

- `services` is a list of class names to be registered as Isabelle service providers (subclasses of `isabelle.Isabelle_System.Service`). Internal class names of the underlying JVM need to be given: e.g. see method `java.lang.Object.getClass`.

  Particular services require particular subclasses: instances are filtered according to their dynamic type. For example, class `isabelle.Isabelle_Scala_Tools` collects Scala command-line tools, and class `isabelle.Scala.Functions` collects Scala functions (§5.3).

5.2.2 Explicit Isabelle/Scala/Java build

The `isabelle scala_build` tool explicitly invokes the build process for all registered components.

```
Usage: isabelle scala_build [OPTIONS]

  Options are:
    -f           force fresh build
    -q           quiet mode: suppress stdout/stderr

  Build Isabelle/Scala/Java modules of all registered components
  (if required).
```

For each registered Isabelle component that provides `etc/build.props`, the specified output `module` is checked against the corresponding input `requirements`, `resources`, `sources`. If required, there is an automatic build using `scalac` or `javac` (or both). The identity of input files is recorded within the output `jar`, using SHA1 digests in `META-INF/isabelle/shasum`.

Option `-f` forces a fresh build, regardless of the up-to-date status of input files vs. the output module.

Option `-q` suppresses all output on stdout/stderr produced by the Scala or Java compiler.

Explicit invocation of `isabelle scala_build` mainly serves testing or applications with special options: the Isabelle system normally does an automatic the build on demand.

5.2.3 Project setup for common Scala IDEs

The `isabelle scala_project` tool creates a project configuration for all Isabelle/Scala/Java modules specified in components via `etc/build.props`, together with additional source files given on the command-line:

```
Usage: isabelle scala_project [OPTIONS] [MORE_SOURCES ...]

  Options are:
    -D DIR       project directory (default: "$ISABELLE_HOME_USER/scala_project")
    -L           make symlinks to original source files
    -f           force update of existing directory

  Setup Maven project for Isabelle/Scala/jEdit --- to support common IDEs
  such as IntelliJ IDEA.
```

The generated configuration is for Maven[2], but the main purpose is to import it into common IDEs, such as IntelliJ IDEA[3]. This allows to explore the sources with static analysis and other hints in real-time.

The generated files refer to physical file-system locations, using the path notation of the underlying OS platform. Thus the project needs to be recreated whenever the Isabelle installation is changed or moved.

Option `-L` produces *symlinks* to the original files: this allows to develop Isabelle/Scala/jEdit modules within an external IDE. The default is to *copy* source files, so editing them within the IDE has no permanent effect on the originals.

Option `-D` specifies an explicit project directory, instead of the default `$ISABELLE_HOME_USER/scala_project`. Option `-f` forces an existing project directory to be *purged* — after some sanity checks that it has been generated by `isabelle scala_project` before.

[2]https://maven.apache.org

[3]https://www.jetbrains.com/idea

Examples

Create a project directory and for editing the original sources:

```
isabelle scala_project -f -L
```

On Windows, this usually requires Administrator rights, in order to create native symlinks.

5.3 Registered Isabelle/Scala functions

5.3.1 Defining functions in Isabelle/Scala

The service class `isabelle.Scala.Functions` collects Scala functions of type `isabelle.Scala.Fun`: by registering instances via `services` in `etc/build.props` (§5.2), it is becomes possible to invoke Isabelle/Scala from Isabelle/ML (see below).

An example is the predefined collection of `isabelle.Scala.Functions` in `$ISABELLE_HOME/etc/build.props`. The overall list of registered functions is accessible in Isabelle/Scala as `isabelle.Scala.functions`.

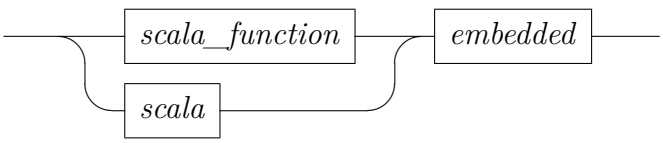
The general class `isabelle.Scala.Fun` expects a multi-argument / multi-result function `List[isabelle.Bytes] => List[isabelle.Bytes]`; more common are instances of `isabelle.Scala.Fun_Strings` for type `List[String] => List[String]`, or `isabelle.Scala.Fun_String` for type `String => String`.

5.3.2 Invoking functions in Isabelle/ML

Isabelle/PIDE provides a protocol to invoke registered Scala functions in ML: this works both within the Prover IDE and in batch builds.

The subsequent ML antiquotations refer to Scala functions in a formally-checked manner.

scala_function : *ML_antiquotation*
scala : *ML_antiquotation*



@{*scala_function name*} inlines the checked function name as ML string literal.

@{*scala name*} and @{*scala_thread name*} invoke the checked function via the PIDE protocol. In Isabelle/ML this appears as a function of type `string list -> string list` or `string -> string`, depending on the definition in Isabelle/Scala. Evaluation is subject to interrupts within the ML runtime environment as usual. A `null` result in Scala raises an exception `Scala.Null` in ML. The execution of @{*scala*} works via a Scala future on a bounded thread farm, while @{*scala_thread*} always forks a separate Java thread.

The standard approach of representing datatypes via strings works via XML in YXML transfer syntax. See Isabelle/ML operations and modules `YXML.string_of_body`, `YXML.parse_body`, `XML.Encode`, `XML.Decode`; similarly for Isabelle/Scala. Isabelle symbols may have to be recoded via Scala operations `isabelle.Symbol.decode` and `isabelle.Symbol.encode`.

Examples

Invoke the predefined Scala function `echo`:

```
ML ‹
  val s = "test";
  val s' = scala‹echo› s;
  assert (s = s')
›
```

Let the Scala compiler process some toplevel declarations, producing a list of errors:

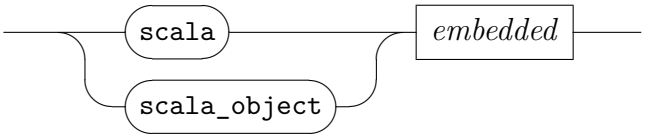
```
ML ‹
  val source = "class A(a: Int, b: Boolean)"
  val errors =
    scala‹scala_toplevel› source
    |> YXML.parse_body
    |> let open XML.Decode in list string end;

  assert (null errors)›
```

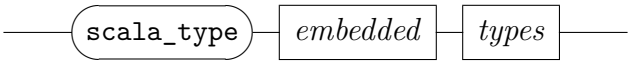
The above is merely for demonstration. See `Scala_Compiler.toplevel` for a more convenient version with builtin decoding and treatment of errors.

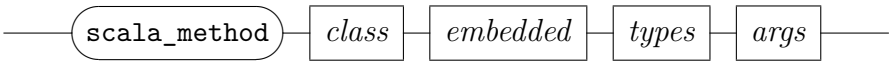
5.4 Documenting Isabelle/Scala entities

The subsequent document antiquotations help to document Isabelle/Scala entities, with formal checking of names against the Isabelle classpath.

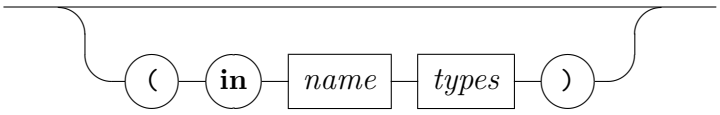
scala : *antiquotation*
scala_object : *antiquotation*
scala_type : *antiquotation*
scala_method : *antiquotation*



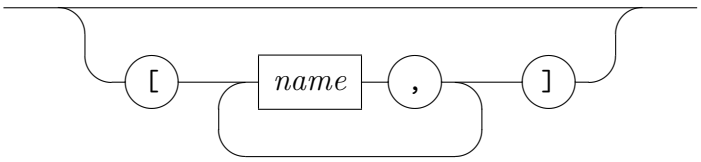




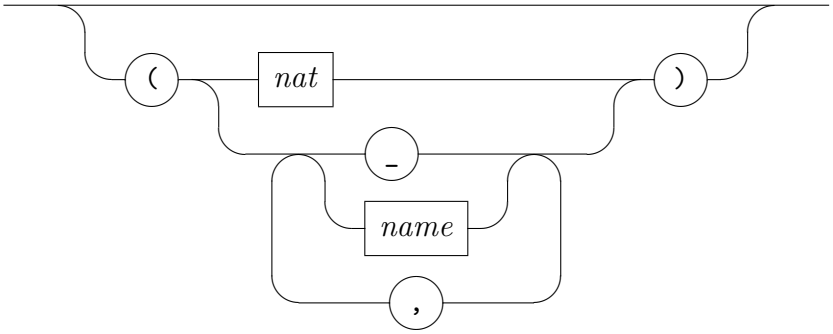
class



types



args



@{*scala s*} is similar to @{*verbatim s*}, but the given source text is checked by the Scala compiler as toplevel declaration (without evaluation). This allows to write Isabelle/Scala examples that are statically checked.

@{*scala_object x*} checks the given Scala object name (simple value or ground module) and prints the result verbatim.

@{*scala_type* $T[A]$} checks the given Scala type name (with optional type parameters) and prints the result verbatim.

@{*scala_method* (*in* $c[A]$) $m[B](n)$} checks the given Scala method m in the context of class c. The method argument slots are either specified by a number n or by a list of (optional) argument types; this may refer to type variables specified for the class or method: A or B above.

Everything except for the method name m is optional. The absence of the class context means that this is a static method. The absence of arguments with types means that the method can be determined uniquely as `(`m `_)` in Scala (no overloading).

Examples

Miscellaneous Isabelle/Scala entities:

- object: `isabelle.Isabelle_Process`
- type without parameter: `isabelle.Console_Progress`
- type with parameter: `List[A]`
- static method: `isabelle.Isabelle_System.bash`

- class and method with type parameters: `List[A].map`
- overloaded method with argument type: `Int.+`

Chapter 6

Phabricator server setup

Phabricator[1] is an open-source product to support the development process of complex software projects (open or closed ones). The official slogan is:

> Discuss. Plan. Code. Review. Test.
> Every application your project needs, all in one tool.

Ongoing changes and discussions about changes are maintained uniformly within a MySQL database. There are standard connections to major version control systems: **Subversion**, **Mercurial**, **Git**. So Phabricator offers a counter-model to trends of monoculture and centralized version control, especially due to Microsoft's Github and Atlassian's Bitbucket.

The small company behind Phabricator provides paid plans for support and hosting of servers, but it is easy to do *independent self-hosting* on a standard LAMP server (Linux, Apache, MySQL, PHP). This merely requires a virtual machine on the Net, which can be rented cheaply from local hosting providers — there is no need to follow big cloud corporations. So it is feasible to remain the master of your virtual home, following the slogan "own all your data". In many respects, Phabricator is similar to the well-known Nextcloud[2] product, concerning both the technology and sociology.

The following Phabricator instances may serve as examples:

- Phabricator development https://secure.phabricator.com
- Wikimedia development https://phabricator.wikimedia.org
- Blender development https://developer.blender.org
- LLVM development https://reviews.llvm.org
- Mozilla development https://phabricator.services.mozilla.com

[1] https://www.phacility.com/phabricator
[2] https://nextcloud.org

- Mercurial development https://phab.mercurial-scm.org
- Isabelle development https://isabelle-dev.sketis.net

Initial Phabricator configuration requires many details to be done right. Isabelle provides some command-line tools to help with the setup, and afterwards Isabelle support is optional: it is possible to run and maintain the server, without requiring the somewhat bulky Isabelle distribution again.

Assuming an existing Phabricator installation, the command-line tool `isabelle hg_setup` (§7.6) helps to create new repositories or to migrate old ones. In particular, this avoids the lengthy sequence of clicks in Phabricator to make a new private repository with hosting on the server. (Phabricator is a software project management platform, where initial repository setup happens rarely in practice.)

6.1 Quick start

The starting point is a fresh installation of **Ubuntu 20.04 LTS**[3]: this version is mandatory due to subtle dependencies on system packages and configuration that is assumed by the Isabelle setup tool.

For production use, a proper *Virtual Server* or *Root Server* product from a hosting provider will be required, including an Internet Domain Name (§6.1.4).

Initial experimentation also works on a local host, e.g. via VirtualBox[4]. The Internet domain `lvh.me` is used by default: it maps arbitrary subdomains to `localhost`.

All administrative commands need to be run as `root` user (e.g. via `sudo`). Note that Isabelle refers to user-specific configuration in the user home directory via `ISABELLE_HOME_USER` (§1.1); that may be different or absent for the root user and thus cause confusion.

6.1.1 Initial setup

Isabelle can manage multiple named Phabricator installations: this allows to separate administrative responsibilities, e.g. different approaches to user management for different projects. Subsequently we always use the default

[3]https://ubuntu.com/download
[4]https://www.virtualbox.org

name "`vcs`": the name will appear in file and directory locations, internal database names and URLs.

The initial setup works as follows (with full Linux package upgrade):

```
isabelle phabricator_setup -U -M:
```

After installing many packages, cloning the Phabricator distribution, initializing the MySQL database and Apache, the tool prints an URL for further configuration. Now the following needs to be provided by the web interface.

- An initial user that will get administrator rights. There is no need to create a special `admin` account. Instead, a regular user that will take over this responsibility can be used here. Subsequently we assume that user `makarius` becomes the initial administrator.

- An *Auth Provider* to manage user names and passwords. None is provided by default, and Phabricator points out this omission prominently in its overview of *Setup Issues*: following these hints quickly leads to the place where a regular *Username/Password* provider can be added.

  Alternatively, Phabricator can delegate the responsibility of authentication to big corporations like Google and Facebook, but these can be easily ignored. Genuine self-hosting means to manage users directly, without outsourcing of authentication.

- A proper password for the administrator can now be set, e.g. by the following command:

  ```
  isabelle phabricator bin/auth recover makarius
  ```

  The printed URL gives access to a login and password dialog in the web interface.

  Any further users will be able to provide a password directly, because the Auth Provider is already active.

- The list of Phabricator **Setup Issues** should be studied with some care, to make sure that no serious problems are remaining. For example, the request to lock the configuration can be fulfilled as follows:

  ```
  isabelle phabricator bin/auth lock
  ```

  A few other Setup Issues might be relevant as well, e.g. the timezone of the server. Some more exotic points can be ignored: Phabricator provides careful explanations about what it thinks could be wrong, while leaving some room for interpretation.

6.1.2 Mailer configuration

The next important thing is messaging: Phabricator needs to be able to communicate with users on its own account, e.g. to reset passwords. The documentation has many variations on *Configuring Outbound Email*[5], but a conventional SMTP server with a dedicated `phabricator` user is sufficient. There is no need to run a separate mail server on the self-hosted Linux machine: hosting providers often include such a service for free, e.g. as part of a web-hosting package. As a last resort it is also possible to use a corporate service like Gmail, but such dependency dilutes the whole effort of self-hosting.

Mailer configuration requires a few command-line invocations as follows:

```
isabelle phabricator_setup_mail
```

This generates a JSON template file for the the mail account details. After editing that, the subsequent command will add and test it with Phabricator:

```
isabelle phabricator_setup_mail -T makarius
```

This tells Phabricator to send a message to the administrator created before; the output informs about success or errors.

The mail configuration process can be refined and repeated until it works properly: host name, port number, protocol etc. all need to be correct. The `key` field in the JSON file identifies the name of the configuration that will be overwritten each time, when taking over the parameters via `isabelle phabricator_setup_mail`.

The effective mail configuration can be queried like this:

```
isabelle phabricator bin/config get cluster.mailers
```

6.1.3 SSH configuration

SSH configuration is important to access hosted repositories with public-key authentication. It is done by a separate tool, because it affects the operating-system and all installations of Phabricator simultaneously.

The subsequent configuration is convenient (and ambitious): it takes away the standard port 22 from the operating system and assigns it to Isabelle/Phabricator.

[5] https://secure.phabricator.com/book/phabricator/article/configuring_outbound_email

```
isabelle phabricator_setup_ssh -p 22 -q 222
```

Afterwards, remote login to the server host needs to use that alternative port 222. If there is a problem connecting again, the administrator can usually access a remote console via some web interface of the virtual server provider.

The following alternative is more modest: it uses port 2222 for Phabricator, and retains port 22 for the operating system.

```
isabelle phabricator_setup_ssh -p 2222 -q 22
```

The tool can be invoked multiple times with different parameters; ports are changed back and forth each time and services restarted.

6.1.4 Internet domain name and HTTPS configuration

So far the Phabricator server has been accessible only on `localhost` (via the alias `lvh.me`). Proper configuration of a public Internet domain name (with HTTPS certificate from *Let's Encrypt*) works as follows.

- Register a subdomain (e.g. `vcs.example.org`) as an alias for the IP address of the underlying Linux host. This usually works by some web interface of the hosting provider to edit DNS entries; it might require some time for updated DNS records to become publicly available.
- Edit the Phabricator website configuration file in `/etc/apache2/sites-available/` to specify `ServerName` and `ServerAdmin` like this:

  ```
  ServerName vcs.example.org
  ServerAdmin webmaster@example.org
  ```

  Then reload (or restart) Apache like this:

  ```
  systemctl reload apache2
  ```

- Install `certbot` from https://certbot.eff.org following the description for Apache and Ubuntu 20.04 on https://certbot.eff.org/lets-encrypt/ubuntubionic-apache. Run `certbot` interactively and let it operate on the domain `vcs.example.org`.
- Inform Phabricator about its new domain name like this:

```
isabelle phabricator bin/config set \
  phabricator.base-uri https://vcs.example.org
```

- Visit the website `https://vcs.example.org` and configure Phabricator as described before. The following options are particularly relevant for a public website:
  - *Auth Provider / Username/Password*: disable *Allow Registration* to avoid uncontrolled registrants; users can still be invited via email instead.
  - Enable `policy.allow-public` to allow read-only access to resources, without requiring user registration.
- Adjust `phabricator.cookie-prefix` for multiple installations with overlapping domains (see also the documentation of this configuration option within Phabricator).

6.2 Global data storage and backups

The global state of a Phabricator installation consists of two main parts:

1. The *root directory* according to `/etc/isabelle-phabricator.conf` or `isabelle phabricator -l`: it contains the main PHP program suite with administrative tools, and some configuration files. The default setup also puts hosted repositories here (subdirectory `repo`).
2. Multiple *MySQL databases* with a common prefix derived from the installation name — the same name is used as database user name.

The root user may invoke `/usr/local/bin/isabelle-phabricator-dump` to create a complete database dump within the root directory. Afterwards it is sufficient to make a conventional **file-system backup** of everything. To restore the database state, see the explanations on `mysqldump` in https://secure.phabricator.com/book/phabricator/article/configuring_backups; some background information is in https://secure.phabricator.com/book/phabflavor/article/so_many_databases.

The following command-line tools are particularly interesting for advanced database maintenance (within the Phabricator root directory):

```
phabricator/bin/storage help dump
phabricator/bin/storage help shell
phabricator/bin/storage help destroy
phabricator/bin/storage help renamespace
```

For example, copying a database snapshot from one installation to another works as follows. Run on the first installation root directory:

```
phabricator/bin/storage dump > dump1.sql
phabricator/bin/storage renamespace --from phabricator_vcs \
  --to phabricator_xyz --input dump1.sql --output dump2.sql
```

Them run on the second installation root directory:

```
phabricator/bin/storage destroy
phabricator/bin/storage shell < .../dump2.sql
```

Local configuration in `phabricator/config/local/` and hosted repositories need to be treated separately within the file-system. For the latter see also these tools:

```
phabricator/bin/repository help list-paths
phabricator/bin/repository help move-paths
```

6.3 Upgrading Phabricator installations

The Phabricator developers publish a new version approx. every 1–4 weeks: see also https://secure.phabricator.com/w/changelog. There is no need to follow such frequent updates on the spot, but it is a good idea to upgrade occasionally — with the usual care to avoid breaking a production system (see also §6.2 for database dump and backup).

The Isabelle/Phabricator setup provides a convenience tool to upgrade all installations uniformly:

```
/usr/local/bin/isabelle-phabricator-upgrade
```

This refers to the `stable` branch of the distribution repositories by default. Alternatively, it also possible to use the `master` like this:

```
/usr/local/bin/isabelle-phabricator-upgrade master
```

See https://secure.phabricator.com/book/phabricator/article/upgrading for further explanations on Phabricator upgrade.

6.4 Reference of command-line tools

The subsequent command-line tools usually require root user privileges on the underlying Linux system (e.g. via `sudo bash` to open a subshell, or directly via `sudo isabelle phabricator ...`).

6.4.1 `isabelle phabricator`

The `isabelle phabricator` tool invokes a GNU bash command-line within the Phabricator home directory:

```
Usage: isabelle phabricator [OPTIONS] COMMAND [ARGS...]

  Options are:
    -l           list available Phabricator installations
    -n NAME      Phabricator installation name (default: "vcs")

  Invoke a command-line tool within the home directory of the named
  Phabricator installation.
```

Isabelle/Phabricator installations are registered in the global configuration file `/etc/isabelle-phabricator.conf`, with name and root directory separated by colon (no extra whitespace). The home directory is the subdirectory `phabricator` within the root.

Option `-l` lists the available Phabricator installations with name and root directory — without invoking a command.

Option `-n` selects the explicitly named Phabricator installation.

Examples

Print the home directory of the Phabricator installation:

```
isabelle phabricator pwd
```

Print some Phabricator configuration information:

```
isabelle phabricator bin/config get phabricator.base-uri
```

The latter conforms to typical command templates seen in the original Phabricator documentation:

```
phabricator/ $ ./bin/config get phabricator.base-uri
```

Here the user is meant to navigate to the Phabricator home manually, in contrast to `isabelle phabricator` doing it automatically thanks to the global configuration `/etc/isabelle-phabricator.conf`.

6.4.2 isabelle phabricator_setup

The `isabelle phabricator_setup` tool installs a fresh Phabricator instance on Ubuntu 20.04 LTS:

```
Usage: isabelle phabricator_setup [OPTIONS]

  Options are:
    -M SOURCE    install Mercurial from source: local PATH, or URL, or ":"
    -R DIR       repository directory (default: "/var/www/phabricator-NAME/repo")
    -U           full update of system packages before installation
    -n NAME      Phabricator installation name (default: "vcs")
    -o OPTION    override Isabelle system OPTION (via NAME=VAL or NAME)
    -r DIR       installation root directory (default: "/var/www/phabricator-NAME")

  Install Phabricator as LAMP application (Linux, Apache, MySQL, PHP).

  The installation name (default: "vcs") is mapped to a regular
  Unix user; this is relevant for public SSH access.
```

Installation requires Linux root permissions. All required packages are installed automatically beforehand, this includes the Apache web server and the MySQL database engine.

Global configuration in `/etc` or a few other directories like `/var/www` uses name prefixes like `isabelle-phabricator` or `phabricator`. Local configuration for a particular installation uses more specific names derived from `phabricator-`*NAME*, e.g. `/var/www/phabricator-vcs` for the default.

Knowing the naming conventions, it is possible to purge a Linux installation from Isabelle/Phabricator with some effort, but there is no automated procedure for de-installation. In the worst case, it might be better to re-install the virtual machine from a clean image.

Option `-U` ensures a full update of system packages, before installing further packages required by Phabricator. This might require a reboot.

Option `-M`: installs a standard Mercurial release from source — the one that is used by the Phabricator hosting service https://admin.phacility.com. This avoids various problems with the package provided by Ubuntu 20.04. Alternatively, an explicit file path or URL the source archive (`.tar.gz`) may be given here. This option is recommended for production use, but it requires to *uninstall* existing Mercurial packages provided by the operating system.

Option `-n` provides an alternative installation name. The default name `vcs` means “version control system”. The name appears in the URL for SSH access, and thus has some relevance to end-users. The initial server URL

also uses the same suffix, but that can (and should) be changed later via regular Apache configuration.

Option `-o` augments the environment of Isabelle system options: relevant options for Isabelle/Phabricator have the prefix "`phabricator_`" (see also the result of e.g. "`isabelle options -l`").

Option `-r` specifies an alternative installation root directory: it needs to be accessible for the Apache web server.

Option `-R` specifies an alternative directory for repositories that are hosted by Phabricator. Provided that it is accessible for the Apache web server, the directory can be reused for the `hgweb` view by Mercurial.[6]

6.4.3 `isabelle phabricator_setup_mail`

The `isabelle phabricator_setup_mail` tool provides mail configuration for an existing Phabricator installation:

```
Usage: isabelle phabricator_setup_mail [OPTIONS]

  Options are:
    -T USER      send test mail to Phabricator user
    -f FILE      config file (default: "mailers.json" within
                 Phabricator root)
    -n NAME      Phabricator installation name (default: "vcs")

  Provide mail configuration for existing Phabricator installation.
```

Proper mail configuration is vital for Phabricator, but the details can be tricky. A common approach is to re-use an existing SMTP mail service, as is often included in regular web hosting packages. It is sufficient to create one mail account for multiple Phabricator installations, but the configuration needs to be set for each installation.

The first invocation of `isabelle phabricator_setup_mail` without options creates a JSON template file. Its `key` entry should be changed to something sensible to identify the configuration, e.g. the Internet Domain Name of the mail address. The `options` specify the SMTP server address and account information.

Another invocation of `isabelle phabricator_setup_mail` with updated JSON file will change the underlying Phabricator installation. This can be done repeatedly, until everything works as expected.

[6] See also the documentation https://www.mercurial-scm.org/wiki/PublishingRepositories and the example https://isabelle.sketis.net/repos.

Option `-T` invokes a standard Phabricator test procedure for the mail configuration. The argument needs to be a valid Phabricator user: the mail address is derived from the user profile.

Option `-f` refers to an existing JSON configuration file, e.g. from a previous successful Phabricator installation: sharing mailers setup with the same mail address is fine for outgoing mails; incoming mails are optional and not configured here.

6.4.4 `isabelle phabricator_setup_ssh`

The `isabelle phabricator_setup_ssh` tool configures a special SSH service for all Phabricator installations:

```
Usage: isabelle phabricator_setup_ssh [OPTIONS]

  Options are:
    -p PORT      sshd port for Phabricator servers (default: 2222)
    -q PORT      sshd port for the operating system (default: 22)

  Configure ssh service for all Phabricator installations: a separate sshd
  is run in addition to the one of the operating system, and ports need to
  be distinct.

  A particular Phabricator installation is addressed by using its
  name as the ssh user; the actual Phabricator user is determined via
  stored ssh keys.
```

This is optional, but very useful. It allows to refer to hosted repositories via ssh with the usual public-key authentication. It also allows to communicate with a Phabricator server via the JSON API of *Conduit*[7].

The Phabricator SSH server distinguishes installations by their name, e.g. `vcs` as SSH user name. The public key that is used for authentication identifies the user within Phabricator: there is a web interface to provide that as part of the user profile.

The operating system already has an SSH server (by default on port 22) that remains important for remote administration of the machine.

Options `-p` and `-q` allow to change the port assignment for both servers. A common scheme is `-p 22 -q 222` to leave the standard port to Phabricator, to simplify the ssh URL that users will see for remote repository clones.[8]

[7] https://secure.phabricator.com/book/phabricator/article/conduit

[8] For the rare case of hosting Subversion repositories, port 22 is de-facto required. Otherwise Phabricator presents malformed `svn+ssh` URLs with port specification.

Redirecting the operating system sshd to port 222 requires some care: it requires to adjust the remote login procedure, e.g. in `$HOME/.ssh/config` to add a `Port` specification for the server machine.

Chapter 7

Miscellaneous tools

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

7.1 Building Isabelle docker images

Docker[1] provides a self-contained environment for complex applications called *container*, although it does not fully contain the program in a strict sense of the word. This includes basic operating system services (usually based on Linux), shared libraries and other required packages. Thus Docker is a light-weight alternative to regular virtual machines, or a heavy-weight alternative to conventional self-contained applications.

Although Isabelle can be easily run on a variety of OS environments without extra containment, Docker images may occasionally be useful when a standardized Linux environment is required, even on Windows[2] and macOS[3]. Further uses are in common cloud computing environments, where applications need to be submitted as Docker images in the first place.

The `isabelle build_docker` tool builds docker images from a standard Isabelle application archive for Linux:

```
Usage: isabelle build_docker [OPTIONS] APP_ARCHIVE

  Options are:
    -B NAME      base image (default "ubuntu")
    -E           set Isabelle/bin/isabelle as entrypoint
    -P NAME      additional Ubuntu package collection ("X11", "latex")
    -l NAME      default logic (default ISABELLE_LOGIC="HOL")
    -n           no docker build
    -o FILE      output generated Dockerfile
    -p NAME      additional Ubuntu package
```

[1] https://docs.docker.com
[2] https://docs.docker.com/docker-for-windows
[3] https://docs.docker.com/docker-for-mac

```
  -t TAG        docker build tag
  -v            verbose

Build Isabelle docker image with default logic image, using a standard
Isabelle application archive for Linux (local file or remote URL).
```

Option `-E` sets `bin/isabelle` of the contained Isabelle distribution as the standard entry point of the Docker image. Thus `docker run` will imitate the `isabelle` command-line tool (§1.2) of a regular local installation, but it lacks proper GUI support: `isabelle jedit` will not work without further provisions. Note that the default entrypoint may be changed later via `docker run --entrypoint="..."`.

Option `-t` specifies the Docker image tag: this a symbolic name within the local Docker name space, but also relevant for Docker Hub[4].

Option `-l` specifies the default logic image of the Isabelle distribution contained in the Docker environment: it will be produced by `isabelle build -b` as usual (§2.3) and stored within the image.

Option `-B` specifies the Docker image taken as starting point for the Isabelle installation: it needs to be a suitable version of Ubuntu Linux. The default `ubuntu` refers to the latest LTS version provided by Canonical as the official Ubuntu vendor[5]. For Isabelle2021-1 this should be Ubuntu 20.04 LTS.

Option `-p` includes additional Ubuntu packages, using the terminology of `apt-get install` within the underlying Linux distribution.

Option `-P` refers to high-level package collections: `X11` or `latex` as provided by `isabelle build_docker` (assuming Ubuntu 20.04 LTS). This imposes extra weight on the resulting Docker images. Note that `X11` will only provide remote X11 support according to the modest GUI quality standards of the late 1990-ies.

Option `-n` suppresses the actual `docker build` process. Option `-o` outputs the generated `Dockerfile`. Both options together produce a Dockerfile only, which might be useful for informative purposes or other tools.

Option `-v` disables quiet-mode of the underlying `docker build` process.

Examples

Produce a Dockerfile (without image) from a remote Isabelle distribution:

[4]https://hub.docker.com

[5]https://hub.docker.com/_/ubuntu

```
isabelle build_docker -E -n -o Dockerfile
  https://isabelle.in.tum.de/website-Isabelle2021-1/dist/Isabelle2021-1_linux.tar.gz
```

Build a standard Isabelle Docker image from a local Isabelle distribution, with `bin/isabelle` as executable entry point:

```
isabelle build_docker -E -t test/isabelle:Isabelle2021-1 Isabelle2021-1_linux.tar.gz
```

Invoke the raw Isabelle/ML process within that image:

```
docker run test/isabelle:Isabelle2021-1 process -e "Session.welcome ()"
```

Invoke a Linux command-line tool within the contained Isabelle system environment:

```
docker run test/isabelle:Isabelle2021-1 env uname -a
```

The latter should always report a Linux operating system, even when running on Windows or macOS.

7.2 Managing Isabelle components

The `isabelle components` tool manages 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
    -u DIR       update $ISABELLE_HOME_USER/components: add directory
    -x DIR       update $ISABELLE_HOME_USER/components: remove directory

  Resolve Isabelle components via download and installation: given COMPONENTS
  are identified via base name. Further operations manage etc/settings and
  etc/components in $ISABELLE_HOME_USER.

  ISABELLE_COMPONENT_REPOSITORY="..."
  ISABELLE_HOME_USER="..."
```

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. An existing file that does not contain a suitable line "`init_components...components/main`" needs to be edited according to the printed explanation.

Options `-u` and `-x` operate on user components listed in `$ISABELLE_HOME_USER/etc/components`: this avoid manual editing if Isabelle configuration files.

7.3 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.

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

7.4 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 environment (§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
```

7.5 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:
    -a           display complete environment
    -b           print values only (doesn't work for -a)
    -d FILE      dump complete environment to file (NUL 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 NUL character, i.e. the string terminator in C. Thus the Isabelle/Scala operation `isabelle.Isabelle_System.init` can import the settings environment robustly, and provide its own `isabelle.Isabelle_System.getenv` function.

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
```

7.6 Mercurial repository setup

The `isabelle hg_setup` tool simplifies the setup of Mercurial repositories, with hosting via Phabricator (chapter 6) or SSH file server access.

```
Usage: isabelle hg_setup [OPTIONS] REMOTE LOCAL_DIR

  Options are:
    -n NAME      remote repository name (default: base name of LOCAL_DIR)
    -p PATH      Mercurial path name (default: "default")
    -r           assume that remote repository already exists

  Setup a remote vs. local Mercurial repository: REMOTE either refers to a
  Phabricator server "user@host" or SSH file server "ssh://user@host/path".
```

The `REMOTE` repository specification *excludes* the actual repository name: that is given by the base name of `LOCAL_DIR`, or via option `-n`.

By default, both sides of the repository are created on demand by default. In contrast, option `-r` assumes that the remote repository already exists: it avoids accidental creation of a persistent repository with unintended name.

The local `.hg/hgrc` file is changed to refer to the remote repository, usually via the symbolic path name "`default`"; option `-p` allows to provided a different name.

Examples

Setup the current directory as a repository with Phabricator server hosting:

```
isabelle hg_setup vcs@vcs.example.org .
```

Setup the current directory as a repository with plain SSH server hosting:

```
isabelle hg_setup ssh://files.example.org/data/repositories .
```

Both variants require SSH access to the target server, via public key without password.

7.7 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!

7.8 Creating instances of the Isabelle logo

The `isabelle logo` tool creates variants of the Isabelle logo, for inclusion in PDFLaTeX documents.

```
Usage: isabelle logo [OPTIONS] [NAME]

  Options are:
    -o FILE      alternative output file
    -q           quiet mode

  Create variant NAME of the Isabelle logo as "isabelle_name.pdf".
```

Option `-o` provides an alternative output file, instead of the default in the current directory: `isabelle_`*name*`.pdf` with the lower-case version of the given name.

Option `-q` omits printing of the resulting output file name.

Implementors of Isabelle tools and applications are encouraged to make derived Isabelle logos for their own projects using this template. The license is the same as for the regular Isabelle distribution (BSD).

7.9 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)
    -t           symbolic tags (derived from Mercurial id)

  Display Isabelle version information.
```

The default is to output the full version string of the Isabelle distribution, e.g. "`Isabelle2021-1: December 2021`.

Option `-i` produces a short identification derived from the Mercurial id of the `ISABELLE_HOME` directory; option `-t` prints version tags (if available).

These options require either a repository clone or a repository archive (e.g. download of https://isabelle.sketis.net/repos/isabelle/archive/tip.tar.gz).

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), **18**
build (tool), 17, **20**, 32, 34
build_docker (tool), **81**

client (tool), **37**
components (tool), **83**
condition (system option), **19**
console (tool), 4, **9**

doc (tool), **84**
document (system option), **18**
document (tool), 32, **34**
document_bibliography (system option), **19**
document_build (system option), **35**
document_comment_latex (system option), **18**
document_echo (system option), **18**
document_heading_prefix (system option), **19**
document_output (system option), **18**
document_tags (system option), **18**
document_variants (system option), **18**
dump (tool), **26**

env (tool), **85**
export (tool), 17, **25**

getenv (tool), **85**

hg_setup (tool), **86**

install (tool), **87**
isabelle (executable), 1, 6
ISABELLE_APPLE_PLATFORM64 (setting), **3**
ISABELLE_BIBTEX (setting), **4**, 35
ISABELLE_BROWSER_INFO (setting), **4**, 22, 33
ISABELLE_BROWSER_INFO_SYSTEM (setting), **4**, 22
ISABELLE_BUILD_OPTIONS (setting), 22
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), **3**
isabelle_java (executable), 9
ISABELLE_JAVA_PLATFORM (setting), **4**
ISABELLE_JAVAC_OPTIONS (setting), 61
ISABELLE_JDK_HOME (setting), **4**
ISABELLE_LINE_EDITOR (setting), **4**
ISABELLE_LOGIC (setting), **4**
ISABELLE_LUALATEX (setting), **4**, 35
ISABELLE_MAKEINDEX (setting), **4**, 35
ISABELLE_PDFLATEX (setting),

4, 35
ISABELLE_PLATFORM64 (setting), **3**
ISABELLE_PLATFORM_FAMILY (setting), **3**
ISABELLE_SCALAC_OPTIONS (setting), 61
ISABELLE_TMP_PREFIX (setting), **5**
ISABELLE_TOOL (setting), **2**
ISABELLE_TOOL_JAVA_OPTIONS (setting), **5**
ISABELLE_TOOLS (setting), **4, 5**
ISABELLE_WINDOWS_PLATFORM32 (setting), **3**
ISABELLE_WINDOWS_PLATFORM64 (setting), **3**

java (tool), **59**

log (tool), **24**
logo (tool), **87**

mkroot (tool), 32, **33**
ML_HOME (setting), **4**
ML_IDENTIFIER (setting), **4**
ML_OPTIONS (setting), **4**
ML_PLATFORM (setting), **4**
ML_SYSTEM (setting), **4**

options (tool), **20**

PDF_VIEWER (setting), **5**
phabricator (tool), **76**
phabricator_setup (tool), **77**
phabricator_setup_mail (tool), **78**
phabricator_setup_ssh (tool), **79**
process (tool), **7**
profiling (system option), **19**

rlwrap (executable), **9**

scala (antiquotation), **66**
scala (ML antiquotation), **64**
scala (tool), **59**
scala_build (tool), **62**
scala_function (ML antiquotation), **64**
scala_method (antiquotation), **66**
scala_object (antiquotation), **66**
scala_project (tool), **63**
scala_type (antiquotation), **66**
scalac (tool), **60**
server (tool), **36**
session_chapter (syntax), **13**
session_entry (syntax), **13**
sessions (tool), **30**
system_heaps (system option), **19**
system_log (system option), **19**

threads (system option), **19**, 23
timeout (system option), **19**
timeout_scale (system option), **19**

update (tool), **28**
USER_HOME (setting), **2**

version (tool), **88**