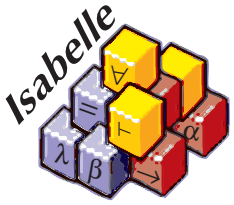


The Isabelle System Manual

Makarius Wenzel

8 October 2017

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 . . . 8
- 1.4 The raw Isabelle Java process . . . 9
- 1.5 YXML versus XML . . . 10

2 Isabelle sessions and build management **11**
- 2.1 Session ROOT specifications . . . 11
- 2.2 System build options . . . 16
- 2.3 Invoking the build process . . . 18
- 2.4 Maintain theory imports wrt. session structure . . . 21

3 Presenting theories **23**
- 3.1 Generating HTML browser information . . . 23
- 3.2 Preparing session root directories . . . 24
- 3.3 Preparing Isabelle session documents . . . 25
- 3.4 Running LaTeX within the Isabelle environment . . . 27

4 Isabelle/Scala development tools **29**
- 4.1 Java Runtime Environment within Isabelle . . . 29
- 4.2 Scala toplevel . . . 29
- 4.3 Scala compiler . . . 30
- 4.4 Scala script wrapper . . . 30

5 Miscellaneous tools **31**
5.1 Resolving Isabelle components 31
5.2 Displaying documents 32
5.3 Viewing documentation 32
5.4 Shell commands within the settings environment 32
5.5 Inspecting the settings environment 33
5.6 Installing standalone Isabelle executables 34
5.7 Creating instances of the Isabelle logo 34
5.8 Output the version identifier of the Isabelle distribution 35

Bibliography **36**

Index **37**

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/IsabelleXXXX`.

   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:

- `ISABELLE_TOOL` is set automatically to the absolute path name of the `isabelle` executables.
- `ISABELLE_OUTPUT` will have the identifiers of the Isabelle distribution (cf. `ISABELLE_IDENTIFIER`) and the ML system (cf. `ML_IDENTIFIER`) appended automatically to its value.

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]

`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_PLATFORM`, `ISABELLE_PLATFORM32`, `ISABELLE_PLATFORM64`.

`ISABELLE_PLATFORM`* is automatically set to a symbolic identifier for the underlying hardware and operating system. The Isabelle platform identification always refers to the 32 bit variant, even this is a 64 bit machine. Note that the ML or Java runtime may have a different idea, depending on which binaries are actually run.

`ISABELLE_PLATFORM64`* is similar to `ISABELLE_PLATFORM` but refers to the proper 64 bit variant on a platform that supports this; the value is empty for 32 bit. Note that the following bash expression (including the quotes) prefers the 64 bit platform, if that is available:

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

`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. "`Isabelle2017`".

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

`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` needs to point to a full JDK (Java Development Kit) installation with `javac` and `jar` executables. This is essential for Isabelle/Scala and other JVM-based tools to work properly. Note that conventional `JAVA_HOME` usually points to the JRE (Java Runtime Environment), not JDK.

`ISABELLE_PATH` is a list of directories (separated by colons) where Isabelle logic images may reside. When looking up heaps files, the value of `ML_IDENTIFIER` is appended to each component internally.

`ISABELLE_OUTPUT`* is a directory where output heap files should be stored by default. The ML system and Isabelle version identifier is appended here, too.

`ISABELLE_BROWSER_INFO` is the directory where theory browser information is stored as HTML and PDF (see also §3.1). The default value is `$ISABELLE_HOME_USER/browser_info`.

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

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

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

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

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

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

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

1.2 The Isabelle tool wrapper

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

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

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

Available tools:
  ...
```

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

1. An external tool found on the directories listed in the `ISABELLE_TOOLS` settings variable (colon-separated list in standard POSIX notation).

   (a) If a file "*tool*`.scala`" is found, the source needs to define some object that extends the class `Isabelle_Tool.Body`. The Scala compiler is invoked on the spot (which may take some time), and the body function is run with the command-line arguments as `List[String]`.

(b) If an executable file "*tool*" is found, it is invoked as stand-alone program with the command-line arguments provided as `argv` array.

2. An internal tool that is registered in `Isabelle_Tool.internal_tools` within the Isabelle/Scala namespace of `Pure.jar`. This is the preferred entry-point for high-end tools implemented in Isabelle/Scala — compiled once when the Isabelle distribution is built. These tools are provided by Isabelle/Pure and cannot be augmented in user-space.

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

Examples

Show the list of available documentation of the Isabelle distribution:

```
isabelle doc
```

View a certain document as follows:

```
isabelle doc system
```

Query the Isabelle settings environment:

```
isabelle getenv ISABELLE_HOME_USER
```

1.3 The raw Isabelle ML process

1.3.1 Batch mode

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

```
Usage: isabelle process [OPTIONS]

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

  Run the raw Isabelle ML process in batch mode.
```

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

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

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

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

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

Example

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.

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
    -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
    -s           system build mode for session image
```

```
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 `-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).

Option `-s` has the same meaning as for `isabelle build` (§2.3).

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 §4.1).

Example

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

```
isabelle_java isabelle.Main
```

1.5 YXML versus XML

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

1. The encoding is always UTF-8.
2. Body text is represented verbatim (no escaping, no special treatment of white space, no named entities, no CDATA chunks, no comments).
3. Markup elements are represented via ASCII control characters $\mathbf{X} = 5$ 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.

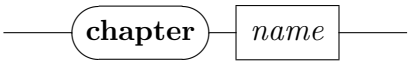
2.1 Session ROOT specifications

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.

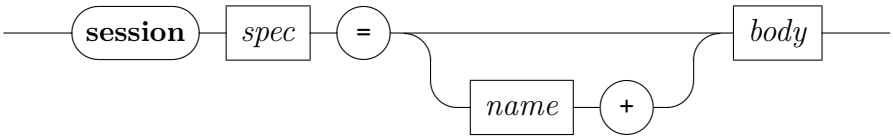
The ROOT file format follows the lexical conventions of the *outer syntax* of Isabelle/Isar, see also [2]. This defines common forms like identifiers, names, quoted strings, verbatim text, nested comments etc. The grammar for *session_chapter* and *session_entry* is given as syntax diagram below; each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

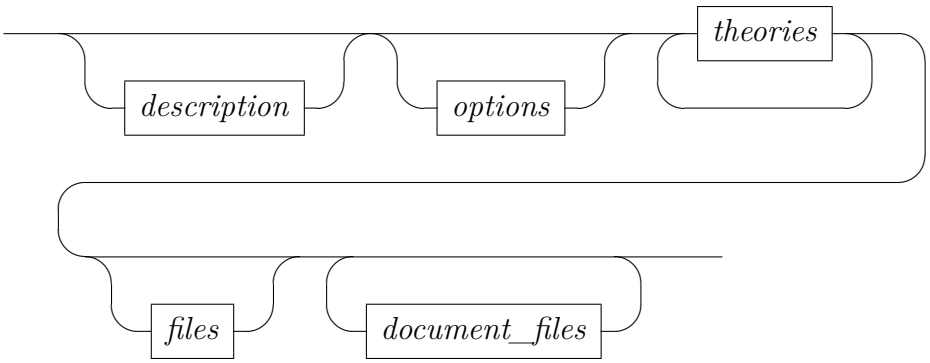
session_chapter



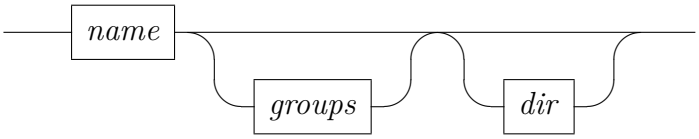
session_entry



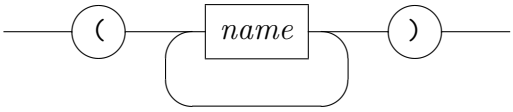
body



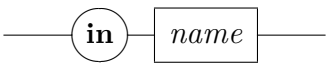
spec



groups

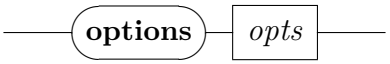


dir

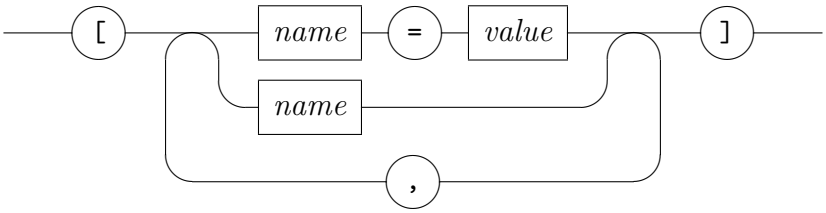


description

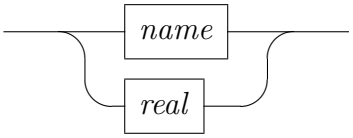
options



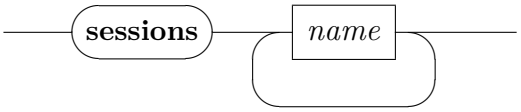
opts



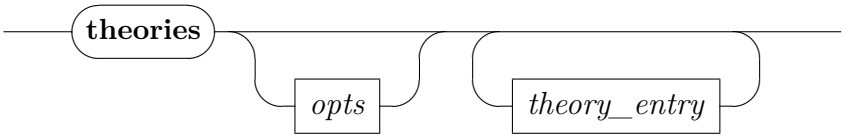
value



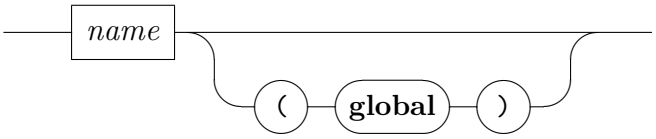
sessions



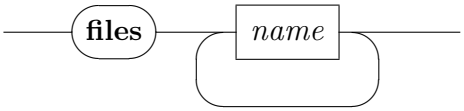
theories



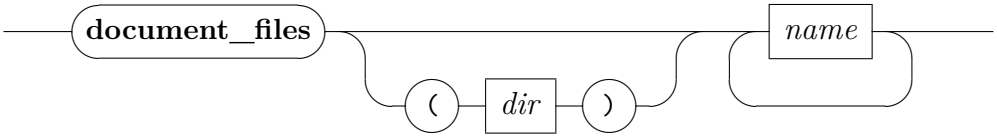
theory_entry



files



document_files



session $A = B +$ *body* defines a new session A based on parent session B, with its content given in *body* (imported sessions, theories and auxiliary source files). 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 theories and auxiliary source files are located relatively to the session directory. The prover process is run within the same as its current working directory.

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

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

sessions *names* specifies sessions that are *imported* into the current name space of theories. This allows to refer to a theory A from session B by

the qualified name $B.A$ — although it is loaded again into the current ML process, which is in contrast to a theory that is already present in the *parent* session.

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

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

A theory name that is followed by (**global**) is treated literally in other session specifications or theory imports. In contrast, the default is to qualify theory names by the session name, in order to ensure globally unique names in big session graphs.

files *files* lists additional source files that are involved in the processing of this session. This should cover anything outside the formal content of the theory sources. In contrast, files that are loaded formally within a theory, e.g. via **ML_file**, need not be declared again.

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*.

document_files *files* abbreviates **document_files** (**in** *document*) *files*, i.e. document sources are taken from the base directory `document` within the session root directory.

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 (http://afp.sourceforge.net), for example.

2.2 System build options

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

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

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

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

  Document variant names are just a matter of conventions. It is also possible to use different document variant names (without tags) for different document root entries, see also §3.3.
- `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`).
- `checkpoint` helps to fine-tune the global heap space management. This is relevant for big sessions that may exhaust the small 32-bit address space of the ML process (which is used by default). When the option is enabled for some **theories** block, a full sharing stage of immutable values in memory happens *before* loading the specified theories.

- `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.

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

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

  Options are:
    -b           include $ISABELLE_BUILD_OPTIONS
    -g OPTION    get value of OPTION
    -l           list options
    -x FILE      export to FILE in YXML format

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

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

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

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

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

2.3 Invoking the build process

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

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

  Options are:
    -D DIR       include session directory and select its sessions
    -N           cyclic shuffling of NUMA CPU nodes (performance tuning)
    -R           operate on requirements of selected sessions
    -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
    -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)
    -s           system build mode: produce output in ISABELLE_HOME
    -v           verbose
    -x NAME      exclude session NAME and all descendants

  Build and manage Isabelle sessions, depending on implicit settings:

  ISABELLE_BUILD_OPTIONS="..."

  ML_PLATFORM="..."
  ML_HOME="..."
  ML_SYSTEM="..."
  ML_OPTIONS="..."
```

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

Any session root directory may refer recursively to further directories of the same kind, by listing them in a catalog file `ROOTS` line-by-line. This helps to

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

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 `-x` *NAME* specify sessions to be excluded. All descendents of excluded sessions are removed from the selection as specified above. 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.

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

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

Option `-c` cleans all descendants of the selected sessions before performing the specified build operation.

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 `-s` enables *system mode*, which means that resulting heap images and log files are stored in `$ISABELLE_HOME/heaps` instead of the default location `ISABELLE_OUTPUT` (which is normally in `ISABELLE_HOME_USER`, i.e. the user's home directory).

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
```

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

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

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

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

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

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

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

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

Clean all sessions without building anything:

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

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

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

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

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

2.4 Maintain theory imports wrt. session structure

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

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

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

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

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

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

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

Option `-I` determines potential session imports, which may be turned into **sessions** within the corresponding `ROOT` file entry. Thus theory imports from other sessions may use session-qualified names. For example, adhoc **imports** `"~~/src/HOL/Library/Multiset"` may become formal **imports** `"HOL-Library.Multiset"` after adding **sessions** `"HOL-Library"` to the `ROOT` entry.

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

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

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

Examples

Determine potential session imports for some project directory:

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

Mercurial repository check for some project directory:

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

Incremental update of theory imports for some project directory:

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

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

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

Figure 3.1: The tool chain of Isabelle session presentation

3.1 Generating HTML browser information

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

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

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

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

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

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

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

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

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

3.2 Preparing session root directories

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

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

  Options are:
    -d           enable document preparation
    -n NAME      alternative session name (default: DIR base name)

  Prepare session root DIR (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.

Option `-d` indicates that the session shall be accompanied by a formal document, with *DIR*`/document/root.tex` as its LaTeX entry point (see also chapter 3).

Option `-n` allows to specify 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, and `ISABELLE_DOCUMENT_FORMAT` the document format to be filled filled into the generated `ROOT` file.

Examples

Produce session `Test` (with document preparation) within a separate directory of the same name:

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

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

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

3.3 Preparing Isabelle session documents

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

```
Usage: isabelle document [OPTIONS] [DIR]

  Options are:
    -c           cleanup -- be aggressive in removing old stuff
    -n NAME      specify document name (default 'document')
    -o FORMAT    specify output format: pdf (default), dvi
    -t TAGS      specify tagged region markup

  Prepare the theory session document in DIR (default 'document')
  producing the specified output format.
```

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

The `-c` option tells `isabelle document` to dispose the document sources after successful operation! This is the right thing to do for sources generated by an Isabelle process, but take care of your files in manual document preparation!

The `-n` and `-o` option specify the final output file name and format, the default is "`document.dvi`". Note that the result will appear in the parent of the target `DIR`.

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

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

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

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

When running the session, Isabelle copies the content of the original `document` directory into its proper place within `ISABELLE_BROWSER_INFO`, according to the session path and document variant. Then, for any processed theory *A* some LaTeX source is generated and put there as *A*`.tex`. Furthermore, a list of all generated theory files is put into `session.tex`. Typically,

the root LaTeX file provided by the user would include `session.tex` to get a document containing all the theories.

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

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

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

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

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

3.4 Running LaTeX within the Isabelle environment

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

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

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

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

Appropriate LaTeX-related programs are run on the input file, according to the given output format: `latex`, `pdflatex`, `dvips`, `bibtex` (for `bbl`), and `makeindex` (for `idx`). The actual commands are determined from the settings environment (`ISABELLE_PDFLATEX` etc.).

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

The `syms` output is for internal use; it generates lists of symbols that are available without loading additional LaTeX packages.

Examples

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

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

Chapter 4

Isabelle/Scala development tools

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

4.1 Java Runtime Environment within Isabelle

The `isabelle java` tool is a direct wrapper for the Java Runtime Environment, within the regular Isabelle settings environment (§1.1). The command line arguments are that of the underlying Java version. It is run in `-server` mode if possible, to improve performance (at the cost of extra startup time).

The `java` executable is the one within `ISABELLE_JDK_HOME`, according to the standard directory layout for official JDK distributions. The class loader is augmented such that the name space of `Isabelle/Pure.jar` is available, which is the main Isabelle/Scala module.

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

```
isabelle java isabelle.GUI_Setup
```

4.2 Scala toplevel

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

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

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

4.3 Scala compiler

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

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

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

4.4 Scala script wrapper

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

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

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

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

Chapter 5

Miscellaneous tools

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

5.1 Resolving Isabelle components

The `isabelle components` tool resolves Isabelle components:

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

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

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

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

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

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

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

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

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

5.2 Displaying documents

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

```
Usage: isabelle display DOCUMENT

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

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

5.3 Viewing documentation

The `isabelle doc` tool displays Isabelle documentation:

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

  View Isabelle documentation.
```

If called without arguments, it lists all available documents. Each line starts with an identifier, followed by a short description. Any of these identifiers may be specified as arguments, in order to display the corresponding document (see also §5.2).

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

5.4 Shell commands within the settings environment

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

ment (§1.1).

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

```
isabelle env bash
```

5.5 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
                 (null terminated entries)

  Get value of VARNAMES from the Isabelle settings.
```

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

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

Examples

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

```
isabelle getenv ISABELLE_HOME_USER
```

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

```
isabelle getenv -b ISABELLE_OUTPUT
```

5.6 Installing standalone Isabelle executables

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

```
Usage: isabelle install [OPTIONS] BINDIR

  Options are:
    -d DISTDIR   refer to DISTDIR as Isabelle distribution
                 (default ISABELLE_HOME)

  Install Isabelle executables with absolute references to the
  distribution directory.
```

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

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

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

5.7 Creating instances of the Isabelle logo

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

```
Usage: isabelle logo [OPTIONS] XYZ

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

  Options are:
    -n NAME      alternative output base name (default "isabelle_xyx")
    -q           quiet mode
```

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

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

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

5.8 Output the version identifier of the Isabelle distribution

The `isabelle version` tool displays Isabelle version information:

```
Usage: isabelle version [OPTIONS]

  Options are:
    -i           short identification (derived from Mercurial id)

  Display Isabelle version information.
```

The default is to output the full version string of the Isabelle distribution, e.g. "`Isabelle2017: October 2017`.

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

Bibliography

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

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

[3] M. Wenzel. *Isabelle/jEdit*. http://isabelle.in.tum.de/doc/jedit.pdf.

Index

bash (executable), **2**, 2
browser_info (system option), **16**
build (tool), **18**, 23

checkpoint (system option), **16**
components (tool), **31**
condition (system option), **17**
console (tool), 4, **8**

display (tool), **32**
doc (tool), **32**
document (system option), **16**
document (tool), 23, **25**
document_output (system option), **16**
document_variants (system option), **16**
DVI_VIEWER (setting), **5**

env (tool), **32**

getenv (tool), **33**

imports (tool), **21**
install (tool), **34**
isabelle (executable), 1, 6
ISABELLE_BIBTEX (setting), **4**
ISABELLE_BROWSER_INFO (setting), **4**, 24
ISABELLE_BUILD_OPTIONS (setting), 19
ISABELLE_DOCS (setting), **4**
ISABELLE_HOME (setting), **1**, **3**
ISABELLE_HOME_USER (setting), **3**
ISABELLE_IDENTIFIER (setting), **3**
isabelle_java (executable), 9
ISABELLE_JDK_HOME (setting), **4**
ISABELLE_LATEX (setting), **4**
ISABELLE_LINE_EDITOR (setting), **4**
ISABELLE_LOGIC (setting), **4**
ISABELLE_OUTPUT (setting), 2, **4**
ISABELLE_PATH (setting), **4**
ISABELLE_PDFLATEX (setting), **4**
ISABELLE_PLATFORM (setting), **3**
ISABELLE_PLATFORM64 (setting), **3**
ISABELLE_PLATFORM_FAMILY (setting), **3**
ISABELLE_TMP_PREFIX (setting), **5**
ISABELLE_TOOL (setting), **2**
ISABELLE_TOOLS (setting), **4, 5**

java (tool), **29**

latex (tool), 23, **27**
logo (tool), **34**

mkroot (tool), 23, **24**
ML_HOME (setting), **4**
ML_IDENTIFIER (setting), **4**
ML_OPTIONS (setting), **4**
ML_PLATFORM (setting), **4**
ML_SYSTEM (setting), **4**

options (tool), **17**

PDF_VIEWER (setting), **5**
process (tool), **7**, 23
profiling (system option), **17**

rlwrap (executable), **9**

scala (tool), **29**
scalac (tool), **30**
session_chapter (syntax), **12**
session_entry (syntax), **12**

threads (system option), **16**, 19
timeout (system option), **17**
timeout_scale (system option), **17**

USER_HOME (setting), **3**

version (tool), **35**