The Isabelle System Manual

Markus Wenzel[1]
TU München

28 August 1998

[1]Section 4.1 was written by Carsten Clasohm. Section 4.3 was written by Stefan Berghofer.

Contents

Chapter 1

Basic concepts

The *Isabelle System Manual* describes Isabelle together with related tools and user interfaces as seen from an outside (operating system oriented) view. See the *Isabelle Reference Manual* for all internal ML level user commands, on the other hand.

The Isabelle system level environment is based on a few general concepts:

- The *Isabelle settings mechanism*, which provides environment variables to all Isabelle programs (including tools and user interfaces).
- *Isabelle proper* (`isabelle`), which runs logic sessions, both interactively or in batch mode. In particular, `isabelle` abstracts over the invocation of the actual ML system to be used.
- The *Isabelle tools wrapper* (`isatool`), which provides a generic startup platform for Isabelle related utilities. Thus tools automatically benefit from the settings mechanism. Furthermore, the shell's search path is kept clean from many small programs.
- The *Isabelle interface wrapper* (`Isabelle`[1]), which provides some abstraction over the actual user interface to be used.

The beginning user would probably just run one of the interfaces (by invoking the capital `Isabelle`), and maybe some basic tools like `doc` (see §2.1). This assumes that the system has already been installed properly, of course.[2]

1.1 Isabelle settings

The Isabelle system heavily depends on the *settings mechanism*. Basically, this is just a collection of environment variables, e.g. `ISABELLE_HOME`, `ML_SYSTEM`, `ML_HOME`. These variables are *not* intended to be set directly from the shell!

Isabelle employs a somewhat more sophisticated scheme of *settings files* — one for site-wide defaults, another for optional user-specific modifications. With

[1]Note the capital `I`!

[2]In case you have to do this yourself, see the `INSTALL` file in the top-level directory of the distribution. The information provided there should be sufficient as a start.

all configuration variables in just a few places, this is much more maintainable and user-friendly than ordinary shell environment variables.

In particular, we avoid the typical situation where prospective users of a software package are told to put this and that in their shell startup scripts, before being able to actually run it. Isabelle requires none such administrative chores of its end-users — the executables can be invoked straight away. Usually, users would just want to put the Isabelle `bin` directory into their shell's search path, of course.

Building the environment

Whenever any of the Isabelle executables is run, their settings environment is built as follows:

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 have to be run from their original location in the distribution directory — copying or linking them somewhere else just won't work!

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

   This file typically contains a rather long list of shell variable assigments, 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 have to be adapted (most likely `ML_SYSTEM` etc.).

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

   Thus individual users may override the site-wide defaults. See also file `etc/user-settings.sample` in the distribution. Typically, a user settings file would contain only a few lines, just the assigments that are really needed. One should definitely *not* start with a full copy the central `$ISABELLE_HOME/etc/settings`. This could cause very annoying maintainance problems later, when the Isabelle installation is updated or changed otherwise.

Note that settings files are actually full GNU bash scripts. So one may use complex shell commands, say `if` or `case` statements to set variables depending on the system architecture or other environment variables, for example. 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` and `ISATOOL` are set automatically to the absolute path names of the `isabelle` and `isatool` executables, respectively.
- `ISABELLE_PATH` and `ISABELLE_OUTPUT` will have the ML system identifier (as specified by `ML_SYSTEM`) automatically appended to their values.

The Isabelle settings scheme is basically quite simple, but non-trivial. For debugging purposes, the resulting environment may be inspected with the `getenv` utility, see §2.4.

Common variables

Below 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 *.

`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 try to set `ISABELLE_HOME` yourself from the shell.

`ISABELLE_HOME_USER` is the user-specific counterpart of `ISABELLE_HOME`. The default value is `~/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 `etc/settings`.

`ISABELLE*`, `ISATOOL*` are automatically set to the full paths of the `isabelle` and `isatool` executables, respectively. Thus other tools and scripts need not assume that the Isabelle `bin` directory is on the current search path of the shell.

`ML_SYSTEM`, `ML_HOME`, `ML_OPTIONS` specify the underlying ML system to be used for Isabelle. The choice of `ML_SYSTEM` identifiers is quite fixed, see the global `etc/settings` file for some examples. The actual compiler binary will be run from directory `ML_HOME`, with `ML_OPTIONS` as first arguments on the command line.

`ISABELLE_PATH*` is a list of directories (separated by colons) where Isabelle logic images may reside. Note that the `ML_SYSTEM` identifier is appended to each component automatically.

ISABELLE_OUTPUT* is a directory where output heap files should be stored by default. The ML_SYSTEM identifier is appended here, too.

ISABELLE_BROWSER_INFO is the directory where theory browser information (HTML and graph data) is stored (see also §4.1). 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 — e.g. when running isabelle directly, or some user interface.

ISABELLE_USEDIR_OPTIONS is implicitly prefixed to the command line of any isatool usedir invocation (see also §2.8). This typically contains compilation options for object-logics — usedir is the basic utility that builds them (cf. the IsaMakefiles in the distribution).

ISABELLE_TOOLS is a colon separated list of directories that are scanned by isatool for utility programs (see also §1.3).

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

DVI_VIEWER specifies the command to be used for displaying dvi files.

ISABELLE_INSTALL_FONTS determines the way that the Isabelle symbol fonts are installed into your currently running X11 display server. X11 fonts are a non-trivial issue, see §3.1 for more information.

ISABELLE_TMP_PREFIX is the prefix from which any isabelle session derives an individual directory for temporary files. The default is somewhere in /tmp.

ISABELLE_INTERFACE is an identifier that specifies the actual user interface that the capital Isabelle should invoke. Currently available are none, xterm and emacs. See §1.4 for more details.

1.2 Isabelle proper — isabelle

The isabelle executable runs logic sessions — either interactively or in batch mode. It provides an abstraction over the underlying ML system, and over the actual heap file locations. Its usage is:

```
Usage: isabelle [OPTIONS] [INPUT] [OUTPUT]

  Options are:
    -e MLTEXT    pass MLTEXT to the ML session
    -m MODE      add print mode for output
    -q           non-interactive session
    -r           open heap file read-only
    -u           pass 'use"ROOT.ML";' to the ML session
    -w           reset write permissions on OUTPUT

  INPUT (default "$ISABELLE_LOGIC") and OUTPUT specify in/out heaps.
  These are either names to be searched in the Isabelle path, or actual
  file names (containing at least one /).
  If INPUT is "RAW_ML_SYSTEM", just start the bare bones ML system.
```

Input files without path specifications are looked up in the `ISABELLE_PATH` setting, which may consist of multiple components separated by colons — these are tried in order. Likewise, base names are relative to the directory specified by `ISABELLE_OUTPUT`. In any case, actual file locations may also be given by including at least one `/` in the name (hint: use `./` to refer to the current directory).

Options

If the input heap file does not have write permission bits set, or the `-r` option is given explicitely, then the session will be read-only. That is, the ML world cannot be committed back into the logic image. Otherwise, a writable session enables commits into either the input file, or into an alternative output heap file (in case this is given as the second argument on the command line).

The read-write state of sessions is determined at startup only, it cannot be changed intermediately. Also note that heap images may require considerable amounts of disk space. Users are responsible themselves to dispose their heap files when they are no longer needed.

The `-w` option makes the output heap file read-only after terminating. Thus subsequent invocations cause the logic image to be read-only automatically.

Using the `-e` option, arbitrary ML code may be passed to the Isabelle session from the command line. Multiple `-e`'s are evaluated in order. Strange things may happen when errorneous ML code is given. Also make sure that the ML commands are terminated properly by semicolon.

The `-u` option is a shortcut for `-e`, passing "`use"ROOT.ML";`" to the ML session.

The `-m` option adds identifiers of print modes to be made active for this session. Typically, this is used by some user interface, for example to enable output of mathematical symbols from a special screen font.

Isabelle normally enters an interactice ML top-level loop (after processing the `-e` texts). The `-q` option inhibits this, thus providing a pure batch mode facility.

Examples

Run an interactive session of the default object-logic (as specified by the `ISABELLE_LOGIC` setting) like this:

```
isabelle
```

Usually `ISABELLE_LOGIC` refers to one of the standard logic images, which are read-only by default. A writable session — based on `FOL`, but output to `Foo` (in the directory specified by the `ISABELLE_OUTPUT` setting) — may be invoked as follows:

```
isabelle FOL Foo
```

Ending this session normally (e.g. by typing control-D) dumps the whole ML system state into `Foo`. Be prepared for several megabytes!

The `Foo` session may be continued later (still in writable state) by:

```
isabelle Foo
```

A read-only `Foo` session may be started by:

```
isabelle -r Foo
```

The next example demonstrates batch execution of Isabelle. We print a certain theorem of `FOL`:

```
isabelle -e "prth allE;" -q -r FOL
```

Note that the output text will be usually interspersed with some garbage produced by the ML compiler.

1.3 The Isabelle tools wrapper — `isatool`

All Isabelle related utilities are called via a common wrapper — `isatool`:

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

  Start Isabelle utility program TOOL with ARGS. Pass "-?" to TOOL
  for more specific help.

  Available tools are:

    browser - Isabelle theory graph browser
    doc - view Isabelle documentation
    ...
```

Basically, Isabelle tools are ordinary executable scripts. These are run within the same settings environment as Isabelle proper, see §1.1. The set of available tools is collected by isatool from the directories listed in the `ISABELLE_TOOLS` setting. Do not try to call the scripts directly. Neither should you add the tool directories to your shell's search path.

See chapter 2 for descriptions of most utilities that come with the Isabelle distributions. Third-parties may add their own, of course.

1.4 The Isabelle interface wrapper — `Isabelle`

Isabelle is a generic theorem prover, even w.r.t. its user interface. The `Isabelle` command (note the capital `I`) provides a uniform way for end-users to invoke a certain interface. Which one to actually start is determined by the `ISABELLE_INTERFACE` setting variable. Currently, the following are recognized:

`none` is just a pass-through to `isabelle`. Thus `Isabelle` basically becomes an alias for `isabelle`.

`xterm` refers to a simple xterm based interface which is part of the Isabelle distribution.

`emacs` refers to David Aspinall's *Isamode* for emacs. Isabelle just provides a wrapper for this, the actual Isamode distribution is available elsewhere.

The factory default for `ISABELLE_INTERFACE` is `xterm`. This interface runs `isabelle` within its own xterm window. Usually, display of mathematical symbols from the Isabelle font is also enabled (see §3.1 for font configuration issues). Furthermore, different kinds of identifiers in logical terms are highlighted appropriately, e.g. free variables in bold and bound variables underlined.

There are some more options available. Just pass `-?` to the `xterm` interface to have its usage printed.

Chapter 2

Miscellaneous tools

Subsequently we describe various Isabelle related utilities — in alphabetical order.

2.1 Viewing documentation — `isatool doc`

The `doc` utility displays online documentation:

```
Usage: isatool doc [DOC]

  View Isabelle documentation DOC, or show list of available documents.
```

If called without arguments, it lists all available documents. Each line starts with an identifier, followed by some comment. Any of these identifiers may be specified as the first argument in order to have the corresponding document displayed.

The `ISABELLE_DOCS` setting specifies the list of directories (separated by colons) to be scanned for documentations. The program for viewing `dvi` files is determined by the `DVI_VIEWER` setting.

2.2 Tuning proof scripts — `isatool expandshort`

The `expandshort` utility tunes ML proof scripts to enhance readability:

```
Usage: expandshort [FILES|DIRS...]

  Recursively find .ML files, expand shorthand goal commands.
  Also contracts uses of resolve_tac, dresolve_tac, eresolve_tac,
  rewrite_goals_tac on 1-element lists; furthermore expands tabs,
  since they are now forbidden in ML string constants.

  Renames old versions of files by appending "~~".
```

In the files or directories supplied as arguments, all occurrences of the shorthand commands `br`, `be` etc. in proof scripts are replaced with the corresponding full commands. The old versions of the files are renamed to have the suffix `~~`.

2.3 Getting logic images — `isatool findlogics`

The `findlogics` utility traverses all directories specified in `ISABELLE_PATH`, looking for Isabelle logic images. Its usage is:

```
Usage: isatool findlogics

  Collect heap file names from ISABELLE_PATH.
```

The base names of all files found on the path are printed — sorted and with duplicates removed. Also note that `ISABELLE_PATH` implicitly depends upon `ML_SYSTEM`. Thus switching to another ML compiler may change the set of logic images available.

2.4 Inspecting the settings environment – `isatool getenv`

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

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

  Options are:
    -a          display complete environment
    -b          print values only (doesn't work for -a)

  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.

Examples

Get the ML system identifier and the location where the compiler binaries are supposed to reside as follows:

```
isatool getenv ML_SYSTEM ML_HOME
  ML_SYSTEM=smlnj-110
  ML_HOME=/usr/local/smlnj-110/bin
```

The next one peeks at the search path that `isabelle` uses to locate logic images:

```
isatool getenv -b ISABELLE_PATH
  /home/me/isabelle/heaps/smlnj-110:/usr/local/isabelle/heaps/smlnj-110
```

Here we have used the `-b` option to suppress the `ISABELLE_PATH=` prefix. The value above is what became of the following assignment in the default settings file:

```
ISABELLE_PATH=$ISABELLE_HOME_USER/heaps:$ISABELLE_HOME/heaps
```

Note how the `ML_SYSTEM` value got appended automatically to each path component. This is a special feature of `ISABELLE_PATH` (and also of `ISABELLE_OUTPUT`).

2.5 Installing standalone Isabelle binaries – `isatool install`

Usually, the Isabelle binaries (`isabelle`, `isatool` etc.) are just run from their location within the distribution directory, probably indirectly by the shell through its `PATH`. In some cases, though, another more traditional installation scheme might be preferably where executables are put into some global system directory (like `/usr/local/bin`).

Doing a plain copy of the Isabelle executables just would not work, though. One should use the `install` utility instead:

```
Usage: install BINDIR

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

  Install standalone Isabelle binaries in directory BINDIR with absolute
  references to DISTDIR/bin, which becomes non-relocatable this way.
```

This installs standalone executables in `BINDIR` containing absolute references to `DISTDIR` (the default is `ISABELLE_HOME` as figured out by the current `isatool` invocation). While the generated scripts themselves may be relocated afterwards, they cease working if the referenced Isabelle distribution is moved somewhere else. Here is an example use of `install`:

```
isatool install /usr/local/bin
```

2.6 Isabelle's version of make — `isatool make`

The Isabelle `make` utility is a very simple wrapper for ordinary Unix `make`:

```
Usage: isatool make [ARGS ...]

  Compiles logic in current directory using IsaMakefile.
  ARGS are directly passed to the system make program.
```

Note that the Isabelle settings environment is also active. Thus one may refer to its values within the `IsaMakefile`, e.g. `$(ISABELLE_OUTPUT)`. Furthermore, programs started from the make file also inherit this environment. Typically, `IsaMakefile`s defer the real work to the `usedir` utility, see §2.8.

The basic `IsaMakefile` convention is that the default target builds the actual logic, including its parents if appropriate. The `images` target is intended to build all local logic images, while the `test` target shall build all related examples. The `all` target shall do `images` and `test`.

Examples

Refer to the `IsaMakefile`s of the Isabelle distribution's object-logics as a model for your own developements. For example, see `src/FOL/IsaMakefile`.

2.7 Make all logics – `isatool makeall`

The `makeall` utility applies Isabelle make to all logic directories of the distribution:

```
Usage: makeall [ARGS ...]

  Apply isatool make to all logics (passing ARGS).
```

The arguments `ARGS` are just passed verbatim to each `make` invocation.

2.8 Running complete logics — `isatool usedir`

The `usedir` utility builds object-logic images, or runs example sessions based on existing logics. Its usage is:

```
Usage: usedir LOGIC NAME

  Options are:
    -b           build mode (output heap image, use dir ".")
    -i BOOL      generate theory browsing information,
                 i.e. HTML / graph data (default false)
    -s NAME      override session NAME

  Build object-logic or run examples. Also creates browsing
  information (HTML etc.) according to settings.
```

Note that the value of the `ISABELLE_USEDIR_OPTIONS` setting is implicitly prefixed to *any* `usedir` call. Since the `IsaMakefile`s of all object-logics distributed with Isabelle just invoke `usedir` for the real work, one may control compilation options globally via above variable. In particular, generation of HTML browsing information is enabled or disabled this way.

Options

Basically, there are two different modes of operation: *build mode* (enabled through the `-b` option) and *example mode* (default).

Calling `usedir` with `-b` runs `isabelle` with input image `LOGIC` and output to `NAME`, as provided on the command line. This will be a batch session, running `ROOT.ML` from the current directory and then quitting. It is assumed that `ROOT.ML` contains all ML commands required to build the logic.

In example mode on the other hand, `usedir` runs a read-only session of `LOGIC` (typically just built before) and automatically runs `ROOT.ML` from within directory `NAME`. It assumes that file `ROOT.ML` in directory `NAME` contains appropriate ML commands to run the desired examples.

The `-i` option controls theory browsing data generation. It may be explicitely turned on or off — the last occurrence of `-i` on the command line wins.

Any `usedir` session is named by some *session identifier*. These accumulate, documenting the way sessions depend on others. For example, consider `Pure/FOL/ex`, which refers to the examples of `FOL` which is built upon `Pure`.

The current session's identifier is by default just the base name of the `LOGIC` argument (in build mode), or of the `NAME` argument (in example mode). This may be overridden explicitely via the `-s` option.

Examples

Refer to the `IsaMakefile`s of the Isabelle distribution's object-logics as a model for your own developements. For example, see `src/FOL/IsaMakefile`.

Chapter 3

Fonts and character encodings

With the advent of print modes in Isabelle, variant forms of output have become very easy. As the canonical application of this feature, `Pure` and major object-logics (`FOL`, `ZF`, `HOL`, `HOLCF`) support optional input and output of nice mathematical symbols as built-in option.

Symbolic output is enabled by activating the `symbols` print mode. User interfaces (e.g. `isa-xterm`, see §1.4) usually do this already by default.

Displaying non-standard characters requires special screen fonts, of course. The `installfonts` utility takes care of this (see §3.1). Furthermore, some ML systems disallow non-ASCII characters in literal string constants. This problem is avoided by appropriate input filtering (see §3.3).

These things usually happen behind the scenes. Users normally do not have to read the explanations below, unless something really fails to work.

3.1 Telling X11 about the Isabelle fonts — `isatool installfonts`

The `installfonts` utility ensures that your currently running X11 display server (as determined by the `DISPLAY` environment variable) knows about the Isabelle fonts. Its usage is:

```
Usage: isatool installfonts

  Install the isabelle fonts into your X11 server.
  (May be safely called repeatedly.)
```

Note that this need not be called manually under normal circumstances — user interfaces depending on the Isabelle fonts usually invoke `installfonts` automatically.

As simple as this might appear to be, it is not! X11 fonts are a surprisingly complicated matter. Depending on your network structure, fonts might have to be installed differently. This has to be specified via the `ISABELLE_INSTALLFONTS` variable in your local settings.

In the simplest situation, X11 is used only locally, i.e. the client program (Isabelle) and the display server are run on the same machine. In this case, most X11 display servers should be happy by being told about the Isabelle fonts directory as follows:

```
ISABELLE_INSTALLFONTS="xset fp+ $ISABELLE_HOME/lib/fonts; xset fp rehash"
```

The same also works for remote X11 sessions in a somewhat homogeneous network, where any X11 display machine also mounts the Isabelle distribution under the *same* name as the client side.

Above method fails, though, if the display machine does have the font files at the same location as the client. In case your server is a full workstation with its own file system, you could in principle just copy the fonts there and do an appropriate `xset fp+` manually before running the Isabelle interface. This is very awkward, of course. It is even impossible for proper X11 terminals that do not have their own file system.

A much better solution is to have a *font server* offer the Isabelle fonts to any X11 display on the network. There are already suitable servers running at Munich and Cambridge. So in case you have a sensible Internet connection to either site, you may just attach yourself as follows:

```
ISABELLE_INSTALLFONTS="xset fp+ tcp/isafonts.informatik.tu-muenchen.de:7200"
```

or

```
ISABELLE_INSTALLFONTS="xset fp+ tcp/font-serv.cl.cam.ac.uk:7100"
```

In the unfortunate case that neither local fonts work, nor accessing our worldwide font service is practical, it might be best to start your own in-house font service. This is in principle easy to setup. The program is called `xfs` (sometimes just `fs`), see the `man` pages of your system. There is an example fontserver configuration available in the `lib/fontserver` directory of the Isabelle distribution.

3.2 Check for non-ASCII characters — `isatool nonascii`

The `nonascii` utility checks files for non-ASCII characters:

```
Usage: nonascii [FILES|DIRS...]

Recursively find .thy/.ML files and check for non-ASCII
characters.
```

Under normal circumstances, non-ASCII characters do not appear in theories and proof scripts.

3.3 Filtering non-ASCII characters — `isatool symbolinput`

Processing non-ASCII text is notoriously difficult. In particular, some ML systems reject character codes outside the range 32–127 as part of literal string constants. In order to circumvent such restrictions, Isabelle employs a general notation where glyphs are referred by some symbolic name instead of their actual encoding: The general form is `\<`*charname*`>`.

The `symbolinput` utility converts a input stream encoded according to the standard Isabelle font layout into pure ASCII text. There is no usage — `symbolinput` just works from standard input to standard output, without any options available.

For example, the non-ASCII input string `"A ∧ B ∨ C"` will be replaced by `"A \\<and> B \\<or> C"`. Note that the `\` are escaped, accomodating concrete ML string syntax.

In many cases, it might be wise not to rely on symbolic characters and avoid non-ASCII text in files altogether (see also §3.2). Then symbolic syntax would be really optional, with always suitable ASCII representations available: In theory definitions, symbols appear only in mixfix annotations — using the `\<`*charname*`>` form, proof scripts are just left in plain ASCII. Thus users with ASCII-only facilities will still be able to read your files.

Chapter 4

Presenting theories

4.1 Generating theory browsing information

Isabelle is able to generate theory browsing information, such as HTML documents that show a theory's definition, the theorems proved in its ML file and the relationship with its ancestors and descendants. HTML is the hypertext markup language used on the World Wide Web to represent text containing links to other documents. These documents may be viewed using Web browsers like Netscape or Lynx.

Besides the three HTML files that are made for every theory (definition and theorems, ancestors, descendants), Isabelle stores links to all theories in an index file. These indexes are themself linked with other indexes to represent the hierarchic structure of Isabelle's logics.

In addition to the HTML files, Isabelle also generates *graph* files that represent the theory hierarchy of a logic. These graphs can be comfortably displayed by a graph browser Java applet embedded in the generated HTML pages. There is also a stand-alone version of the graph browser which allows browsing theory graphs without having to start a Web browser first. This version also includes features such as generating PostScript files, which are not available in the applet version. The graph browser will be described later in this chapter.

To generate theory browsing information for logics that are part of the Isabelle distribution, simply append "`-i true`" to the `ISABELLE_USEDIR_OPTIONS` setting before making a logic. For example, to generate browsing information for `FOL`, first add something like this to your Isabelle settings file:

```
ISABELLE_USEDIR_OPTIONS="-i true"
```

Then `cd` into the `src/FOL` directory of the Isabelle distribution and do `isatool make` (or even `isatool make all`).

The directory in which to store theory browsing information is determined by the `ISABELLE_BROWSER_INFO` setting.

As some of Isabelle's logics are based on others (e.g. `ZF` on `FOL`) and because the HTML files list a theory's ancestors, you should not make HTML files for a logic if the HTML files for the base logic do not exist. Otherwise some of the hypertext links might point to non-existing documents.

The entry point to all logics is the `index.html` file located in the directory denoted by `ISABELLE_BROWSER_INFO`.

A complete HTML version of all distributed Isabelle object-logics and examples may be accessed on the WWW at:

```
http://www4.informatik.tu-muenchen.de/~isabelle/library/
```

Of course, this is not necessarily consistent with your local version!

To present your own theories on the WWW, simply copy the whole `ISABELLE_BROWSER_INFO` directory to your WWW server.

4.2 Extending or adding a logic

If you add a new subdirectory to Isabelle's logics (or add a completely new logic), provide a `ROOT.ML` file which reads in the theory files. The `ROOT.ML` file will then be processed via the function

```
use_dir : string -> unit
```

which takes a path as its parameter, changes the working directory, executes `ROOT.ML`, and makes sure that theory browsing information is generated properly. The `index.html` file written in this directory will be automatically linked to the first index found in the (recursively searched) super directories.

The `usedir` utility (see also §2.8) will automatically take care of this.

The generated HTML files contain all theorems that were proved in the theory's ML file with `qed`, `qed_goal` and `qed_goalw`, or stored with `bind_thm` and `store_thm`. Additionally, there is a hypertext link to the whole ML file.

You can add section headings to the list of theorems by using

```
section: string -> unit
```

in a theory's ML file, which converts a plain string to a HTML heading and inserts it before the next theorem proved or stored with one of the above functions.

4.3 Browsing theory graphs

The graph browser is a tool for visualizing dependency graphs of Isabelle theories. Certain nodes of the graph (i.e. theories) can be grouped together in "directories", whose contents may be hidden, thus enabling the user to filter out irrelevant information. The browser is written in Java, it can be used both as a stand-alone application and as an applet.

Invoking the graph browser

The stand-alone version of the graph browser is wrapped up as an Isabelle tool called `browser`:

```
Usage: browser [GRAPHFILE]
```

When no filename is specified, the browser automatically changes to the directory `ISABELLE_BROWSER_INFO/graph/data`.

The applet version of the browser can be invoked by opening the `index.html` file in the directory `ISABELLE_BROWSER_INFO` from your Web browser and selecting "version for Java capable browsers". There is also a link to the corresponding theory graph in every logic's `index.html` file.

Using the graph browser

The browser's main window, which is shown in figure 4.1, consists of two subwindows: In the left subwindow, the directory tree is displayed. The graph itself is displayed in the right subwindow.

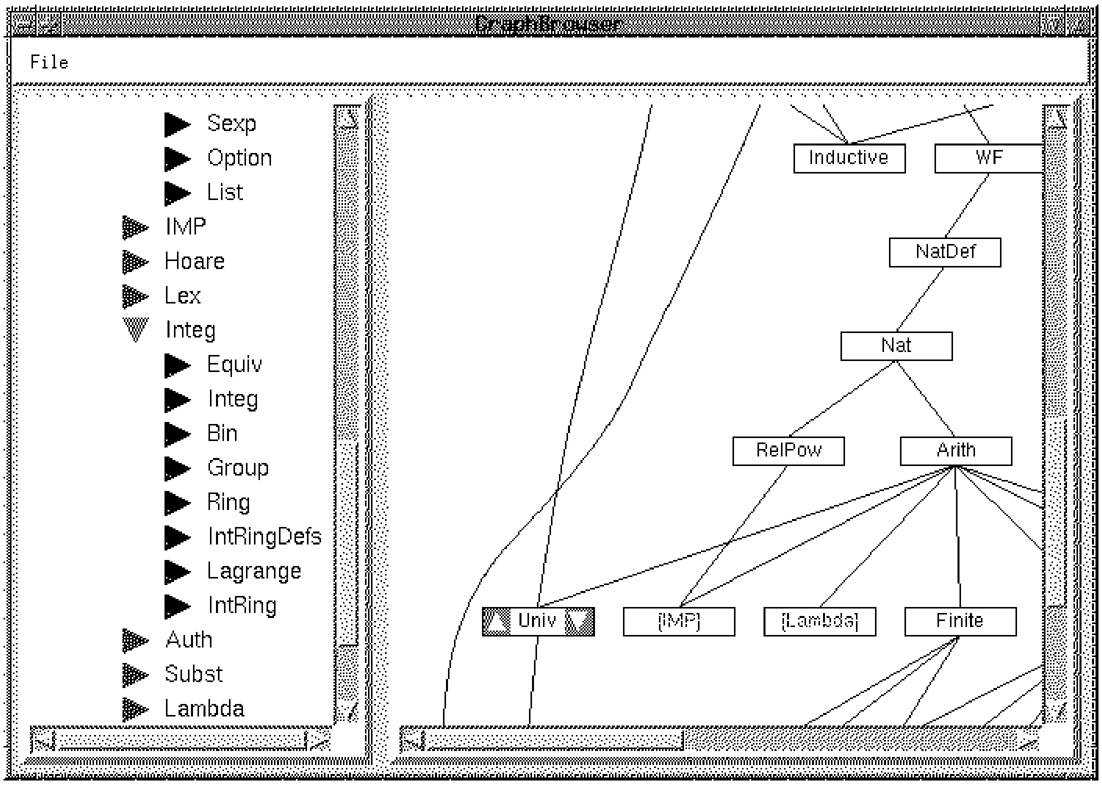

Figure 4.1: Browser main window

The directory tree window

This section describes the usage of the directory browser and the meaning of the different items in the browser window.

- A red arrow before a directory name indicates that the directory is currently "folded", i.e. the nodes in this directory are collapsed to one single node. In the right subwindow, the names of nodes corresponding to folded directories are enclosed in square brackets and displayed in red colour.
- A green downward arrow before a directory name indicates that the directory is currently "unfolded". It can be folded by clicking on the directory name. Clicking on the name for a second time unfolds the directory again. Alternatively, a directory can also be unfolded by clicking on the corresponding node in the right subwindow.
- Blue arrows stand before ordinary node (i.e. theory) names. When clicking on such a name, the graph display window focuses to the corresponding node. Double clicking invokes a text viewer window in which the contents of the theory file are displayed.

The graph display window

When pointing on an ordinary node, an upward and a downward arrow is shown. Initially, both of these arrows are green. Clicking on the upward or downward arrow collapses all predecessor or successor nodes, respectively. The arrow's colour then changes to red, indicating that the predecessor or successor nodes are currently collapsed. The node corresponding to the collapsed nodes has the name "`[....]`". To uncollapse the nodes again, simply click on the red arrow or on the node with the name "`[....]`". Similar to the directory browser, the contents of theory files can be displayed by double clicking on the corresponding node.

The "File" menu

Please note that due to Java security restrictions this menu is not available in the applet version. The meaning of the menu items is as follows:

Open... Open a new graph file.

Export to PostScript Outputs the current graph in PostScript format, appropriately scaled to fit on one single sheet of paper. The resulting file can printed directly.

Export to EPS Outputs the current graph in Encapsulated PostScript format. The resulting file can be included in other documents.

Quit Quit the graph browser.

*Syntax of graph definition files

A graph definition file has the following syntax:

$$\begin{aligned} \mathit{graph} &= \{\ \mathit{vertex}\ ;\ \}^+ \\ \mathit{vertex} &= \mathit{vertexname}\ \mathit{vertexID}\ \mathit{dirname}\ [\texttt{+}]\ \mathit{path}\ [\texttt{<}|\texttt{>}]\ \{\ \mathit{vertexID}\ \}^* \end{aligned}$$

The meaning of the items in a vertex description is as follows:

vertexname The name of the vertex.

vertexID The vertex identifier. Note that there may be two vertices with equal names, whereas identifiers must be unique.

dirname The name of the "directory" the vertex should be placed in. A "+" sign after *dirname* indicates that the nodes in the directory are initially visible. Directories are initially invisible by default.

path The path of the corresponding theory file. This is specified relatively to the path of the graph definition file.

List of successor/predecessor nodes A "<" sign before the list means that successor nodes are listed, a ">" sign means that predecessor nodes are listed. If neither "<" nor ">" is found, the browser assumes that successor nodes are listed.

Index