Division of Informatics, University of Edinburgh

Laboratory for Foundations of Computer Science

Isamode — Theorem Proving with Isabelle inside Emacs

by

David Aspinall

Informatics Research Report EDI-INF-RR-0020

Division of Informatics **May 2000**
http://www.informatics.ed.ac.uk/

Isamode — Theorem Proving with Isabelle inside Emacs

David Aspinall

Informatics Research Report EDI-INF-RR-0020

DIVISION *of* INFORMATICS
Laboratory for Foundations of Computer Science

May 2000

Abstract :

This paper documents Isamode, a user-interface and suite of editing functions for using the theorem prover Isabelle inside Emacs.

Keywords : theorem proving, Isabelle, Emacs

Isamode

Theorem proving with Isabelle inside GNU Emacs
Isamode version 2.7, for Isabelle99.
May 2000.

David Aspinall

This manual documents Isamode, Version 2.7, for use with Isabelle99.

Version stamp: `$Id: Isamode.texi,v 3.8 2000/05/18 10:46:53 da Exp $`

1 Using Isabelle with Emacs

There are several good reasons for using Emacs when you're proving theorems with Isabelle. For example, you can retain a complete history of your interaction inside an Emacs buffer, which you can browse or search through. You can use the editing and history mechanism to speed up typing commands to Isabelle. You can conveniently cut and paste in the shell buffer, to edit your completed proofs into an ML batch file, or issue commands kept in a file.

These things are possible just by running Isabelle inside an Emacs shell buffer. *Isamode* is an Emacs Lisp package which builds on this to provide additional Isabelle-specific editing and interaction facilities. It goes a small way to softening the harshness of Isabelle's text-only user interface; the hope is that it will help make the learning curve for new Isabelle users less steep, whilst at the same time speeding up use for experienced Isabelle users.

Features of Isamode

- An **Isabelle interaction mode** allows you to interact with Isabelle in a shell-like buffer.
- **Menus and keyboard accelerators** issue common tactics and proof commands.
- The **current proof state** is displayed in a separate buffer.
- **Rule tables** show the rules available in the current logic, for building tactics.
- A **theory file mode** provides features for editing '`.thy`' theory files.

The proof state, rule tables and listener are called *associated buffers*, because each is associated with a particular Isabelle interaction buffer. Each of the associated buffers can each appear in a separate Emacs frame on the screen.

Prerequisites

This manual assumes a basic understanding of Isabelle; you should at least have read the *Introduction to Isabelle* manual.

To get the most from Isamode, you should understand a little about how Emacs lisp packages work, in particular, how to set user options. Emacs is self-documenting, so you can begin from `C-h` and find out everything! Here are some useful commands:

`C-h i`	`info`
`C-h m`	`describe-mode`
`C-h b`	`describe-bindings`
`C-h f`	`describe-function`
`C-h v`	`describe-variable`
	`M-x edit-options`

2 Interacting with Isabelle

Using Isabelle with Isamode, you interact in a shell-like buffer based on the Emacs command-interpreter package, `comint`. The same package lies behind the shell mode in Emacs. The `comint` package has many useful features. For example, there is a command history ring, accessed with the keys *M-p* (`comint-previous-input`) and *M-n* (`comint-next-input`). Another feature is context-sensitive completion, used in Isamode for filenames and ML identifiers.

You can find out more about `comint` by reading the documentation in the file '`comint.el`' or, as usual, using *C-h m* (`describe-mode`) to show the keys and commands available. Many of the commands are also available on pull-down menus.

The main command to start Isabelle is *M-x isabelle*.

isabelle Command

Prompt for a logic name, and then create or switch to an Isabelle process for the chosen *logic*. The isabelle session takes place in an *interaction buffer*, named **logic**.

See below for more details about how to select a logic.

M-x isabelle behaves as other similar Emacs commands: if there is already a buffer **logic**, then `isabelle` *logic* will switch to that buffer (and its associated buffers). But it is perfectly possible to have more than one Isabelle session with the same logic — you simply rename the first buffer.

2.1 Choosing which Logic

If you type `M-x isabelle` you are prompted for a logic name; you may hit ⟨SPC⟩ for a list of possibilities. Alternatively, selecting 'Session' from the Isabelle menu will also display a sub-menu of logic names. The list shown will contain all the object logics found when Isamode was started. Object logics are searched for using the Isabelle tool `isatool findlogics`.

If you want to use a newly built logic which isn't in the completion list, typing *C-u M-x isabelle* will run the command `isatool findlogics` again to update the internal record of object logics available. (Note: this doesn't rebuild the menu at present).

The command *M-x isabelle* calls the internal function `isabelle-session`, which may be useful if you wish to write special start-up scripts for Emacs to fire-up Isabelle, or define keys to switch to a particular logic session.

isabelle-session *logic* Internal Function

Create or switch to an Isabelle process in an interaction buffer with the base logic *logic*. The buffer is named **logic**.

2.2 Startup Sequence

isa-session-prelude Variable

If non-nil, an ML string to issue at the start of every Isabelle session.
Handy for setting options inside Isabelle (for example, the load path for logics), or for loading a personal startup file that does such configuration.

The default value is set via the Isabelle settings environment, to be the value of the environment variable ISAMODE_*prelude*. An example setting is:

```
ISAMODE_PRELUDE='use "/home/da/isabelle/startup.ML";'
```

which attempts to read in a file '`startup.ML`' from my Isabelle directory. You might set this variable in the private settings file '`$ISABELLE_HOME_`*user*`/etc/settings`'.

note: the string is sent silently; make sure it is properly terminated and doesn't lead to errors - you won't see the results of executing it!

2.3 Completion

comint-dynamic-complete Command
The primary completion command in the Isabelle interaction buffer, usually bound to ⟨TAB⟩. If the point is inside an ML string, completion will assume you are asking for a filename, otherwise that you are completing some ML identifier.

isa-completion-list Constant
This constant contains the table used for ML identifier completion. Completion is provided for identifiers denoting tactics, tacticals and proof commands. See the file '`isa-rules.el`'.

Completion is not implemented for rule names because typically they are short, and may be displayed in a rule table anyway (see Chapter 4 [Rule tables], page 8).

2.4 Backslashes

Isabelle uses ML strings to represent types and terms of the meta-logic. Terms and types can be very long strings which are easier to read, type and edit if they span several lines. ML strings which span lines must contain the escape sequence '\ \' to indicate to the ML parser that the carriage return is not part of the string and that the string will be continued on the next line.

When you are inputting such strings it is annoying to need to remember to include these backslashes. So Isamode makes the job easy for you by doing it automatically, both in interaction buffers, and inside '`.thy`' files (see Section 6.1 [Theory files], page 14).

If you type ⟨RET⟩ at the end of a line which contains an unclosed string, Isamode will insert a backslash (if you haven't done already) before sending it to the ML process. Another backslash will be inserted on the following line.

If you don't like this behaviour, you can disable it by customizing *isa-auto-backslashes*.

isa-auto-backslashes Variable
If non-nil, Isamode will attempt to automatically add backslashes for long strings. See function *isa-send-input* in '`isa-mode.el`'.

2.5 Menus

Isamode provides a main menu, called '`Isabelle`', which appears by default in Isabelle interaction buffers and theory file buffers. After loading Isamode, it will also appear in ML buffers, if you use `sml-mode` (see Section 6.2 [ML files], page 15). To add the main menu to the menubar of the current buffer in other cases, there is a command, *M-x isa-menus*.

isa-menus Command
Add main '`Isabelle`' menu to current menubar.

The main menu has options for starting Isabelle sessions, displaying documentation, or engaging various editing functions (see Chapter 6 [Theory and ML files], page 14). In addition to the '`Isabelle`' menu, there are four menus provided for interaction mode.

'`Option`' allows you to activate or switch to buffers associated with this one, or adjust various Isabelle parameters. There is a quit option here.

'`Goal`'
'`Prover`'
'`Tactic`' are menus providing short-cuts for typing frequently used Isabelle commands. Each item in these menus inserts text into the Isabelle buffer. Most have key bindings too, so you can save many keystrokes.

The convention for the Isabelle command key sequences is:

`C-c C-t` *key*
: for tactics...

`C-c C-r` *key*
: for rewriting tactics...

`C-c C-p` *key*
: for prover tactics...

`C-c C-s` *key*
: for goal commands... (remember goal 's'tack or 's'tart proof)

The commands are shown in full in the following section

2.6 Interaction mode menu operations

Here's a list of the Isamode interaction buffer commands in the '`Goal`', '`Prover`' and '`Tactic`' menus. We show the text they insert, and the keystrokes bound to the commands by default. In the table, *d* stands for the designated subgoal (see Section 3.1 [Proofstate commands], page 6 — don't forget `C-`⟨up⟩ and `C-`⟨down⟩). *rules*, *thmname*, etc, stand for some data that must be typed to complete the command. If there is no data needed, the text may be sent immediately to Isabelle; otherwise you should type something followed by ⟨RET⟩.

Primary Tactics

Emacs command	Inserted text	Key
`isa-resolve_tac`	`by (resolve_tac [`*rules*`] `*d*`);`	`C-c C-t C-r`
`isa-resolve_tac-prems`	`by (resolve_tac prems `*d*`);`	`C-c C-t C-p`
`isa-assume_tac`	`by (assume_tac `*d*`);`	`C-c C-t C-a`
`isa-eresolve_tac`	`by (eresolve_tac [`*rules*`] `*d*`);`	`C-c C-t C-e`
`isa-dresolve_tac`	`by (dresolve_tac [`*rules*`] `*d*`);`	`C-c C-t C-d`
`isa-forward_tac`	`by (forward_tac [`*rules*`] `*d*`);`	`C-c C-t C-f`
`isa-match_tac`	`by (match_tac [`*rules*`] `*d*`);`	`C-c C-t r`
`isa-match_tac-prems`	`by (match_tac prems `*d*`);`	`C-c C-t p`
`isa-eq_assume_tac`	`by (eq_assume_tac [`*rules*`] `*d*`);`	`C-c C-t a`
`isa-ematch_tac`	`by (ematch_tac [`*rules*`] `*d*`);`	`C-c C-t e`
`isa-dmatch_tac`	`by (dmatch_tac [`*rules*`] `*d*`);`	`C-c C-t d`

Rewriting Tactics

`isa-rewrite_goals_tac`	`by (rewrite_goals_tac [`*rules*`]);`	`C-c C-r g`
`isa-rewrite_tac`	`by (rewrite_tac [`*rules*`]);`	`C-c C-r w`
`isa-fold_goals_tac`	`by (fold_goals_tac [`*rules*`]);`	`C-c C-r f`
`isa-cut_facts_tac`	`by (cut_facts_tac [`*rules*`] `*d*`);`	`C-c C-r c`
`isa-cut_facts_tac-prems`	`by (cut_facts_tac prems `*d*`);`	`C-c C-r p`

Prover Tactics

isa-simp_tac	by (simp_tac *simpset*_ss *d*);	*C-c C-p C-s*
isa-asm_simp_tac	by (asm_simp_tac *simpset*_ss *d*);	*C-c C-p C-a*
isa-asm_full_simp_tac	by (asm_full_simp_tac *simpset*_ss *d*);	*C-c C-p C-f*
isa-fast_tac	by (fast_tac *claset*_cs *d*);	*C-c C-p f*
isa-best_tac	by (best_tac *claset*_cs *d*);	*C-c C-p b*
isa-step_tac	by (step_tac *claset*_cs *d*);	*C-c C-p s*
isa-contr_tac	by (contr_tac *d*);	
isa-mp_tac	by (mp_tac *d*);	
isa-eq_mp_tac	by (eq_mp_tac *d*);	

Goal Commands

isa-undo	undo();	*C-c C-s u*
isa-back	back();	*C-c C-s b*
isa-chop	chop();	*C-c C-s c*
isa-choplev	choplev *lev*;	
isa-goal-thy	val prems = goal thy "*goal*";	*C-c C-s g*
isa-goalw-thy	val prems = goalw thy *rewrites* "*goal*";	*C-c C-s w*
isa-result	val *thmname* = result();	*C-c C-s r*
isa-push-proof	push_proof();	*C-c C-s p*
isa-pop-proof	val prems = pop_proof();	*C-c C-s o*
isa-rotate-proofs	val prems = rotate_proof();	*C-c C-s n*

3 Proof state

The *proof state buffer* maintains a display of the current proof state (goal and subgoal list) during interactive proof. Usually this would appear interspersed with user input on the terminal — one of the major benefits of using Isamode is that it separates this output and keeps the machine dialogue uncluttered.

Internally, this is implemented by setting up a process-filter which watches output from the Isabelle process, waiting for character sequences that look like displays of the proof state, as for example, when you type `pr();`. If a proof state buffer is active, these sequences will be stripped from the output and used to update the buffers contents.

proofstate Command

Start or display a proof state buffer associated with the current Isabelle interaction buffer. At most one proof state buffer can be associated with a given Isabelle interaction buffer. This command gives an error if the current buffer is not an Isabelle buffer.

M-x proofstate is usually bound to *C-c M-p*, which is useful in single frame mode (see Section 7.1 [Display options I], page 20) because it provides a quick way of switching buffers in another window to show the proofstate display.

3.1 Proofstate Commands

In the proof state buffer, the cursor usually appears opposite one of the subgoals in the list (if there are any). This subgoal is called the *designated subgoal*; it can be selected using the cursor keys ⟨up⟩ and ⟨down⟩, or from the interaction buffer, with *C-*⟨up⟩ and *C-*⟨down⟩.

The designated subgoal is the one you wish to prove; it is the subgoal that will be used by default by the tactic insertion commands in the interaction buffer (See Section 2.6 [Interaction menu operations], page 4.) or in a rule table (see Section 4.1 [Ruletable commands], page 8)

Other useful keys in the proofstate buffer are ⟨LEFT⟩ (`proofstate-previous-level`) and ⟨RIGHT⟩ (`proofstate-next-level`), which page back and forward through levels of the proof. To go directly to a specific level, type the level as a prefix argument (e.g., *M-2 M-7* for level 27), and hit ⟨SPC⟩ (`proofstate-refresh`). Without an argument, `proofstate-refresh` returns to the latest level.

If you resize the proofstate window, it is useful to set Isabelle's pretty printer margin appropriately. This is done by pressing ⟨RET⟩ (`proofstate-resize-and-refresh`).

3.2 A note about large proof states

You may discover that very large proof states sometimes fail to parse and end up in the Isabelle buffer, instead of the proof state buffer.

The reason for this is that the Emacs code filters all the output from the Isabelle process so that proof states can be stripped out. This is done by matching on text between ML prompts; text up to the next prompt is put into a hidden output buffer before matching. Unfortunately, this has a bad interaction with `use` and `use_thy`, because `use` may produce a lot of output (perhaps slowly) before the next prompt appears, and it is very strange not to see any output before `use` has finished.

So if the output gets larger than a certain threshold, Isamode assumes that it is from `use` (or something similar), and won't contain a proof state. The variable that controls this threshold is `isa-text-spill-size`, which defaults to `2000`. If you notice large proof states spilling into the Isabelle buffer, one solution is to reduce the goals limit; another is to set this variable higher by, for example:

```
(setq isa-text-spill-size 5000)
```

in your '`.emacs`'. But remember that there will be a bigger delay with `use`.

The present output filtering technique is unsatisfactory and may be changed in the future when improved mechanisms of communication between Isabelle and Emacs are possible.

4 Rule tables

A *rule table buffer* displays a formatted table of ML identifiers, typically including rule names, simplifier sets, classical rule sets and tactics. You can use the table for reference purposes, or actively: there are commands to copy the names into the interaction buffer, to build tactics based on the names, or to display the rule.

Isamode comes supplied with basic rule tables for the standard object logics. You can easily extend these to add your favourite rules and tactics, or create new rule tables for your own theories, by writing '`.rules`' files.

The command to show a rule table is `M-x ruletable`, normally bound to the key *`C-c M-l`* (which is useful in single frame mode).

ruletable Command

Prompt for a ruletable to display. This command gives an error if the current buffer is not an Isabelle buffer.

Rule table files are searched for in directories in the list `isa-ruletable-paths`.

isa-ruletable-paths User Option

This variable is a list of directory names which are used to store rule table '`.rules`' files.

The default value is found using the `isatool` program; either it is set from *ISAMODE_RULETABLE_PATH*, if that variable set, or it is set to be the single directory *ISAMODE_HOME/ruletables*. You should set *ISAMODE_RULETABLE_PATH* to be a colon-separated sequence of paths, as usual for Unix path variables.

Just as for object logics, Isamode keeps an internal record of the rule tables available and their file names. If you type *`C-u M-x ruletable`*, this internal record will be re-built.

There may be any number of rule table buffers associated with a single Isabelle interaction buffer. When running under a window system, the first 3 (by default) will receive their own windows. The modeline at the bottom of rule table buffer displays both the name of the logic it is associated with, and the name of the theory for which it is a table.

4.1 Ruletable Commands

The cursor keys are used to move amongst the rule names, or you can use the mouse to highlight a rule name. Hitting *`SPC`* or pressing the left mouse button, *`button-1`*, will insert the rule name into the interaction buffer, possibly preceded by a comma. This allows you to quickly construct lists of rules.

The middle mouse button, *`button-2`*, can be used to execute a tactic:

`S-button-2` executes `by (rtac` *rule d*`);`

`C-button-2` executes `by (dtac` *rule d*`);`

`M-button-2` executes `by (etac` *rule d*`);`

`button-2` executes a tactic based on the name of the rule.
How the tactic is chosen is described in Section 4.3 [Rule Categories], page 9.

where *d* is the designated subgoal (see Section 3.1 [Proofstate commands], page 6).

Pressing ⟨RET⟩ or the right mouse button, *`button-3`*, displays the rule in a temporary buffer using `prth` — hit ⟨q⟩ immediately afterward to remove it. If the name denotes something other than a rule, it may be displayed correctly if what it denotes can be guessed from its name, See Section 4.3 [Rule Categories], page 9. Otherwise, you may just see a type error!

The key *TAB* toggles the display mode of the table. The display can be in *short form*, where descriptive headings for rule tables are not displayed and lines are concatenated (this is useful for single frame mode), or *long form*, where all headings and grouping is shown. *TAB* also reformats the table, so it is useful if you re-size the Emacs window.

4.2 Creating Rule tables

When you write your own theories and logics, you may want to extend or create rule tables. This is very easy to do. Choose a name for your rule table (typically, the name of the theory or logic it documents), and create a text file '*name*`.rules`'.

The text file should consist of *headings* and *identifiers*. Here's a fragment of '`FOL.rules`':

```
Equality
 refl  sym  trans  subst  ssubst
Propositions
 conjI  conjunct1  conjunct2  conjE
 disjI1  disjI2  disjE
 impI  mp  impE
Quantifiers
 allI  spec  allE  all_dupE
```

Headings begin at the start of a line and should not contain whitespace; nothing should follow a heading. Identifiers should be valid ML identifiers, and be separated by tabs or spaces. A line of identifiers begins with some whitespace.

When the rule table file is loaded by Isamode, it is parsed and stored internally; the headings and position of line breaks are recorded. This allows rule tables to be shown in both long and short form: the short form just breaks the line at headings, and concatenates lines of identifiers between headings. More details of the internal format of rule tables are in Section 4.4 [Rule table internals], page 11.

At present there is no mechanism for automatic generation of rule tables. In the future there should be ways of building tables using the theorem database, although it will still be useful to design and edit tables by hand, because of the ability to add headings and comment text in the tables; presently Isabelle's theorem database is too primitive to provide useful meta-comments about theorems.

4.3 Rule Categories

Rule tables are not limited to containing rule names, they may display any ML identifier. Rules (ML type `thm`), tactics (ML type `tactic` or `int->tactic`), rule sets (`simpset`,`claset`) are all candidates. The more logic-dependent names that are shown in a rule table, the less a user has to remember or refer to documentation for.

We can also take the opportunity to make use of all these logic-dependent things. One use is by building appropriate tactics automatically inside Emacs to apply during proof: typically with something of type `claset` one might use `fast_tac`, or `simp_tac` with a `simpset`, etc. For rules, we'd like to distinguish between introduction rules (to use `resolve_tac`) and elimination rules (to use `eresolve_tac`), for example.

Isamode does some 'automatic tactic building' from the rule table, but it isn't very flexible as yet. Clicking with the middle mouse key on a name in a rule table executes a command which guesses which tactic to apply based on the name. In XEmacs, this default tactic is displayed when you move the pointer over the rule.

Emacs decides what kind of entity something in a rule table is, and then which tactic to apply, by pattern matching on its name. This works most of the time, as many identifiers used in Isabelle follow a regular naming convention.

Naming convention table

Here are the categories of identifier that you might want to include in ruletables, together with the patterns used to distinguish them and the default tactics they trigger.
n stands for a digit `0-9`, *d* for a subgoal number, and *rl* for some characters in a rule name.

Pattern	Category	Default tactic
rl`I` or *rl*`I`*n*	*Introduction rule*	`rtac` *rl d*
rl`E` or *rl*`E`*n*	*Elimination rule*	`etac` *rl d*
rl`D` or *rl*`D`*n*	*Destruction rule*	`dtac` *rl d*
	Arbitrary rule	`rtac` *rl d*
rl`_def`	*Definition*	`rewrite_goals_tac [`*rl*`]`
rl`_rls`	*Rule set*	`resolve_tac` *rl d*
rl`_ss`	*Simplifier rule set*	`asm_full_simp_tac` *rl d*
rl`_cs`	*Classical rule set*	`fast_tac` *rl d*
rl`_tac`	*Indexed Tactic*	*rl d*
	Tactic	*rl*

Naming convention example

For example, `iffD2` will be recognised as a destruction rule, whereas `iffI` will be recognised as an introduction rule. A name such as `trans` that matches none of the patterns given is assumed to be an 'Arbitrary rule' to be used with `resolve_tac`. Often this isn't what you want; the heuristic is maybe correct for about 70% of the identifiers used in the built-in object logics. It's a good idea to follow the convention for your own logics.

See Appendix C [Ideas and Dreams], page 26 for some discussion on future improvements to these features.

4.4 Rule table internals

This final section is for lisp hackers only, who may wish to manipulate rule tables at the lisp level (for example, to automatically generate them from '`.thy`' files).

The ruletable data is held in the variable `isa-theory-rules`. The functions `isa-add-theory-rules` and `isa-add-theory-rulegroup` may be called to extend the ruletable variable.

Rule table format

isa-theory-rules Constant

Table of logics and rule names for rule-table buffers.

A list of lists, one per theory or logic, like this:

```
((theory-name rulegroup rulegroup ...)
                    ...
                    ...                    )
```

where *theory-name* is a string. Each *rulegroup* has the form:

```
(rulegroup-name . [rule rule ...])
```

here *rulegroup-name* is a string that must contain a single word. Each *rule* is either a string of a single word, or `nil`; the latter represents a further (unnamed) subgrouping of the rules in the table, and will cause a line-break in the table.

An alternative form for a theory entry is:

```
(theory-name . rules-filename)
```

Where *rules-filename* is a full filename for a '`.rules`' file which contains the ruletable for *theory-name*.

Rule table extension functions

isa-add-theory-rules *rule-list* Function

Add or overwrite a rule table for a theory held in `isa-theory-rules`.

isa-add-theory-rulegroup *theory-name rulegroup* Function

Add or overwrite a *rulegroup* for a theory in `isa-theory-rules`.

5 Listening (and talking)

A *listener buffer* receives a verbatim copy of every command that is issued to an Isabelle session. It is useful for recording interactive proofs: you can cut and paste lines into an ML file to store proofs. Listener buffers can now also talk: you can type commands directly into them, and send lines to the Isabelle session. This makes it easy to create proof scripts.

The command to create a listener buffer is `M-x listener`. It is usually bound to `C-c M-l`, which is useful in single frame mode, or to clear the listener buffer when you start a fresh proof.

listener Command
: Start a new listener buffer associated with the current Isabelle interaction buffer. This gives an error if the current buffer is not an Isabelle interaction buffer.

In fact, it is possible for any buffer to be treated as a listener, and you can easily turn listening on and off. This makes it easy to write proof scripts directly into '`.ML`' files.

listener-minor-mode Command
: Toggle listening for the current buffer.

5.1 Listener Commands

The listener minor mode provides functions useful for editing proof scripts; you can mix and match typing in the buffer that "listens" with typing in and using menus from the Isabelle process buffer.

You can send command lines from a listener buffer to the associated Isabelle process using `C-c C-l` or `C-c C-n` (which runs `listener-use-line`). This copies text from the start of the line, up to the next semi-colon that isn't enclosed in a string or comment. See Section 6.2 [ML files], page 15 for further details of a similar command in ML buffers.

Another way of sending lines to the Isabelle process is by pasting them into a listener buffer with `C-c C-y` (`listener-yank-and-use`), instead of just `C-y`. This is like yanking the text, going to the beginning of the line, and using `C-c C-n`.

You can send an "undo" command to the Isabelle process with `C-c C-u` or `C-c C-p` (`listener-use-line`). After sending the undo, this attempts to move backwards to the previous interactive command.

The movement functions alone are available with `C-c C-f` (`isa-forward-interactive-line`) and `C-c C-b` (`isa-backward-interactive-line`).

Two other commands useful inside the listener are `isa-batchify` and `isa-unbatchify`, which let you very easily create batch proofs from interactive ones and vise-versa. See Section 6.3 [Batchifying proofs], page 16. This makes dedicated listener buffers most useful for playing back batched forms of interactive proofs. For example, you could copy a batch proof from a file into the listener, unbatchify it, and then watch it replay step by step using `C-c C-n`.

6 Theory and ML files

As well as providing facilities for interacting with Isabelle, Isamode has an in-built mode for editing Isabelle's '`.thy`' files. This mode understands the syntax for ML strings (and will insert and correct backslashes) and allows convenient switching to `sml-mode` in the "ML" section of theory files. It provides templates to help remind you of the syntax for theory files.

isa-thy-mode Command

Switch the current buffer to the major mode '`Theory Mode`'. This command will normally be issued automatically via `auto-mode-alist`.

6.1 Editing theory files

The theory mode understands the sections in theory files and provides several commands to use them. *C-c C-n* (`isa-thy-goto-next-section`) moves to the next section, *C-c C-p* (`isa-thy-goto-previous-section`) moves to the previous section. You can also use ⟨CTRL⟩ with the up and down arrow keys for this.

C-c C-t (`isa-thy-insert-template`) provides a template for the file or current section. This is useful if you can't remember the syntax for theory files; you will be asked for text. The templates do not cover all the possible variations for theory files and should be considered merely as prompts for each section: you are advised to read the Isabelle Reference Manual to understand the full situation.

C-c C-u (`isa-thy-use-buffer`) sends the buffer to an Isabelle process with `use_thy` (remember that this may trigger the Isabelle theory reading mechanism to load other files). This is the same key that is used by `sml-mode`.

The interaction buffer chosen (or created) by *C-c C-u* is determined by parsing the file. For reliable results, you should include a special comment towards the start of the file, which has the form:

```
Logic Image: logic-name
```

Then the use functions will send text to the buffer that would be selected by *M-x isabelle* ⟨RET⟩ *logic-name* ⟨RET⟩.

The ⟨TAB⟩ key indents the current line. Indentation includes the automatic insertion of backslashes in strings that span more than one line. This is handy because axioms are ML strings which are often long, and it is tedious to have to remember to put backslashes in the correct places (see the discussion in Section 2.4 [Backslashes in interaction], page 3).

The key ⟨RET⟩ is by default bound to `newline-and-indent` so that backslashes are inserted automatically when you type ⟨RET⟩ at the end of a line containing an open string. See Section 6.4 [Theory mode User Options], page 17 for details of how to control the backslashing and indentation behaviour.

C-k is bound to `isa-thy-kill-line` which behaves in the same way as the usual `kill-line`, except there is some attempt to interpret continuation backslashes inside strings. For example, *C-k* in the middle of a line which is continued will only delete up to the terminating backslash, not the end of line; *C-k* anywhere after the final non-whitespace character on the line will delete continuation backslashes when joining lines.

If you find the automatic treatment of backslashes distracting and would rather deal with them manually, you can revert to the usual key bindings by putting this code in your '`.emacs`':

```
(add-hook 'isa-thy-mode-hook
   (function (lambda ()
               (define-key isa-thy-mode-map "\C-k"
                           'kill-line)
```

```
(define-key isa-thy-mode-map "\C-m"
            'newline))))
```

If `isa-thy-use-sml-mode` is non-`nil`, *C-c C-c* invokes `sml-mode` as a minor mode in the ML section. This is done automatically by `TAB`.

C-c C-o finds and switches to the associated ML file, that is, the file with the same base name but extension '`.ML`' in place of '`.thy`'.

isa-thy-insert-template Command
Insert an entry for the current section in a '`.thy`' file.

isa-thy-use-buffer Command
Send the buffer to an Isabelle process with `use_thy`. The choice of Isabelle interaction buffer can be influenced with a special '`Logic Image:`' comment.

isa-thy-find-other-file Command
Find the associated ML or theory file. This is the file with the same base name but extension '`.ML`' in place of '`.thy`'.

6.2 Editing ML files

Theory files may have associated ML files, typically containing proofs of some basic consequences of a theory's axioms. Further ML files may be used with Isabelle, containing big proofs, proof procedures, and so on. Isamode is intended to be used in conjunction with the Emacs mode for editing ML files, which is called `sml-mode`.

If you visit a theory file before an ML file, `sml-mode` will be customized to work with Isamode.

You can use *C-c C-o* (`isa-thy-find-other-file`) to visit the theory file corresponding to an ML file — the opposite operation to *C-c C-o* in theory mode.

Uses of *C-c C-u* (`isa-thy-use-buffer`), *C-c C-r* (`isa-thy-use-region`), *C-c C-l* (`isa-thy-use-line`) will send the buffer, region or line to an Isabelle interaction buffer. The first two of these keys normally send text to an ML process buffer. The logic chosen can be directed in the same way as for theory files, i.e., with a `Logic Image:` comment towards the start of the file see Section 6.1 [Theory files], page 14.

`isa-thy-use-line` is intended for copying lines of interactive proofs, to replay them. A line is understood to be the text starting on current line and extending to the next semi-colon (thus it may span more than one buffer line). Semi-colons inside strings and comments are ignored.

isa-thy-use-region Command
Send the region to an Isabelle process with `use`. The choice of Isabelle interaction buffer can be influenced with a special '`Logic Image:`' comment.

isa-thy-use-line Command
Send the current line (delimited by semi-colons or blank lines) to an Isabelle process with `use`. The choice of Isabelle interaction buffer can be influenced with a special '`Logic Image:`' comment.

isa-ml-file-extension User Option
The file name extension to use for ML files; the default is '`.ML`' which is used in the Isabelle distribution.

6.3 Interactive and Batch Proofs

Isabelle has two forms of proof script which may be kept in ML files. These are *interactive proofs*, which begin with `goal`, and *batch proofs*, which begin with `prove_goal`. Interactive proofs update the proof state at each step, when a tactic is applied using '`by`'. Batch proofs involve no state; they execute in a single step without output and can be stored in ML structures. Batch proofs might be considered as a 'final' form for debugged proofs; lemmas in many of the distributed object logics are proved with batch proofs.

Isamode has functions to help you convert between interactive and batch proofs. `isa-batchify` creates a batch proof from an interactive one and `isa-unbatchify` goes the other way. The original versions of these functions were supplied in an emacs file distributed with Isabelle called '`goalify.el`', which is superseded by Isamode.

Here is an example of an interactive proof:

```
val prems = goal FOL.thy "P & Q --> P";
by (resolve_tac [impI] 1);
by (eresolve_tac [conjE] 1);
by (assume_tac 1);
qed "easy_theorem";
```

and here is a corresponding batch proof:

```
qed_goal "easy_theorem"  FOL.thy "P & Q --> P"
 (fn prems =>
        [
        (rtac impI 1),
        (etac conjE 1),
        (assume_tac 1)
        ]);
```

I created the batch proof above by typing in the interactive proof line by line into a `*FOL*` buffer, cutting the text from the `*FOL-listener*` and then using `isa-batchify` on the region. To do this, you must set the mark at the start of the proof with *`C-SPC`*, and move to the end (after `result`), before using *`M-x isa-batchify`*.

Notice that the tactics involving single rules were replaced with short forms. The command *`M-x isa-expandshorts`* can be used to do this without converting the form of a proof.

Using *`M-x isa-unbatchify`* on the above region converts back into an interactive proof:

```
val prems = goal FOL.thy "P & Q --> P";
by (rtac impI 1);
by (etac conjE 1);
by (assume_tac 1);
val easy_theorem = result();
```

If you write batch proofs in ML files, it may be wise to write them in a regular form such as the above, so that it is possible to convert them back into interactive proofs in the future if you need to (for example, to adapt them to different theorems).

isa-batchify Command
: Convert an interactive proof into a batch proof.

isa-unbatchify Command
: Convert a batch proof into an interactive proof.

isa-expandshorts Command
: Normalize tactics and commands, like the shell-script '`expandshorts`' in the Isabelle distribution. Tactic command shorthands (`ba`,`br`,...) are expanded and long-forms with singleton arguments (`resolve_tac [`*rule*`]`) are contracted (to `rtac` *rule*).

I hope future versions of these functions will be easier to use (without needing to set a region to use them) and more reliable — at present they are a bit fussy about the format of the proofs, which should be pretty much as shown above.

6.4 Theory mode User Options

Here are some options you may use to control the layout of theory files and behaviour when editing theory files.

isa-thy-heading-indent Variable
Indentation for section headings.

isa-thy-indent-level Variable
Indentation level for Isabelle theory files.

isa-thy-indent-strings Variable
If non-nil, indent inside strings.
This option is useful because often strings contain logical formulae and it is desirable to indent them according to parenthesis nesting.

You may wish to disable indenting inside strings if your logic uses any of the usual bracket characters in unusual ways.

isa-thy-use-sml-mode User Option
If non-nil, invoke sml-mode inside "ML" section of theory files.
It will be triggered by line indentation that occurs with the ⟨TAB⟩ or ⟨RET⟩.

The default value is `nil`.

6.5 Fontification of Theory Files

Theory files can be automatically highlighted using `font-lock`, which is integrated with XEmacs. If you have font-locking enabled by default (via the menu Options->Syntax Highlighting), theory files should be automatically syntax-highlighted with section heading, string and comment highlighting when you load and edit them.

If you wish, the highlighting can be configured by adjusting the variable `isa-thy-mode-font-lock-keywords`.

isa-thy-mode-font-lock-keywords Variable
Font lock keywords for `isa-thy-mode`.
Default is set automatically from `isa-thy-headings-regexp`. This as a value for `font-lock-keywords`, see the documentation of that variable for more details.

font-lock-keywords Variable
A list of the keywords to highlight.
Each element should be of the form:

matcher
(*matcher* . *match*)
(*matcher* . *facename*)
(*matcher* . *highlight*)
(*matcher highlight* ...)
(eval . *form*)

where *highlight* should be either *match-highlight* or *match-anchored*.

form is an expression, whose value should be a keyword element, evaluated when the keyword is (first) used in a buffer. This feature can be used to provide a keyword that can only be generated when Font Lock mode is actually turned on.

For highlighting single items, typically only *match-highlight* is required. However, if an item or (typically) items is to be highlighted following the instance of another item (the anchor) then *match-anchored* may be required.

match-highlight should be of the form:

(*match facename override laxmatch*)

Where *matcher* can be either the regexp to search for, a variable containing the regexp to search for, or the function to call to make the search (called with one argument, the limit of the search). *match* is the subexpression of *matcher* to be highlighted. *facename* is either a symbol naming a face, or an expression whose value is the face name to use. If you want *facename* to be a symbol that evaluates to a face, use a form like "(progn sym)".

override and *laxmatch* are flags. If *override* is t, existing fontification may be overwritten. If '`keep`', only parts not already fontified are highlighted. If '`prepend`' or '`append`', existing fontification is merged with the new, in which the new or existing fontification, respectively, takes precedence. If *laxmatch* is non-nil, no error is signalled if there is no *match* in *matcher*.

For example, an element of the form highlights (if not already highlighted):

```
 "\\<foo\\>" Discrete occurrences of "foo" in the value of the
variable ‘font-lock-keyword-face’.
 ("fu\\(bar\\)" . 1) Substring "bar" within all occurrences of "fubar" in
the value of ‘font-lock-keyword-face’.
 ("fubar" . fubar-face) Occurrences of "fubar" in the value of ‘fubar-face’.
 ("foo\\|bar" 0 foo-bar-face t)
Occurrences of either "foo" or "bar" in the value
of ‘foo-bar-face’, even if already highlighted.
```

match-anchored should be of the form:

(*matcher pre-match-form post-match-form match-highlight* ...)

Where *matcher* is as for *match-highlight* with one exception; see below. *pre-match-form* and *post-match-form* are evaluated before the first, and after the last, instance MATCH-ANCHORED's *matcher* is used. Therefore they can be used to initialise before, and cleanup after, *matcher* is used. Typically, *pre-match-form* is used to move to some position relative to the original *matcher*, before starting with MATCH-ANCHORED's *matcher*. *post-match-form* might be used to move, before resuming with MATCH-ANCHORED's parent's *matcher*.

For example, an element of the form highlights (if not already highlighted):

```
("\\<anchor\\>" (0 anchor-face) ("\\<item\\>" nil nil (0 item-face)))

Discrete occurrences of "anchor" in the value of ‘anchor-face’, and subsequent
discrete occurrences of "item" (on the same line) in the value of ‘item-face’.
(Here pre-match-form and post-match-form are nil.  Therefore "item" is
initially searched for starting from the end of the match of "anchor", and
searching for subsequent instance of "anchor" resumes from where searching
for "item" concluded.)
```

The above-mentioned exception is as follows. The limit of the *matcher* search defaults to the end of the line after *pre-match-form* is evaluated. However, if *pre-match-form* returns a position greater than the position after *pre-match-form* is evaluated, that position is

used as the limit of the search. It is generally a bad idea to return a position greater than the end of the line, i.e., cause the *matcher* search to span lines.

Note that the *match-anchored* feature is experimental; in the future, we may replace it with other ways of providing this functionality.

These regular expressions should not match text which spans lines. While M-x `font-lock-fontify-buffer` handles multi-line patterns correctly, updating when you edit the buffer does not, since it considers text one line at a time.

Be very careful composing regexps for this list; the wrong pattern can dramatically slow things down!

7 Customizing the Display

There are two ways of customizing the frame display when using Isamode: by setting Emacs user-options and by setting X resources.

7.1 Display options

The user options controlling the display mostly appear in the file '`isa-display.el`'.

isa-multiple-frame-mode User Option
If non-nil, use multiple frames.
Setting this to nil restricts Isabelle-interaction mode to use a single frame.

To toggle this setting while Isamode is running, use the function `isa-toggle-multi-frame`.

The default value is `nil`.

isa-toggle-multi-frame &optional *set* Command
Toggle use of multiple frames, or switch on if arg non-nil.

isa-startup-defaults User Option
List of symbol names of associated buffers to initialise by default.
The buffers on this list will be automatically started when an Isabelle session is started.

The default value starts a proofstate display and a ruletable display.

The list of symbols allowed is given by `isa-possible-associated-buffer-names`.

The default value is `(proofstate)`.

isa-possible-associated-buffer-names Variable
Names of types of buffers which may be associated to an Isabelle session.

isa-default-menubar Variable
The base menubar (in XEmacs format) for Isabelle interaction mode.
A value of nil means the only menus that appear are the Isabelle ones. You can set this to `default-menubar` (in XEmacs), to have the usual menus in addition. This may clutter the display a little, or result in menus being lost off the end of the menubar!

7.2 Display options, continued

isa-single-frame-display-props Variable
Display properties for Isabelle buffers in single frame mode.
This variable controls the default heights for the various Isabelle buffers in single frame mode, and also a ''`shrink to fit`'' attribute. It is an association list of mode symbols and cons cells, of the form:

(*mode-name* (*prop-list*))

where *prop-list* is a list of property cons values, of the form

`(window-height .` *integer*`)` `or` `(shrink-to-fit .` *boolean*`)`

(Note: these properties are implemented inside Isamode and are not a standard part of XEmacs.)

The default value for `isa-single-frame-display-props` is:

```
(setq isa-single-frame-display-props
   ((proofstate-mode     (window-height . 8))
    (listener-mode       (window-height . 5))
    (ruletable-mode      (window-height . 15))
    (ruletable-mode      (shrink-to-fit . t)))
)
```

isa-multi-frame-display-props Variable

Display properties for Isabelle buffers in multiple frame mode.
This variable controls the the default frame parameters for multiple frame mode. It is an association list used to set symbol properties. The properties are standard XEmacs ones.

The default value for `isa-multi-frame-display-props` looks something like this:

```
(setq isa-multi-frame-display-props
  '((proofstate-mode     (frame-name . proofstate))
    (listener-mode       (frame-name . listener))
    (ruletable-mode      (frame-name . ruletable))
    (isa-mode            (frame-name . isabelle))
    (ruletable           (instance-limit . 3))
    (ruletable           (max-frame-height . 25))
    (ruletable           (frame-defaults .
                          ((top . 560)   (left . 5)
                           (height . 22) (width . 60)
                           (menu-bar-lines . 0)
                           (minibuffer . nil)
                           )))
    (isabelle            (frame-defaults .
                          ((top .  0)    (left .  0)
                           (height . 35) (width . 80)
                           )))
    (listener            (frame-defaults .
                          ((top . 620)   (left . 530)
                           (height . 7)  (width . 65)
                           (menu-bar-lines . 0)
                           (minibuffer . nil)
                           )))
    (proofstate          (frame-defaults .
                          ((top .  30)    (left . 700)
                           (height . 35)  (width . 50)
                           (menu-bar-lines . 0)
                           (minibuffer . nil)
                           (vertical-scroll-bars . nil)
                           ))))
)
```

This shows the setting of the frame names for the Isabelle buffer modes (which automatically gives them individual X-windows), and the frame default parameters for each of the frame names. Some of these values may be inappropriate for your display, so you may wish to change them in your '`~/.emacs`'.

See the Emacs lisp reference manual for more information about frame parameters.

7.3 X Resources

Frame properties are an unsatisfactory part of the Emacs window interface: the kind of settings they control are usually kept in the X resources database. Luckily, XEmacs allows X resources to be used in place of `isa-multi-frame-display-props`.

Here are some sample X defaults for XEmacs that work well on my X terminal. See the file '`etc/Xdefaults-xemacs`' in the Isamode distribution for more examples, and the documentation in See Info file '`emacs`', node '`Resources X`'. Further details are in See Info file '`elisp`', node '`X Frame Parameters`'.

```
/*  ===== X resources for Isamode in XEmacs ===== */

/* some frame geometries. */

Emacs*listener.geometry:   55x10
Emacs*ruletable.geometry:  57x15
Emacs*proofstate.geometry: 52x35

/* ruletable frames have large red headings */

Emacs*ruletable.ruletableGroupname.attributeFont:  \
        -adobe-helvetica-bold-r-normal--*-160-75-75-*-*-iso8859-1

Emacs*ruletable.ruletableGroupname.attributeForeground: red

/* proofstate frames have bold goal subgoal numbers */

Emacs*proofstate.proofstateGoal.attributeFont:  \
        -adobe-courier-bold-r-normal--*-140-75-75-*-*-iso8859-1
Emacs*proofstate.proofstateSubgoalNumber.attributeFont:  \
        -adobe-courier-bold-r-normal--*-140-75-75-*-*-iso8859-1
```

Frame properties take precedence over X resources, so you will need

```
(setq isa-multi-frame-display-props
 '((proofstate-mode     (frame-name . proofstate))
   (listener-mode       (frame-name . listener))
   (ruletable-mode      (frame-name . ruletable))
   (isa-mode            (frame-name . isabelle))
   (ruletable           (instance-limit . 3))))
```

or something similar in your '`~/.emacs`' to ensure the resource specifications have effect.

8 Acknowledgements

My thanks go to the following people for their suggestions, testing and bug-reports: Sara Kalvala, Larry Paulson, Christian Prehofer, Tobias Nipkow, Chris Owens, Ole Steen Rasmussen, Claudio Russo, Markus Wenzel.

I am happy to receive any grateful comments, clever suggestions, or moaning reports of problems concerning this program and its documentation. The only rewards for developing Isamode are your feedback and the nefarious pleasures of elisp hacking, so please drop a line!

David Aspinall,
Laboratory for foundations of Computer Science,
Division of Informatics,
University of Edinburgh,
King's Buildings,
Edinburgh.
email: `David.Aspinall@dcs.ed.ac.uk`

Appendix A Obtaining the software

Isamode

The Isamode distribution consists of Emacs lisp files and documentation. You can obtain the latest version by anonymous `ftp` from Edinburgh.

1. Connect to `ftp ftp.dcs.ed.ac.uk`
2. Login as '`anonymous`' with your internet address as password
3. `type binary`
4. `cd pub/da`
5. `get Isamode.tar.gz`

It's easier inside Emacs! Visit the virtual directory

```
/anonymous@ftp.dcs.ed.ac.uk:pub/da/
```

move to the file '`Isamode.tar.gz`' and press ⟨C⟩ to copy the file to a local directory.

The file '`Isamode.tar.gz`' should be gunzipped, then extracted using tar:

```
gunzip -c Isamode.tar.gz | tar xf -
```

Please tell me if you have any problems.

Isabelle

Of course, Isamode is useless without Isabelle!

Isabelle is available by anonymous ftp from the University of Cambridge.

Appendix B Installing Isamode

The current version of Isamode has been tested with XEmacs version 20.4. XEmacs changes continuously, and it is too difficult to make Isamode backward compatible with earlier versions. So make sure you are running the latest and greatest!

Version 2.2 of Isamode was the last to be compatible with FSF GNU Emacs version 18. Version 2.6 of Isamode was the last to be compatible with FSF GNU Emacs altogether. The divergence between FSF GNU Emacs and XEmacs makes it too difficult to maintain Isamode for both varieties. XEmacs provides a more friendly user interface, and the inclusion of toolbars, images, etc.

To install Isamode, you need to do the following things:

1. Edit the file '`isa-site.el`'. Here you must edit the variable *isa-isatool-command* to be the full path name for the command `isatool`, if it is not on your `PATH` by default. If it *is* on `PATH` when Emacs is started, you may not need to do any customization at all.
2. Edit the file '`Makefile`'. There are two variables to control where the files are to be installed.
3. Execute the command `make compile` to create '`.elc`' files
4. Execute the command `make install` to install '`.el`' and '`.elc`' files. It may be most convenient to put the file '`isa-site.el`' somewhere on the Emacs load path.
5. If you want to start Isabelle from inside any Emacs session, add the line:

   ```
   (load "isa-site")
   ```

   to an Emacs site-default file or your personal '`~/.emacs`'. Otherwise you may wish to start special Emacs sessions for Isabelle using one of the tools provided with Isabelle.

More detailed instructions are given in the file '`INSTALL.txt`' which is part of the distribution. Installation should be straightforward, so please let me know if you have any problems.

To customize Isamode, read about the user options available in Chapter 7 [Display Customization], page 20 and Section 6.4 [Theory mode User Options], page 17.

Happy theorem proving!

Appendix C Ideas and Dreams

Isamode was developed partly as a proof of concept: that writing a front end inside a new-generation Emacs is a viable way of attaining a powerful user-interface for Isabelle and similar systems. I believe this has been demonstrated by now. But improvements in the interface are certainly still possible.

Several ideas for future improvement of Isamode are included below.

Please feel free to send me comments on this list, more ideas to add, or — even better — offers to implement something from it!

Proof stepping

More flexible ways of using `isa-thy-use-line` would be nice. Presently it works best inside ML files: it could be linked in with the listener or `*proof*` buffer (see below) in a function `isa-replay-proof` which grabs a proof (interactive or non-interactive) from an ML file into a fresh proof buffer and begins replaying it. Ways of moving through interactive lines would be useful (they should really be part of `sml-mode`).

Customization per-logic

We perhaps need *logic*`.el` and even *theory*`.el` files for customization per-logic and per-theory. At present too much is hard coded inside Isamode — it would be good to have ways of generating (at least first attempts at) rule tables from the theorem databases, or from the HTML that is automatically generated by Isabelle during building logics. As an alternative to introducing more files, a special `emacs` section could be added to theory files. We'd like ways of controlling and customizing the tactics that appear in the menus and the completion table in `isa-completion-list`.

Better theory handling

We really need both `logic-name` and `root-logic-name`, the latter used for rule tables, provers, etc. This is because people may save logic images that are extensions of built-in logics. Also, user logics should be provided for in a better way. Perhaps Emacs should attempt to understand or communicate with Isabelle about the theory tree structure and make system? Browsing of theories?

Improved Listener

It should listen actively! A complete history of commands is a bit too general really, particularly when it is available in the interaction buffer, by paging backwards or using the comint history mechanism. A better idea would be to have multiple listeners or `*proof*` buffers which accumulates commands since the most recent `"goal"`. The `push_proof` commands could be interpreted to manage several such buffers. Commands issued to Isabelle which don't affect/reference the proofstate could be appended to a "proof preamble" (e.g. making lists of rules, simpsets, etc). With a little bit of help from Isabelle (see below), `undo();` can be correctly interpreted and lines deleted from the transcript as needed. Lines might be annotated with the corresponding proof level. It can't be perfectly robust, of course, but we should be able to do pretty well. Suggested functions: `extract-interactive-proof`, `extract-batch-proof` to create *proof* buffers from regions of ML files, and a command `replay-proof-step` to step through each command in an interactive proof.

Caching proof states

Stepping backward and forward through proof levels is slow, probably mostly due to Isabelle's pretty printer. Emacs could cache them in an internal buffer, given an understanding of `undo`, etc, (or at least, a conservative cache-flushing when the proof level is seen to go down).

Rule table categories

This scheme might be improved by making use of ML's type information to help infer the rule category. Type information could prevent Emacs from constructing an ill-typed tactic, for example, by showing the difference between subgoal indexed tactics (`int->tactic`) and tactics the apply to the whole proofstate (`tactic`).

Type information isn't fine enough to distinguish between the different kinds of rules, however, so a better solution would be to add the rule categories to the rule tables themselves, at the expense of added complexity in emacs lisp. This may be implemented in some future version of Isamode. (In the meantime, and at any rate, following the naming convention above for your own logics is probably a good idea). Then general ways of handling things such as LK's '`pack`' type may be integrated.

A different problem is the choice of the default tactic itself, given the category of the identifier. Often you *won't* want to apply `fast_tac` to everything of type `claset`! There needs to be a more flexible way of specifying what the default tactic is (for example, based on the last menu action).

Improved batchify

`isa-batchify` could do with improving: it should be a bit more flexible, permitting proofs starting with `goal ...`, and even not ending with `result` (it might prompt for a rule name). (Also, we shouldn't use `replace-regexp` in the function — it means undo doesn't work properly.)

Tracing Output

Should probably appear a temporary buffer or the proof state window. Prompting might be in the minibuffer (or dialogue boxes?).

Text insertion

Is the behaviour of tactic, etc, shortcuts annoyingly inconsistent? - should `assume`, `resolve prems`, etc, wait for ⟨RET⟩ to submit input? Another possibility is to use the minibuffer for input in tactic-entering functions: (or editing the current line of ML) if the tactic menus become accessible from the Proof State buffer, the Isabelle interaction buffer itself would become only of esoteric interest. (for "advanced" use!).

Improved templates

In theory mode files; 'hot spots' (that disappear when you type something into them) are nicer than minibuffer prompting, I think. But probably not worth the trouble.

Sensitizing rules

Ruletable option for filtering of resolvable rules to highlight only those applicable in the current proofstate.

Buttons

They're quicker to use than pull-down menus: we might have a special buffer of them (e.g. Tactics table), a special section in the ruletable buffer, or tool bars. In a tactics table we could have standard tactics, and then additional tactics on a per-logic basis. But we probably don't really want a proliferation of windows . . .

Driving by mouse

We're on the way a completely mouse-driven interface. Whether you want to use a mouse or not, it should be possible to use the mouse most of the time, for simple proofs. More control keys for the interaction buffer would be useful (for example, completion on rule names after all).

Help from Isabelle

Several of the improvements suggested so far would benefit from 'behind the scenes' interaction with Isabelle. At the moment, the interface is fairly simple minded – proof states are displayed in another buffer simply by removing anything from Isabelle's output that looks like a list of subgoals. It would probably be better to get Isabelle to participate actively in the interaction, writing some of the interface code in ML and sending special control sequences to Emacs. It would probably be more appropriate to site all of the rule table data, tactic names, etc, within Isabelle.

Bells and whistles

Icon bitmaps, colours, more fontification, etc, etc . . . . It might be good to highlight and tag completed proofs somehow, or annotate in some way. Also, implement "intelligent" filtering.

Special characters

Isamode has rudimentary support for the special X fonts with symbols, via the variable `isa-use-special-font`. The 8-bit input functions provided in the Isabelle distribution are not yet integrated (I tried them once and locked up my keyboard!). This is future work, including automatic processing of theory files to convert between the symbol format and the long-hand TEX-oriented names.

Questions

What ideas or improvements are missing from the above list?
What are the priorities?
What's the state of art in other theorem prover interfaces?
Which things can be done usefully and easily inside Emacs and which not?

Index

L

M

N

O

P

R

S

T

U

X

Table of Contents