Ergo 5 Reference Manual

Nickson, Robinson and Utting

December 24, 2003

Contents

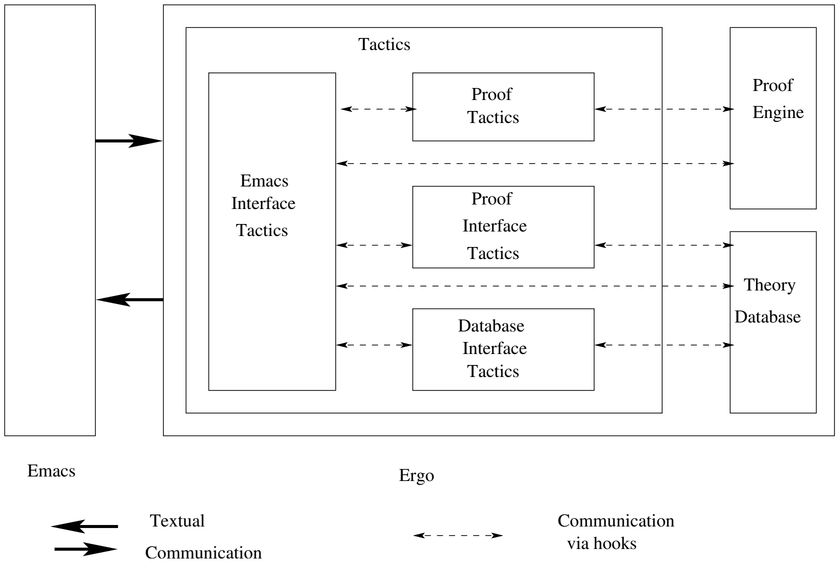

Figure 1: The layered architecture of Ergo

1 Introduction

This document describes the commands available in the theorem prover Ergo. It assumes that the reader is familiar, to some extent, with Ergo (refer to the online tutorials). As such, it is intended to serve as a reference only.

Ergo is a generalised Sequent Calculus theorem prover designed and implemented at the Software Verification Research Centre at the University of Queensland.

The architecture of Ergo is based on a layered design (Figure 1). The *proof engine* forms the core of Ergo. The *theory database* is Ergo's repository of information containing object logics. The next layer contains tactics (user−writable code written in Qu−Prolog [**?**]) that implement the command−line interface. This layer constructs the theory database and controls the proof engine. Finally, the user front−end (Emacs) communicates with Ergo via tactic interfaces.

The commands available in Ergo depend upon the user interface and theory being used: the default *gumtree* proof interface (section 3), constructing theories (section 4), or the higher level Ergo−Emacs interface [**?**].

Every Ergo command has a declaration that gives the command's name, the types and modes of its arguments (section 6.4), the interface from which it is available and a description of the command. The textual descriptors for each command can be extended with *concept nodes*.

Concept nodes describe groups of commands or other general concepts and are linked into a network by cross−references. They allow the automatic generation of on−line help (p6), as well as the remaining sections of this manual (tex_doc_trees p52). Note that the generated manual also includes an index of commands and parameters.

In addition to the commands supplied by each user interface, there are also a number of general commands (available from all user interfaces). These allow users to inspect the ergo database, obtain help and so on (section 2).

The remainder of this manual is organised as follows:

- General Commands (Section 2)
- Proof Commands (Section 3)
- Theory/Database Commands (Section 4)
- Parameters (Section 5)
- Concepts (Section 6)
- Index of Commands and Parameters.

2 General commands (viewing data and setting parameters)

The following commands are available in all Ergo user interfaces. They allow you to browse the contents of the Ergo database and the online help system, inspect and set global parameters (p72) etc.

dependent_rules(Rules, Dependents) (p6)
help (p6)
help(Topic, Thy) (p6)
macro(File) (p7)
msg(Caller, Level, Msg) (p7)
new_annotation(Type, Doc) (p7)
runtime (p8)
set Parameter === Value (p8)
shell(String) (p8)
show (p8)
show(Topic) (p8)
show(Topic, Thy) (p9)
show_dependent_rules(Rules) (p9)
show_proof_file(File) (p9)
show_proof_script_file(File) (p9)
show_rule(Name, Thy, Kind) (p10)
show_table(Table) (p10)
show_tables (p10)
show_tactic(Anns, Tac) (p10)

show_tactic_script(Tac) (p10)
show_tactic_table(Table) (p11)
show_tactic_tables (p11)
silent_cmd(Cmd) (p11)
statistics (p11)
tactic Decl (p11)

2.1 dependent_rules(Rules, Dependents)

Dependents is the list of all theorems whose proofs depend (directly or indirectly) on Rules.

Rules and Dependents are both lists of rule names of the form Thy:Name, where Thy is a theory that is accessible from the current theory and Name is the short name of a rule in Thy.

mode dependent_rules(@list(rulename), ?list(rulename)).

2.2 help

Entry point for help about dbms commands.

mode help.

2.3 help Topic
help(Topic, Thy)

Get help on Topic, which may be a command name, cmd/arity, a node name, or a string (double−quoted).

- Cmd: search for any node documenting Cmd of any arity.
- Cmd/Arity: search for any node documenting Cmd/Arity. (Arity may also be: parameter)
- Node: display the named node.
- ”String”: display all nodes containing String.

Thy may be a theory name or ’global’. If Thy is specified, then only that theory is searched.

mode help(@term).
mode help(@term, @theory).

2.4 macro_and_delete(File) macro(File)

Sets the input stream to be the file called File. When the end of that file is reached, the original input stream (eg. the terminal) is restored and, for macro_and_delete only, File is deleted.

Macro files can be nested, to an arbitrary depth.

The theory/tactic load_path (p73) is *not* used to find File.

mode macro_and_delete(@filename).
mode macro(@filename).

2.5 msg(Caller, Msg) msg(Caller, Level, Msg)

This is the recommended printing mechanism for all informational messages from tactics and from other parts of the prover.

Caller: the name of the tactic or package that generated the message.
Level: an integer specifying the importance of the message.
Msg: a valid writelist (p89).

msg/2 always succeeds, but only prints Msg if the current verbosity (p75) number is greater than or equal to Level (if there is no Level argument, then Level defaults to 1).

Msg is automatically indented and followed by a newline, except when it is the empty list, in which case nothing is printed.

mode msg(@nonvar, @writelist).
mode msg(@nonvar, @integer, @writelist).

2.6 new_annotation(Type, Doc)

This command declares a new kind of proof node annotation.

Type must be an atom or a compound term whose arguments describe the types of the annotations arguments. That is, Type is similar to mode declarations (p83), but without the @/?/+/− prefixes on the type names.

Doc must be the name of a documentation node that describes the new kind of annotation.

It is not good style to include proof node names in annotations, or any other kind of data that will become meaningless when a proof tree is abstracted into a tactic.

Example: new_annotation(usercmd(string), usercmd_annotation).

mode new_annotation(@ground, @atom).

2.7 runtime

Displays total user cputime for the current Ergo session and the relative cputime since the last call to runtime/0 or statistics(runtime,...).

mode runtime.

2.8 set Parameter === Value

Set one of the parameters (p72).

mode set(@compound).

2.9 shell(String)

Executes the command in String by passing it to a shell subprocess.

mode shell(@chars).

2.10 show

Lists available topics for show/1 and show/2.

mode show.

2.11 show(Topic)

Lists all facts in the Ergo database about topic(s) Topic. Theory specific topics use the current theory.

mode show(?topics).

2.12 show(Topic, Thy)

Lists all facts in theory Thy about topic Topic. Thy may be 'global', 'parameter' or the name of a loaded theory.

mode show(?topics, @theory).
mode show(?topics, @list(theory)).

2.13 show_dependent_rules(Rules)

Lists all the theorems in the accessible theories whose proofs depend (directly or indirectly) on Rules.

Rules may be a single rule name of the form Thy:Name, or a list of rule names of that form.

mode show_dependent_rules(@rulename).
mode show_dependent_rules(@list(rulename)).

2.14 show_proof(RuleName)
show_proof_file(File)

Converts the proof file for RuleName into a proof tactic and prints it. Annotations are ignored; only the primitive rules are shown, combined by sequential and parallel composition operators according to the structure of the proof.

mode show_proof(@rulename).
mode show_proof_file(@filename).

2.15 show_proof_script(RuleName)
show_proof_script_file(File)

Converts the proof file for RuleName into a proof script and prints it. It does this by processing the usercmd annotations (p76) that appear within the proof file. The show_proof_script_file form uses the given File directly, without searching.

mode show_proof_script(@rulename).
mode show_proof_script_file(@filename).

2.16 show_rule(Rule) show_rule(Rule, Kind) show_rule(Name, Thy, Kind)

Display inference rule Rule. If Rule is not fully specified, displays all rules on backtracking. If Kind is specified, restrict display to that kind of rule (postulate, theorem, fundamental_rule, definition).

mode show_rule(?rulename).
mode show_rule(?rulename, ?atom).
mode show_rule(?atom, ?theory, ?atom).

2.17 show_table(Table)

List all the currently loaded entries in table.

mode show_table(?atom).

2.18 show_tables

List all the currently loaded tables.

mode show_tables.

2.19 show_tactic(Anns, Tac)

Pretty−prints the gumtree tactic Tac. Anns=true means the tactic includes annotations; otherwise, they are discarded.

mode show_tactic(@bool, @gumtreetactic).

2.20 show_tactic_script(Tac)

Converts the gumtree tactic Tac into a proof script and prints it. It does this by processing the usercmd annotations (p76) that appear within Tac.

mode show_tactic_script(@gumtreetactic).

2.21 show_tactic_table(Table)

List all the currently loaded entries in the tactic table.

mode show_tactic_table(?atom).

2.22 show_tactic_tables

List all the currently loaded tactic tables.

mode show_tactic_tables.

2.23 silent_cmd(Cmd)

Call the general command Cmd silently − that is no prompt is produced after the command is executed. This can be useful in, for example, GUI interfaces where a general command should not cause any I/O on the usual user interaction window.

mode silent_cmd(@term).

2.24 statistics

Print all Qu−Prolog statistics for the current Ergo session.

mode statistics.

2.25 tactic Decl

This command is used to declare, define and delete tactics. Examples:
tactic tac1(X) mode tac1(thy(term)) arity 1 −−−> N
tactic tac1(X) doc doc1 mode tac1(thy(term)) arity 1 −−−> N
declares tac1/1 as a gumtree tactic whose argument is of type thy(term) with arity 1 −−−> N. A doc node containing doc1 is added for the tactic.

tactic tac1(X) === term(X)
defines a 'clause' for tac1/1

tactic delete tac1(Pattern)
deletes all clauses for tac1/1 whose head argument matches Pattern.

Tactics must be declared before defined or deleted. Multiple identical declarations of tactics are allowed. Only dynmaic tactics (i.e. tactics that were not loaded from compiled files) can be deleted.

mode tactic(@compound).

3 Proof commands (The 'GumTree' User Interface for Ergo 5 proofs)

'Gumtree' is a user interface for creating Ergo 5 proofs. The number in the prompt is the number of the first open (non−proved) node in the current proof tree. However, you may choose to work on any open node.

The following commands and groups of commands are available:

3.1 Proof Information Commands

The following commands viewing information about the current proof.

3.1.1 show_annotations(Ids)

Prints all the annotations of the given proof node(s).

mode show_annotations(@nodeid).
mode show_annotations(@list(nodeid)).

3.1.2 show_constraints
show_constraints(Vars)

Prints out the constraints on the current proof. These include any not_free_in constraints on the proof variables, plus any user−defined constraint predicates and context assumptions. The show_constraints(Vars) version only prints constraints that are related to the variables in Vars, whereas show_constraints/0 shows constraints associated with all the variables in the root node and all the open nodes of the proof Note that if you want to see whether or not there are any constraints between two variables, you typically need to mention *both* those variables in the Vars list.

mode show_constraints.
mode show_constraints(@list(anyvar)).

3.1.3 show_context(Id)

Prints details of context of the given proof node.

mode show_context(@nodeid).

3.1.4 show_node(Id)

Prints details about the given proof node.

mode show_node(@nodeid).

3.1.5 show_node_and_context(Id)
show_nodes_and_context(Ids)

Print details of given proof node(s) and their context.

mode show_node_and_context(@nodeid).
mode show_nodes_and_context(@list(nodeid)).

3.1.6 show_open_nodes
show_open_nodes(Id)

Display the names and conclusions of all open nodes. If the Id argument is supplied, it must be the name of a node in the current proof, and just the open nodes within that subtree of the proof will be displayed.

mode show_open_nodes.
mode show_open_nodes(@nodeid).

3.1.7 show_open_nodes_and_context
show_open_nodes_and_context(Id)

Prints details of all open proof nodes and their contexts. If the Id argument is supplied, it must be the name of a node in the current proof, and just the open nodes within that subtree of the proof will be displayed.

mode show_open_nodes_and_context.
mode show_open_nodes_and_context(@nodeid).

3.1.8 show_prooftree(Root)

Display a representation of the tree of proofnodes below Root.

mode show_prooftree(@nodeid).

3.1.9 show_prooftree_tactic(Root)

Display the proof tree below Root as a tactic. Note that the tactic displayed does not provide any information about the order in which the tree was generated. To do this use show_subproof.

mode show_prooftree_tactic(@nodeid).

3.1.10 show_subproof(Node)

Display the subtree of the current proof rooted at Node. Annotations are ignored; only the primitive rules are shown, combined by sequential and parallel composition operators according to the structure of the proof.

mode show_subproof(@nodeid).

3.1.11 show_subproof_script(Node)

Prints a proof script of the Node subtree of the current proof. It does this by processing the usercmd annotations (p76) that appear within the subtree. If Node was proved by a command outside the subtree, then the resulting proof script will be incomplete. To alert you to this possibility, a warning message is printed if Node has no usercmd annotation.

mode show_subproof_script(@nodeid).

3.2 Proof Tree Commands

These commands allow you to extend and manage the current proof tree. Most of the commands take a 'nodeid' argument (an integer) to determine which node of the proof tree they will be applied to.

The proof tree commands are:

3.2.1 −−−> Names

This names the first N open nodes, where N is the length of Names. Names must be a list of variables or a single variable (N=1).

This command is equivalent to: * ::: [names(Names), skip].

mode −−−>(−nodeid).
mode −−−>(+list(nodeid)).

3.2.2 annotate(L)

A primitive gumtree_tactic (p80): Add the annotation(s) L to the current subgoal. The subgoal is not affected; like ‘one’, this tactic passes a single subgoal unchanged to the next tactic in a sequential composition.

mode annotate(@list(annotation)) **arity** 1 −−−> 1.

3.2.3 annotation(PNode, Ann)

True when Ann unifies with one of the existing annotations on proof node PNode.

mode annotation(@nodeid, ?annotation).

3.2.4 annotations(PNode, AnnList)

AnnList is a list of all the annotations on proof node PNode. This is similar to the annotation/2 (p17) command but returns the list of all annotations, rather than one by one.

mode annotations(@nodeid, ?list(annotation)).

3.2.5 commit(CmdNum)
commit(CmdNum, Label)

Removes all undo points and alternative solutions between the current command and the command numbered CmdNum. This means that those commands cannot be undone individually, only as a block.

The history display changes to reflect this by packaging up all the commands after CmdNum into a single history entry marked as 'commit:'. If Label is given, it will be displayed as the 'action' part of the new history entry.

The primary reason for using this command is to reduce the size of the history list or to commit to the current solution of some non−deterministic command.

mode commit(@integer).
mode commit(@integer, @string).

3.2.6 constraints(Vars, Dist, NFI, Other)

A primitive gumtree_tactic (p80): The constraints of the rule being proved −
Vars is the list of variables in the rule
Dist is the list of disjointness constraints
NFI is the list of not−free−in constraints
Other is the list of other constraints

mode constraints(?list(thy(term)), ?list(thy(term)), ?list(thy(term)), ?list(thy(term)))
arity A −−−> A.

3.2.7 context(K, N, T)

A primitive gumtree_tactic (p80): The current subgoal has a context of kind K, number N, term T. The subgoal is not affected; like 'one', this tactic passes a single subgoal unchanged to the next tactic in a sequential composition. It is nondeterministic: alternative contexts are returned on backtracking.

mode context(@cname, ?ctermid, ?thy(term)) **arity** 1 ---> 1.

3.2.8 context_assume(CName, PNode, CId, CTerm)

This procedure ensures that CId:::CTerm is available within the CName context of PNode. If it is not already present, and the CName context of PNode is inherited from the root node of the proof, then CId:::CTerm is added to the root node of the proof as an assumption.

mode context_assume(@cname, @proofnode, ?ctermid, ?thy(term)).

3.2.9 context_diff(PNode, Ancestor, CName, CDiff)

CDiff is the difference between the CName context of proof nodes PNode and Ancestor. PNode must be within the subtree of Ancestor.

CDiff is a 'context difference' expression, which is similar to a 'context specification', but contains literal context term identifiers from the proof, rather than variables for context term identifiers. This is because it is more useful for user interfaces to see the current reality of the proof.

mode context_diff(@proofnode, @proofnode, @cname, ?cdiff).

3.2.10 context_diff_to_spec(CDiff, VarMap, CSpec)

This procedure converts the 'context difference' expression CDiff into a 'context specification' (p88) expression CSpec, by replacing literal context term identities (integers) with corresponding variables. These variables will be instantiated back to new identities each time the resulting CSpec is used to create a new context list.

VarMap determines the replacements that will be performed to obtain CSpec from CDiff. VarMap determines the mapping from context term identities to variables. It is an open list containing a cid(CId)=Var term for each CId in CDiff. If necessary, these terms will be added to VarMap by this procedure. VarMap may also contain proofnode(Id)=Var terms and these will be used to replace any literal proof node identities that appear in use(CName,ProofNode) terms in CDiff.

mode context_diff_to_spec(@cdiff, ?list(compound), ?cspec).

3.2.11 context_index(K, I, CId)

A primitive gumtree_tactic (p80): The current subgoal has a context of kind K, index I and ID CId. The subgoal is not affected; like 'one', this tactic passes a single subgoal unchanged to the next tactic in a sequential composition. This tactic is not normally expected to be used by users, it is used for constructing and rerunning proofs to avoid searching for context terms.

mode context_index(@cname, ?number, ?ctermid) **arity** 1 −−−> 1.

3.2.12 context_terms(T)

A primitive gumtree_tactic (p80): The current subgoal has context terms T. The context terms is a list of terms of the form ContextType = ContextList where ContextList is a list of the form c(ID,Term). ID is the id of the context term Term. The subgoal is not affected; like 'one', this tactic passes a single subgoal unchanged to the next tactic in a sequential composition.

mode context_terms(?list(cterm)) **arity** 1 −−−> 1.

3.2.13 fail

A primitive gumtree_tactic (p80).

mode fail **arity** A −−−> B.

3.2.14 fix(N)

A primitive gumtree_tactic (p80): Ensure there are exactly N current subgoals.

mode fix(?integer) **arity** A −−−> A.

3.2.15 history

Display the history of proof−related commands, with the most recent command last. An 'undo' command will undo the effect of the last command in the command history.

mode history.

3.2.16 is_disabled_rule(Thy, Rule)

Succeeds if Thy:Rule is disabled from being used in the current proof. For example, during the proof of a postulated rule, all rules that depend upon that rule are disabled so that circular proofs between rules cannot be created. Unfortunately, there is no way of displaying all the disabled rules as yet.

mode is_disabled_rule(@theory, @shortrulename).

3.2.17 iuse(Query)

A primitive gumtree_tactic (p80): Transform the current open node using an instantiated rule satisfying Query.

mode iuse(?query) **arity** 1 −−−> A.

3.2.18 iuse(Query, Output)

A primitive gumtree_tactic (p80): Transform the current open node using an instantiated rule satisfying Query.

mode iuse(?query, ?thy(term)) **arity** 1 −−−> A.

3.2.19 name(N)

A primitive gumtree_tactic (p80): The variable N is bound to the name of the current subgoal.

mode name(?nodeid) **arity** 1 −−−> 1.

3.2.20 names(L)

A primitive gumtree_tactic (p80): L are the names of some prefix of the current subgoal list. names([a,b,c]) == [name(a), name(b)), name(c)].

mode names(?list(nodeid)) **arity** A −−−> A.

3.2.21 one

A primitive gumtree_tactic (p80): Ensure there is a single current subgoal.

mode one **arity** 1 ---> 1.

3.2.22 premises(Prems)

A primitive gumtree_tactic (p80): Prems is the list of premises of the rule.

mode premises(?list(term)) **arity** A ---> A.

3.2.23 proof_disable_rules(Rules)

Prevents all the rules in Rules from being used in future proof steps within the current proof. Rules must be a list of rule names, each of the form Thy:ShortName.

mode proof_disable_rules(@list(rulename)).

3.2.24 proof_new_node(Id, Parent, Term)

This routine constructs a new proof node.

This procedure is for use by the core prover only, so is not allowed within the user−supplied constraints of inference rules.

It copies all the parent context lists to the constructed node, so should *not* be used to construct root nodes of proof trees.

mode proof_new_node(?nodeid, @nodeid, ?thy(term)).

3.2.25 proof_step(Thy, Rule, Id, Children)

Proves node Id, by applying the rule Thy:Rule. Children is the list of child node names that Thy:Rule produces.

mode proof_step(?theory, ?rulename, @nodeid, ?list(nodeid)).

3.2.26 proof_tactic(Ann, NameMap, Root, Tac)

Tac is a tactic that will regenerate the proof tree given by PNode. PNode can be the name of any proof node −− it does not have to be the root node of the current proof.

Ann must be true or false. If it is true, then Tac includes commands to regenerate all the annotations of the proof tree as well. If it is false, no annotations will be recorded.

NameMap must be a closed list of NodeId=Var pairs. The generated tactic will include a name(Var) command for each node NodeId. This gives a way of naming the nodes that will be created when the tactic runs.

mode proof_tactic(@bool, @list(compound), @nodeid, ?gumtreetactic).

3.2.27 prooftreeIDs(Root, Tree)

Tree is a representation of the tree of proofnodes below Root.

mode prooftreeIDs(@nodeid, ?compound).

3.2.28 rerun_proof(RuleName)

Applies the tactic from a proof file to the current proof node.

RuleName may be of the form Thy:Name, or just Name, in which case Thy defaults to the current theory. In both cases, the tactic is read from the file called 'Thy/Name.proof' and the load_path (p73) is used to determine the directories in which this file may appear.

mode rerun_proof(@rulename).

3.2.29 rerun_proof_file(File)

Applies the tactic from the given proof file to the current proof node.

File must be the full path (relative or absolute) to the proof file (these typically have a .proof extension).

mode rerun_proof_file(@filename).

3.2.30 retry

Finds the next solution to the last proof−related command if it has multiple solutions.

mode retry.

3.2.31 rule(R)

A primitive gumtree_tactic (p80): invoke an inference rule.

mode rule(?rulename) **arity** 1 −−−> A.

3.2.32 skip

A primitive gumtree_tactic (p80): Accepts all current subgoals (this is a left− and right−identity for sequential composition).

mode skip **arity** A −−−> B.

3.2.33 srule(R)

A primitive gumtree_tactic (p80): invoke an inference rule. Structural unify is first attempted on the rule and proof terms and if that fails normal unify is used. Structural unify avoids changes of bound variables and and so reduces the number of substitutions and delayed not−free−in constraints.

mode srule(?rulename) **arity** 1 −−−> A.

3.2.34 start_wi

Start a simple window inference interface that simply displays the window relation and focus of the top−level window.

mode start_wi.

3.2.35 stop_wi

Stop the window inference interface.

mode stop_wi.

3.2.36 tactic_annotations(Tac, Patt, Anns)

Extracts annotations from the gumtree tactic Tac. Anns is the list of all annotations within Tac that match Pattern.

mode tactic_annotations(@gumtreetactic, @term, ?list(term)).

3.2.37 term(T)

A primitive gumtree_tactic (p80): T is the term of the current subgoal. The subgoal is not affected; like one (p21), this tactic passes a single subgoal unchanged to the next tactic in a sequential composition.

mode term(?thy(term)) **arity** 1 −−−> 1.

3.2.38 terms(L)

A primitive gumtree_tactic (p80): L is a list of terms of some prefix of the current subgoal list. terms([a,b,c]) == [term(a), term(b)), term(c)].

mode terms(?list(thy(term))) **arity** A −−−> A.

3.2.39 trace
trace(Caller)
trace(Caller, Level, Msg)

A primitive gumtree_tactic (p80): Accepts all current subgoals. If the current verbosity (p75) is greater than or equal to Level, It displays a message of the form: Caller[subgoals]: Msg. 'trace' is equivalent to 'trace(trace, 1, [''])'.

mode trace.
mode trace(@nonvar).
mode trace(@nonvar, @integer, @writelist).

3.2.40 two

A primitive gumtree_tactic (p80): Ensure there are exactly two current subgoals.

mode two **arity** 2 −−−> 2.

3.2.41 undo

Undoes the last proof−related command.

mode undo.

3.2.42 undo(CmdNum)

Undoes all commands back to (and including) the one numbered CmdNum.

mode undo(@integer).

3.2.43 use(Query)

A primitive gumtree_tactic (p80): Transform the current open node using a rule satisfying Query.

mode use(?query) **arity** 1 −−−> A.

3.2.44 use(Query, Output)

A primitive gumtree_tactic (p80): Transform the current open node using a rule satisfying Query.

mode use(?query, ?thy(term)) **arity** 1 −−−> A.

3.2.45 with_context_delaying(Code)
with_context_delaying(Patt, Code)

These procedures change the behaviour of most procedures that search context lists so that they add the given context term if it is not already present in the context. This makes it easy to prove anything! However, all such added context terms are recorded as constraints of each inference rule generated from the proof. This is typically useful for proving inference rules that manipulate context lists.

This changed behaviour applies only during the execution of Code. It is allowable for Code to contain nested calls to these procedures.

Patt provides more control over what context terms can be added. Only those that are unifiable with Patt can be added.

mode with_context_delaying(+code).
mode with_context_delaying(@thy(term), +code).

3.2.46 zero

A primitive gumtree_tactic (p80): Ensure there is no current subgoal (this is a left− and right−identity for parallel composition).

mode zero **arity** 0 −−−> 0.

3.2.47 {G}

A primitive gumtree_tactic (p80): Execute Prolog goal G (must not affect proof!), As with 'skip', any number of subgoals are passed unchanged to the next tactic in a sequential composition.

mode {}(?term) **arity** A −−−> A.

3.3 Query Procedures

Query Procedures for Proof Nodes and Proof Tree.

irules(Query) (p27)
proof_constraints(Root, Vars, Dist, NFI, User, Context) (p27)
proof_node_children(Id, Children) (p28)
proof_node_context(Id, Contexts) (p28)
proof_node_parent(Id, Parent) (p28)
proof_node_rule(Id, Thy, Rule, RuleType) (p28)
proof_node_rule_and_children(Id, Thy, Rule, RuleType, Children) (p28)
proof_node_terms(Ids, Preds) (p29)
proof_node_timestamp(Node, Timestamp) (p29)
proof_open_nodes(Id, OpenNodes) (p29)
proof_root(Node) (p29)
proof_root_postulate(Node, Thy, Name) (p29)

proof_root_sequents(Sequents) (p30)
rules(Query) (p30)
theorem_constraints(Vars, Dist, NFI, Others) (p30)

3.3.1 irules(Query)

A primitive gumtree_tactic (p80): Print out all instantiated rules matching Query and current open proof node.

mode irules(?query).

3.3.2 proof_constraints(Nodes, Dist, NFI, User, Context) proof_constraints(Root, Vars, Dist, NFI, User, Context)

These commands return lists of all the various delayed constraints associated with the proof, such as (x not_free_in A).

Root must be the root node of a proof and Vars can be any subset of the object variables and metavariables that appear anywhere within that proof.

The Nodes form of this command is perhaps easier to use, but provides less control over which variables you are interested in. Nodes must be a list of proof nodes, and the Root node of the proof must be the first one in the list. Typically, Nodes will contain all the open nodes of the proof, as well as the Root node.

Dist is a list of distinctness (x not_free_in y) constraints. These show which object variables are assumed to be distinct (that is, not unifiable).

NFI is a list of (x not_free_in Term) constraints.

User is a list of other constraints on the proof terms, including user−defined rule_constraint predicates. For example: delay_until(ground(X),integer(X)) might appear in this list.

Context is a list of context−related constraints, such as context_search(CName, Root, IdVar, Term). These reflect assumptions about the context terms that will be available when the current rule has been proved and is being applied in future proofs.

mode proof_constraints(@list(nodeid), −list(rule_constraint), −list(rule_constraint), −list(rule_constraint), −list(rule_constraint)).
mode proof_constraints(@nodeid, @list(var), −list(rule_constraint), −list(rule_constraint), −list(rule_constraint), −list(rule_constraint)).

3.3.3 proof_node_children(Id, Children)

Children is the list of subproof node names associated with the proof node Id. This fails if Id refers to an open node.

mode proof_node_children(@nodeid, ?list(nodeid)).

3.3.4 proof_node_context(Id, Contexts)

Contexts is a list of CName=CTermIds for the context of Id.

WARNING: some of the terms returned may have constraints associated with them, such as x not_free_in A. These constraints may prevent a context term from being used to discharge a subproof even though it looks as if the context term should match the conclusion of the subproof. The constraints are not returned or recorded by this procedure. So, you should regard the results of this procedure as giving only a rough indication (a superset) of what context terms are available for use. The only way to be certain that a context term can be used to prove a given conclusion is to apply the appropriate inference rule.

mode proof_node_context(@nodeid, @list(compound)).

3.3.5 proof_node_parent(Id, Parent)

Parent is the name of the parent node of the proof node Id.

mode proof_node_parent(@nodeid, ?nodeid).

3.3.6 proof_node_rule(Id, Thy, Rule, RuleType)

True if Thy:Rule was used to discharge proof node Id. This fails if Id refers to an open node. RuleType is the type of rule tactic used − i.e. rule or srule.

mode proof_node_rule(@nodeid, ?theory, ?rulename, ?atom).

3.3.7 proof_node_rule_and_children(Id, Thy, Rule, RuleType, Children)

True if Thy:Rule was used to discharge proof node Id and Children is the list of subproof node names produced by that rule. This fails if Id refers to an open node. RuleType is the type of rule tactic used − i.e. rule or srule.

mode proof_node_rule_and_children(@nodeid, ?theory, ?rulename, ?atom, ?list(nodeid)).

3.3.8 proof_node_term(Id, Pred)
proof_node_terms(Ids, Preds)

Pred is the main term (the conclusion) of the proof node Id.

mode proof_node_term(@nodeid, ?thy(term)).
mode proof_node_terms(@list(nodeid), ?list(thy(term))).

3.3.9 proof_node_timestamp(Node, Timestamp)

Succeeds when Timestamp is the timestamp for a closed node Node.

mode proof_node_timestamp(@nodeid, ?integer).

3.3.10 proof_open_nodes(Id, OpenNodes)

OpenNodes is the list of node names of all open nodes in the subtree whose root is Id.

mode proof_open_nodes(@nodeid, ?list(nodeid)).

3.3.11 proof_root(Node)

Succeeds when Node is the root node of a proof

mode proof_root(@nodeid).

3.3.12 proof_root_postulate(Node, Thy, Name)

Succeeds when Node is the root node of a proof of postulate Thy:Name. Name is always the short name of the postulate (no parameters). Currently, Thy is always the current theory, but in the future it may be a subtheory of the current theory.

mode proof_root_postulate(@nodeid, ?theory, ?rulename).

3.3.13 proof_root_sequents(Sequents)

Get the Sequents of the root node (node 1).

mode proof_root_sequents(−list(sequent)).

3.3.14 rules(Query)

A primitive gumtree_tactic (p80): Print out all rules matching Query.

mode rules(?query).

3.3.15 theorem_constraints(Vars, Dist, NFI, Others)

Get the constraint information associated with the theorem being proved. Vars is the list of variables in the statement of the theorem, Dist is the list of distinctness constraints, NFI is the list of not_free_in constraints and Others is the list of other constraints.

mode theorem_constraints(−list(var), −list(rule_constraint), −list(rule_constraint), −list(rule_constraint)).

3.4 Save/Close/Read Proof Commands

These commands are for:

- saving the current proof staring at a given node to a file
- converting the current proof to an inference rule (or theorem)
- quiting the current proof without saving anything
- reading a proof file

proof_complete(Name, Root, OpenNodes, Constraints) (p31)
proof_complete_postulate(Root, Unproved) (p31)
quit_proof (p32)
read_proof_file(File, Version, Tac, Properties) (p32)
save_postulate_proof(Root) (p32)
save_proof(Name, Root, OpenNodes, Constraints) (p33)
save_proof_to_file(Thy, Name, Node, OpenNodes, Constraints) (p33)
write_proof_file(File, PNode, Properties) (p34)

3.4.1 proof_complete(Name, Root, OpenNodes, Constraints)

Converts a proof into an inference rule, called Name, and writes that new inference rule into the current theory.

Name is the name of the new inference rule. It must be an atom or a compound term whose functor is an atom. In the latter case, the arguments are treated as being parameters of the resulting rule.

Root is the name of the root node of a proof. That is, a node that was created by the proof_start/3 procedure.

OpenNodes is a list of the names of all unproved nodes in the proof. For example, if OpenNodes=[56,45] and Root=1, then the resulting inference rule will be:

```
Seq56,
Seq45
----------
Seq1
```

where Seq45, Seq56 and Seq1 stand for the sequents at proof nodes 56, 45 and 1, respectively. OpenNodes is typically used to order the antecedents of the rule and perhaps to check that the proof does not contain other unproved nodes. If it is a variable, then it is instantiated to contain all the unproved nodes of the proof in some arbitrary order.

Constraints is a comma−separated sequence of 'rule_constraint' commands. These commands will be added as constraints on the resulting rule, in addition to other constraints that are already associated with the proof. If no extra constraints are needed, it should be 'true'.

mode proof_complete(@std_nonvar, @nodeid, ?list(nodeid), +constraints).

3.4.2 proof_complete_postulate(Root)
proof_complete_postulate(Root, Unproved)

This completes the proof of a postulate.

Root must be the root node of a proof of a postulate. If the proof has correctly discharged that postulate, without instantiating variables or adding extra constraints, then the postulate will be marked as proved.

Unproved is a list of the names of all unproved nodes in the proof, in an order that corresponds to the order of the upper sequents of the postulated inference rule. The one argument version sets Unproved=[].

mode proof_complete_postulate(@proofnode).
mode proof_complete_postulate(@proofnode, ?list(proofnode)).

3.4.3 quit_proof

Quits out of the current proof without saving anything.

mode quit_proof.

3.4.4 read_proof(RuleName, Version, Tac, Properties) read_proof_file(File, Version, Tac, Properties)

These both read a proof record (tactic + property list) from a file.

The read_proof/4 command uses the load_path (p73) parameter to search for the proof file, whereas the read_proof_file/4 command requires a precise path to a proof file.

RuleName may be of the form Thy:Name, or just Name, in which case Thy defaults to the current theory. In both cases, the tactic is read from a file called 'Thy/Name.proof', and the load_path (p73) is used to determine the directories under which this file may appear.

File must be the full path (relative or absolute) to a proof file. Proof files typically have a .proof extension.

Version is an atom that describes the version of the proof file format that the proof was read from. The latest version is '5.0'.

Tac is the gumtree tactic that is read from the proof file. Properties is a closed list of PropertyName=Value pairs. See Proof File Properties (p87).

mode read_proof(@rulename, ?atom, ?gumtreetactic, ?propertylist).
mode read_proof_file(@filename, ?atom, ?gumtreetactic, ?propertylist).

3.4.5 save_postulate_proof(Root)

Completes the proof of a postulate and saves the proof in a file. That is, this command invokes proof_complete_postulate/1 (p31), then write_proof_file/3 (p34)which saves the proof into the file './Theory/Name.proof)'.

Root must be the name of the root node of the proof of a postulate. If the unproved nodes and constraints of the proof do not match the postulate, then this command will fail with an error message.

Note the difference between this command and the save_proof (p33) command. This command can only be called within the proof of a postulate (started via prove_rule/1 (p70)) and it does not create a new rule, but just marks the postulate as having been proved. The save_proof command can be called at any time during any proof and always creates a *new* rule in the current theory.

mode save_postulate_proof(@nodeid).

3.4.6 save_proof(Name, Root)
save_proof(Name, Root, OpenNodes, Constraints)

Saves a proof into the current theory (as a new rule) and into a file. That is, this command invokes proof_complete/4 (p31), then write_proof_file/3 (p34) which saves the proof into the file './Theory/Name.proof)'.

Name is the name of the new rule in the current theory. The outermost functor of Name, say F, is also used to determine the file in which the proof is saved: 'Thy/F.proof'.

Root must be the name of the root node of a proof.

OpenNodes determines the order of the upper sequents of the new inference rule. Constraints is any additional constraints that the new inference rule should have. These two arguments are described more fully in the proof_complete (p31) command documentation.

Name, OpenNodes and Constraints are saved in the proof file so that the same inference rule can be generated when the proof is rerun.

The two argument form sets OpenNodes=[] and Constraints=true. This is useful for saving simple theorems.

Note the difference between this command and the save_postulate_proof (p32) command. This command can be called at any time, during any proof, and it always creates a *new* rule in the current theory. The save_postulate_rule command must be called within the proof of a postulate (started via prove_rule/1 (p70)) and it does not create a new rule, but just marks the postulate as having been proved.

mode save_proof(@rulename, @nodeid).
mode save_proof(@rulename, @nodeid, ?list(nodeid), +constraints).

3.4.7 save_proof_to_file(Thy, Name, Node, OpenNodes, Constraints)

Writes the proof subtree Node into File, but does not create a theorem. This command is invoked automatically by the higher level proof saving commands, such as save_postulate_proof/1 (p32) and save_proof/4 (p33). However, you may occasionally find it useful to call it directly instead. For instance, if you want to save an incomplete proof for further work later, or if you wish to save just one subtree of a proof (the higher level commands can only save entire proofs).

The proof is saved into the file 'Thy/Name.proof' under the current directory. Thy and Name should both be atoms that do not contain any characters that might be illegal within filenames However, if Name is a compound term, then the functor of that term is used as the filename instead of Name.

Node can be the name of any proof node -- it does not have to be the root node of the current proof, though it often is.

OpenNodes and Constraints are saved as additional information in the proof file. They can be useful for rerunning the proof. OpenNodes must be a closed list of proof node names, or a variable. If it is a variable, then it is instantiated to all the open nodes in the subtree specified by Node.

mode save_proof_to_file(@atom, @rulename, @nodeid, ?list(nodeid), @constraints).

3.4.8 write_proof_file(File, PNode, Properties)

Writes the proof subtree PNode into File

File is typically an atom of the form 'Theory/Name.proof'. Any directories that File refers to must already exist.

PNode can be the name of any proof node -- it does not have to be the root node of the current proof, though it usually is.

Properties must be a closed list of PropertyName=Value pairs. See Proof File Properties (p87).

mode write_proof_file(@atom, @nodeid, @list(compound)).

3.5 Tactic Combinator Commands

These commands are for combining several tactics in parallel or in sequence.

succs(T) (p36)
with_context_delaying(Patt, T) (p36)
|(T, U) (p36)

3.5.1 !(T)

A gumtree_tactic (p80) combinator: do T, but discard alternatives.

mode !(+tactic) **arity** A.

3.5.2 ;(T, U)

A gumtree_tactic (p80) combinator: do T then U.

mode ;(+tactic, +tactic) **arity** A ---> B.

3.5.3 [T1,T2,...]

A gumtree_tactic (p80) combinator: do each Ti in parallel on a subsequence of the current open nodes.

mode .(+tactic, +list(tactic)) **arity** A ---> B.

3.5.4 deabbrev_and_call_gumtree_tactic(T)

Deabbreviate and call the gumtree tactic.

mode deabbrev_and_call_gumtree_tactic(+tactic).

3.5.5 exec_gumtree_tactic(T)

Execute tactic.

mode exec_gumtree_tactic(+tactic).

3.5.6 fails(T)

A gumtree_tactic (p80) combinator: The tactic fails(T) succeeds if the tactic T would fail.

mode fails(+tactic) **arity** A −−−> A.

3.5.7 if_tac(Cond, Then, Else)

A gumtree_tactic (p80) combinator: Do Cond (removing choices) followed by Then, otherwise do Else. It has the same semantics as Prolog's if−then−else.

mode if_tac(+tactic, +tactic, +tactic) **arity** A −−−> B.

3.5.8 succs(T)

A gumtree_tactic (p80) combinator: The tactic succs(T) succeeds if the tactic T would succeed.

mode succs(+tactic) **arity** A −−−> A.

3.5.9 with_context_delaying(Patt, T)

A gumtree_tactic (p80) combinator: it executes tactic T in a mode which allows implicit context to be added to the current proof. That is, within T procedures that search the context of a proof node will add a required context term if it is not already there and it is unifiable with Patt. All context terms that are added in this way become constraints on any inference rules that result from this proof.

mode with_context_delaying(@thy(term), +tactic) **arity** A −−−> B.

3.5.10 |(T, U)

A gumtree_tactic (p80) combinator: do T or U.

mode |(+tactic, +tactic) **arity** A −−−> B.

3.6 User Defined Gumtree Tactics

The current user defined gumtree tactics are:

3.6.1 add_hyp(Tactic)

The same as add_hyp(Tactic, _, _).

mode add_hyp(?tactic) **arity** A.

3.6.2 add_hyp(Tactic, Result, Hyp)

Use Tactic to justify adding the new hypothesis Result with ID Hyp.

mode add_hyp(?tactic, ?thy(term), ?ctermid) **arity** A.

3.6.3 add_hyp_ante(Tactic, Ante)

The same as add_hyp_ante(Tactic, _, Ante, _).

mode add_hyp_ante(?tactic, ?tactic) **arity** A.

3.6.4 add_hyp_ante(Tactic, Result, Ante, Hyp)

Use Tactic to justify adding the new hypothesis Result with ID Hyp. Tactic is a rule application with antecedents that are discharged by the tactic Ante.

mode add_hyp_ante(?tactic, ?thy(term), ?tactic, ?ctermid) **arity** A.

3.6.5 add_hyp_f(Tactic)

The same as add_hyp_f(Tactic, rule(assump)).

mode add_hyp_f(?tactic) **arity** A.

3.6.6 add_hyp_f(Tactic1, Tactic2)

The same as add_hyp_f(Tactic1, Tactic2, _, _).

mode add_hyp_f(?tactic, ?tactic) **arity** A.

3.6.7 add_hyp_f(Tactic1, Tactic2, Result, Hyp)

Use Tactic1 and Tactic2 to justify adding the new hypothesis Result with ID Hyp. Tactic1 is a rule application with an antecedent justified by Tactic2.

mode add_hyp_f(?tactic, ?tactic, ?thy(term), ?ctermid) **arity** A.

3.6.8 close_win

Close the current window.

mode close_win **arity** A.

3.6.9 focus(F)

F is the focus of the top−level window.

mode focus(?thy(term)) **arity** A −−−> A.

3.6.10 oh(Hyp)

Open a window at the hyp labelled Hyp.

mode oh(?ctermid) **arity** A.

3.6.11 oh(NewHyp, OldHyp)

Open a window at the hyp labelled OldHyp. NewHyp is the label of the hypothesis produced when the window is closed.

mode oh(?ctermid, ?ctermid) **arity** A.

3.6.12 op(PosList)

Open a window at the subterm described in the position list. The position list is a list of positions that determine the subterm of interest where each position is one of: an positive integer − specifies an argument of a structure an object variable, or a term b(x) where x is an object variable − specifies the body of a quantified term, where the supplied object variable matches against the bound variable a term t(N) where N is a positive integer − specifies the type argument of the N'th bound variable of a quantified term.

mode op(?list(term)) **arity** A.

3.6.13 start_window_proof

Start a window style proof − rewrite the goal F to F <= true. The window inference commands (tactics) are:

start_window_proof/0 (p40)

stop_window_proof/0 (p41)

op/1 (p39)

close_win/0 (p39)

oh/2 (p39)

oh/1 (p39)

trans_def_f/1 (p42)

trans_def_f/0 (p42)

trans_def_r/1 (p43)

trans_def_r/0 (p43)

trans_t/1 (p45)

trans_rt/2 (p44)

trans_rt/1 (p44)

trans/2 (p42)

trans/1 (p42)

trans_f/2 (p43)

trans_f/1 (p43)

trans_r/2 (p44)

trans_r/1 (p44)

trans_t_ante/2 (p45)

trans_rt_ante/3 (p45)

trans_rt_ante/2 (p45)

trans_ante/3 (p42)

trans_ante/2 (p42)

add_hyp/3 (p38)

add_hyp/1 (p38)

add_hyp_ante/4 (p38)

add_hyp_ante/2 (p38)

The transformation tactics take one or more arguments that are tactics. Such tactics are typically rule applications.

mode start_window_proof **arity** A.

3.6.14 stop_window_proof

Stop a window style proof.

mode stop_window_proof **arity** A.

3.6.15 tactic_table(Name)

This proof command executes one of the tactics in the tactic table called Name.

The term of the current proof node are used to restrict the set of matching tactics in Table.

mode tactic_table(?cmdtable) **arity** 1 −−−> A.

3.6.16 tactic_table(Name, Val)

This proof command executes one of the tactics in the tactic table called Name.

The term of the current proof node and Val are used to restrict the set of matching tactics in Table.

mode tactic_table(?cmdtable, ?term) **arity** 1 −−−> A.

3.6.17 trans(Tactic)

The same as trans(Tactic, _).

mode trans(?tactic) **arity** A.

3.6.18 trans(Tactic, Result)

Transform the focus to Result using Tactic.

mode trans(?tactic, ?thy(term)) **arity** A.

3.6.19 trans_ante(Tactic, Ante)

The same as trans_ante(Tactic, _, Ante).

mode trans_ante(?tactic, ?tactic) **arity** A.

3.6.20 trans_ante(Tactic, Result, Ante)

Transorm the focus to Result using Tactic. Tactic is a rule application with antecedents that are discharged by the tactic Ante.

mode trans_ante(?tactic, ?thy(term), ?tactic) **arity** A.

3.6.21 trans_def_f

The same as trans_def_f(_).

mode trans_def_f **arity** A.

3.6.22 trans_def_f(Result)

Transform the window focus by unfolding a definition. The result of the unfolding is Result.

mode trans_def_f(?thy(term)) **arity** A.

3.6.23 trans_def_r

The same as trans_def_r(_).

mode trans_def_r **arity** A.

3.6.24 trans_def_r(Result)

Transform the window focus by folding a definition. The result of the folding is Result.

mode trans_def_r(?thy(term)) **arity** A.

3.6.25 trans_f(Tactic)

The same as trans_f(Tactic, _).

mode trans_f(?tactic) **arity** A.

3.6.26 trans_f(Tactic, Result)

Transform the focus to Result using Tactic in the 'forward direction'.

mode trans_f(?tactic, ?thy(term)) **arity** A.

3.6.27 trans_f_ante(Tactic, Ante)

The same as trans_f_ante(Tactic, _, Ante).

mode trans_f_ante(?tactic, ?tactic) **arity** A.

3.6.28 trans_f_ante(Tactic, Result, Ante)

Transorm the focus to Result using Tactic in the 'forward direction'. Tactic is a rule application with antecedents that are discharged by the tactic Ante.

mode trans_f_ante(?tactic, ?thy(term), ?tactic) **arity** A.

3.6.29 trans_r(Tactic)

The same as trans_r(Tactic, _).

mode trans_r(?tactic) **arity** A.

3.6.30 trans_r(Tactic, Result)

Transform the focus to Result using Tactic in the 'reverse direction'.

mode trans_r(?tactic, ?thy(term)) **arity** A.

3.6.31 trans_r_ante(Tactic, Ante)

The same as trans_r_ante(Tactic, _, Ante).

mode trans_r_ante(?tactic, ?tactic) **arity** A.

3.6.32 trans_r_ante(Tactic, Result, Ante)

Transorm the focus to Result using Tactic in the 'reverse direction'. Tactic is a rule application with antecedents that are discharged by the tactic Ante.

mode trans_r_ante(?tactic, ?thy(term), ?tactic) **arity** A.

3.6.33 trans_rt(Tactic)

The same as trans_rt(Tactic, _).

mode trans_rt(?tactic) **arity** A.

3.6.34 trans_rt(Tactic, Result)

Transform the true focus to Result using Tactic.

mode trans_rt(?tactic, ?thy(term)) **arity** A.

3.6.35 trans_rt_ante(Tactic, Ante)

The same as trans_rt_ante(Tactic, _, Ante).

mode trans_rt_ante(?tactic, ?tactic) **arity** A.

3.6.36 trans_rt_ante(Tactic, Result, Ante)

The true focus is transformed to Result using Tactic. Tactic is a rule application with antecedents that are discharged by the tactic Ante.

mode trans_rt_ante(?tactic, ?thy(term), ?tactic) **arity** A.

3.6.37 trans_rule(Tactic)

Apply a sequent calculus style Tactic when the window relation is <=.

mode trans_rule(?tactic) **arity** A.

3.6.38 trans_t(Tactic)

Transform the focus of the window to true using Tactic with input arity one (typically a rule application).

mode trans_t(?tactic) **arity** A.

3.6.39 trans_t_ante(Tactic, Ante)

Transorm the focus to true using Tactic. Tactic is a rule application that has antecedents that are discharged by the tactic Ante.

mode trans_t_ante(?tactic, ?tactic) **arity** A.

3.7 Variable Instantiation Commands

These commands control the instantiation of proof variables (both meta−variables and object−variables). The proof variables are split into two classes: ‘fixed variables’ and ‘new variables’.

Fixed variables are those that appear in the original theorem. They can only be instantiated manually, not automatically.

New variables have typically been introduced during the proof. They may be instantiated automatically during proof steps.

4 Theory/Database Commands (The DBMS user interface)

This section describes the 'DBMS' user interface, which allows you to build theories and manage the Ergo database.

Most DBMS commands operate on the 'current' theory, which is the one whose name is displayed in the prompt. You can set the current theory by typing: [TheoryName].

The following groups of commands are available: General Commands (viewing data, setting parameters etc.) (p5)

4.1 I/O Syntactic Sugar Commands.

This set of commands allows sugared syntax to be defined for the terms of the current theory. Each theory can have a different set of abbreviations, operators etc.

delete_operators(Thy) (p49)
delete_visible_theory(Thy) (p49)
disable_abbreviation (p50)
inherit_all_operators(Thy) (p50)
inherit_all_visible_theories(Thy) (p50)

4.1.1 abbreviation E === I.

Defines E as an abbreviation for I.

I may be of the form: Term provided Cond, where Cond is 'input' for input−only abbreviations, or some Prolog code to perform a conversion between E and I. However, there are various constraints on such code which not yet described here.

mode abbreviation(@compound).

4.1.2 add_abbrevs(Thy)

Copies all abbreviations in theory Thy into the current theory.

mode add_abbrevs(@theory).

4.1.3 add_operators(Thy)
add_all_operators(Thy)

Copy operators from theory Thy into the current theory. Operators that clash with existing operators of the current theory will generate an error message and will not be added to the current theory.

Note that add_operators/1 copies only those operators that were originally defined in Thy, whereas add_all_operators/1 also copies any operators that have been copied into Thy from other theories. That is, add_all_operators/1 is transitive, whereas add_operators/1 is non−transitive.

mode add_operators(@theory).
mode add_all_operators(@theory).

4.1.4 add_operator(Prec, Kind, Name)

Add an operator, called Name, to the current theory. Note that this command only affects the concrete syntax of the theory and it is not necessary for Name to be a function or quantifier of the current theory before it can be declared as an operator.

Kind can be: prefix, postfix, nonassoc, leftassoc, rightassoc or quant.

Prec defines the precedence of the operator. It can be any number between 1 and 1200, where smaller numbers mean tighter binding. Most user−defined operators should have a precedence less than 990, because precedences above that are reserved for the prover itself.

This command fails (and generates an error message) if the new operator clashes with existing global operators or with existing operators of the current theory.

mode add_operator(@integer, @atom, @atom).

4.1.5 add_visible_theory(Thy)

Adds Thy to the set of visible theories.

Eg. visible_theory(t) allows 't.c' to be written as 'c'.

mode add_visible_theory(@theory).

4.1.6 deabbrev(T)

Display a term with all abbreviations unfolded. This is a useful way of seeing the unambiguous internal form of any term that is legal in the current theory.

mode deabbrev(@thy(term)).

4.1.7 deabbrev_focus

Display the current focus with all abbreviations unfolded. This is a useful way of seeing the unambiguous internal form of the current focus.

mode deabbrev_focus.

4.1.8 declare_object_vars(A)

Declare an object variable prefix for the current theory.

Eg. declare_object_vars(tim) declares tim, tim_0, tim_1...

mode declare_object_vars(@atom).

4.1.9 delete_abbrev(Abbrev)
delete_abbrev(Abbrev, Term)

Deletes the given abbreviation. It checks that Abbrev and Term are not both variables, since that would delete an arbitrary abbreviation.

mode delete_abbrev(?term).
mode delete_abbrev(?term, ?thy(term)).

4.1.10 delete_abbrevs(Thy)

Deletes all abbreviations that were copied from theory Thy. If Thy is a variable, ALL abbreviations will be deleted.

mode delete_abbrevs(@theory).

4.1.11 delete_operator(Name)
delete_operator(Name, Kind)

Delete the theory−specific operator called Name. This can be used to delete operators that were added directly to the current theory, or operators that were copied from some other theory (see also delete_operators/1 (p49)).

mode delete_operator(@atom).
mode delete_operator(@atom, ?atom).

4.1.12 delete_operators(Thy)

Delete all operators that were copied from theory Thy. If Thy is a variable, ALL operators of the current theory If Thy is the name of the current theory, all operators added directly to the current theory will be deleted, but operators copied from ancestor theories will be left unchanged.

mode delete_operators(?theory).

4.1.13 delete_visible_theory(Thy)

Deletes Thy from the set of visible theories.

mode delete_visible_theory(@theory).

4.1.14 enable_abbreviation
disable_abbreviation

Enable/disable automatic abbreviation of theory terms on output.

mode enable_abbreviation.
mode disable_abbreviation.

4.1.15 inherit_all_operators(Thy)

Inherits all operators and obvar prefixes from theory Thy. Thy must be an ancestor of the current theory.

This inheritance of operators is not transitive. That is, if Thy inherits operators from some other theory, this command does not inherit them into the current theory. (If you wish to check that Thy does not inherit operators, simply check that it does not contain any inherited_operators/1 facts.)

The difference between this command and the add_all_operators/1 (p47) command, is that this one shares the operators of Thy, which means that future changes to the operators of Thy will automatically be reflected in the current theory, whereas add_all_operators/1 makes a local copy of all the operators from Thy, thus allowing the operators of the current theory to diverge from those of Thy in the future. Also, operator inheritance links between theories cannot be deleted, whereas operators that are copied can be deleted using the delete_operator/2 (p49) or delete_operators/1 (p49) commands.

mode inherit_all_operators(@theory).

4.1.16 inherit_all_visible_theories(Thy)

Makes all theories that are visible in Thy visible in the current theory. Does this by sharing the visible_theory/1 facts of Thy, rather than copying them into the current theory. Thy must be an ancestor of the current theory.

mode inherit_all_visible_theories(@theory).

4.2 Miscellaneous Commands.

The following miscellaneous commands are available:

[Thy]. (p51)
quit_proof (p51)
tex_doc_clear_memory (p51)

tex_doc_cmdtables(File, Tables) (p51)
tex_doc_tree(File, Root) (p52)
tex_doc_trees(File, Roots) (p52)

4.2.1 [Thy].

Change to theory Thy, and load it if it is not already loaded. This will also cause all its ancestor theories to be loaded.

The directories that the theories are loaded from will be determined by the load_path (p73) parameter.

mode .(@theory, @emptylist).

4.2.2 quit_proof

This does nothing when called from the theory management level. It exists simply so that a quit_proof command is available within all of Ergo's command−line user interfaces.

mode quit_proof.

4.2.3 tex_doc_clear_memory

Clears all the tex_doc_done/1 facts, which record which doc_nodes have been printed out by the texdoc procedures. This is done automatically whenever the texdoc packages is loaded.

mode tex_doc_clear_memory.

4.2.4 tex_doc_cmdtables(File, Tables)

Writes all the commands in Tables out into File, in LaTeX format. Command show(command_table) shows possible values for Tables.

mode tex_doc_cmdtables(@atom, @atom).
mode tex_doc_cmdtables(@atom, @list(atom)).

4.2.5 tex_doc_tree(File, Root)

Writes a tree of documentation nodes into File in LaTeX format. File should be a quoted atom. For example: '../doc/refman.tex'. Root must be the name of a global doc_node, which will be the root of the generated documentation tree.

The subtrees of each node of the tree are found by following cross references that appear in the doc_extension facts. requires code.sty.

mode tex_doc_tree(@atom, @docnode).

4.2.6 tex_doc_trees(File, Roots)

Writes a forest of documentation nodes into File in LaTeX format. File should be a quoted atom. For example: '../doc/refman.tex'. Roots must be a list of the names of global doc_nodes, which will be the roots of the generated documentation trees.

The subtrees of each forest are found by following cross references that appear in the doc_extension facts.

mode tex_doc_trees(@atom, @list(docnode)).

4.3 Rule/Tactic Table Commands

A 'tactic table' is a set of rules and tactics. For example, one tactic table might contain various simplification rules, while another might contain rule and tactics for type checking. The three main advantages of grouping rules and tactics together into tactic tables are:

Abstraction: You can apply a tactic table to a proof node without caring exactly which rule or tactic within the tactic table will be used.

Extensibility: You can add extra rules/tactics to a tactic table at, any time, even from a different theory. This is more flexible than writing a tactic which tries a fixed set of rules.

Efficiency: Tables are designed to allow efficient indexing so that the term of the current proof node, and an optional argument to table/2 are used to speed up the search for an appropriate rule/tactic within the table. Currently, the indexing is rather naive (just standard Prolog clause indexing), but we have plans to use discrimination trees as fast whole−term indexes.

The idea is that you define a 'tactic table', using declare_tactic_table then use in_tactic_table/2,3,4 to add tactics to that table. Each tactic that you add should accept one proof node as input. To add individual rules to the table, just use ‘rule(Thy:Name)’ as the value for Tactic.

After setting up a table like this, the tactics tactic_table/1 and tactic_table/2 can be called. They execute one of the tactics in the table.

4.3.1 User Defined Tactic Tables

The current user defined tactic tables are:

reverseRel The reverseRel tactic table stores rules that reverse a relation

strongerRel The strongerRel tactic table stores rules that replace a relation with a stronger relation.

4.3.2 declare_tactic_table Decl

This tactic is used to declare tables. Each table stores a set of tactics, indexed by a theory term. The typical reason for defining a table is to group together a set of related tactics or rules, then use the tactic_table/1,2 (p41) command to call one of the tactics in a given table.

declare_tactic_table Name − declare Name as the name of a tactic table. declare_tactic_table Name doc Doc − declare Name as the name of a tactic table and associate Doc as the documentation for Name.

mode declare_tactic_table(@term).

4.3.3 erase_tactic_table(Name)

Erase all entries in the current theory for the tactic table called Name.

mode erase_tactic_table(@atom).

4.3.4 in_tactic_table(Name, Tactic)
in_tactic_table(Name, Tactic, Term)
in_tactic_table(Name, Tactic, Term, Arg)

This adds Tactic into the tactic table called Name. Note that rules can be added by specifying 'rule(Thy:Name)' as the value for Tactic.

The tactic table (Name) must have been declared using declare_tactic_table/2.

Term and Arg can be used to restrict the times when Tactic will be called. If tactic_table/1 is called on a proof node containing a term T, then Tactic will only be called when T unifies with Term. The tactic_table/2 tactic is similar, but in addition, it requires its second argument to unify with Arg.

mode in_tactic_table(@atom, @tactic).
mode in_tactic_table(@atom, @tactic, ?thy(term)).
mode in_tactic_table(@atom, @tactic, ?thy(term), ?term).

4.4 Save/Exit Commands.

This set of commands allow changes made to theories and other facts to be saved to disk in various ways.

4.4.1 abandon

Exit the prover immediately without saving anything! WARNING: this command discards all theory changes and proofs done since the last save command.

mode abandon.

4.4.2 bye

If called from the outermost theory−level user interface, this command saves all theories (and global data) and exits the prover.

If called from within a nested theory−level user interface, this command will fail with an error message. You should use the resume command instead.

mode bye.

4.4.3 resume

Closes the current (nested) theory−level user interface and returns to the previous user interface (typically a suspended proof). This command cannot be used to exit the outermost theory−level user interface. Use the bye/0 command to do that.

mode resume.

4.4.4 save

Save all modified theories and global data.

mode save.

4.4.5 save_snapshot(File)

Save a snapshot of the current session into File. A saved session can be resumed later using the shell command, ”restore File”.

Expect some warnings about new definitions being ignored when you execute this command and when you restore the resulting snapshot.

mode save_snapshot(@filename).

4.4.6 touch_theory(Thy)

Marks Thy so that it will be saved by the next save command. Thy must be the name of a loaded theory, or ’global’.

mode touch_theory(@atom).

4.5 Table Commands

A 'table' is an association between terms. For example, one table might be used to associate terms with operator names to aid in the construction of rules for window inference, while another table might associate terms with rules, with the intention of using the rules in a more flexible way than with tactic tables. The idea is that you define a 'table', using declare_table then use in_table/3 to add term pairs to that table.

After setting up a table like this, the call table/3 can be used to extract information from the table.

4.5.1 User Defined Tables

The current user defined tables are:

symbols The symbol table associate a name with the top−level operator of a term.

4.5.2 declare_table Decl

This tactic is used to declare tables. Each table stores a set of term pairs. The typical reason for defining a table is to group together a set of related terms, then use the table/2 (p**??**) command to extract information from the table.

declare_table Name(Type1, Type2) − declare Name as the name of a table, where Type1 and Type2 are the types of the two terms in the table entries. declare_table Name(Type1, Type2) doc Doc − declare Name as the name of a table and associate Doc as the documentation for Name.

mode declare_table(@term).

4.5.3 erase_table(Name)

Erase all entries in the current theory for the table called Name.

mode erase_table(@atom).

4.5.4 in_table(Name, Term1, Term2)

This adds the pair of terms Term1, Term2 into the table called Name.

The table (Name) must have been declared using declare_table/2.

mode in_table(@atom, @term, @term).

4.5.5 table(Name, Term1, Term2)

This is used to look up the pair of terms Term1, Term2 in the table called Name.

The table (Name) must have been declared using declare_table/2.

mode table(@atom, ?term, ?term).

4.6 Tactic File Commands.

This set of commands allows global and theory−specific tactic files to be managed. These commands use the load_path (p73) parameter to search for the tactic files.

4.6.1 add_global_tactic_file(File)

Load the given File, which should contain global tactics (that is, tactics that are not theory−specific). File will be loaded automatically in future sessions.

If a global tactic file called File is already loaded, then this command fails with an error message.

Loading the tactics in File for this session only can be done using the command load_global_tactic_file (p59).

mode add_global_tactic_file(@filename).

4.6.2 add_tactic_file(File)
add_tactic_file(File, Thy)

Load the given File from the current theory (or the ancestor theory Thy), which should contain tactics that are related to the current theory. In future sessions, File will be loaded automatically whenever this theory is loaded.

If a tactic file called File is already loaded, this command fails with an error message. To succeed in such a case, use the command require_tactic_file (p61).

Loading the tactics in File for this session only can be done using the command load_tactic_file (p60).

mode add_tactic_file(@filename).
mode add_tactic_file(@filename, @theory).

4.6.3 compile_global_tactic_file(File)

Compile the given File, which should contain global tactics (that is, tactics that are not theory−specific).

If the compiled file has not previously been loaded, it can be loaded using add_global_tactic_file (p58) (for this and future sessions) or load_global_tactic_file (p59) (for this session only).

mode compile_global_tactic_file(@filename).

4.6.4 compile_tactic_file(File)
compile_tactic_file(File, Thy)

Compile the given source File, which should contain tactics that are related to the current theory.

If the file has not been previously loaded as a compiled file, it can be loaded using either add_tactic_file (p58) (for this and future sessions) or load_tactic_file (p60) (for this session only),

mode compile_tactic_file(@filename).
mode compile_tactic_file(@filename, @theory).

4.6.5 ensure_loaded_global_tactic_file(File)

Load File, if it has not already been loaded. File should contain global tactics (that is, tactics that are not theory−specific). File will be loaded automatically in future sessions.

If a global tactic file called File has already been loaded, this command succeeds silently. To fail in such a case, use the command load_global_tactic_file (p59).

mode ensure_loaded_global_tactic_file(@filename).

4.6.6 ensure_loaded_tactic_file(File)
ensure_loaded_tactic_file(File, Thy)

Load File, if it is not already loaded. File should contain tactics that are related to the current theory. In future sessions, File will NOT be loaded automatically whenever this theory is loaded.

If a tactic file called File is already loaded into the current, theory, this command succeeds silently. To fail in such a case, use the command load_tactic_file (p60).

mode ensure_loaded_tactic_file(@filename).
mode ensure_loaded_tactic_file(@filename, @theory).

4.6.7 load_global_tactic_file(File)

For this session only, load the given File which should contain global tactics (that is, tactics that are not theory−specific).

The global tactics in File can be loaded for all future sessions using add_global_tactic_file (p58).

mode load_global_tactic_file(@filename).

4.6.8 load_tactic_file(File)
load_tactic_file(File, Thy)

Load the given File from the current theory (or the ancestor theory Thy), which should contain tactics that are related to the current theory.

File will NOT be loaded automatically in future sessions. This is done by using the command add_tactic_file (p58).

mode load_tactic_file(@filename).
mode load_tactic_file(@filename, @theory).

4.6.9 recompile_all_tactic_files

Recompile all tactic files in the current theory. Each file is compiled unless a corresponding object file exists and is up to date. Recompiling an individual tactic file can be done using the command recompile_tactic_file (p60).

mode recompile_all_tactic_files.

4.6.10 recompile_global_tactic_file(File)

Recompile the given source File, unless a corresponding object file exists and is up to date.

If the file has not been previously loaded as a compiled file, it can be loaded using either add_tactic_file (p58) (for this and future sessions) or load_tactic_file (p60) (for this session only),

mode recompile_global_tactic_file(@filename).

4.6.11 recompile_tactic_file(File)
recompile_tactic_file(File, Thy)

Recompile the given source File (which should contain tactics that are related to the current theory) unless a corresponding object file exists and is up to date.

If the file has not been previously loaded as a compiled file, it can be loaded using either add_tactic_file (p58) (for this and future sessions) or load_tactic_file (p60) (for this session only),

mode recompile_tactic_file(@filename).
mode recompile_tactic_file(@filename, @theory).

4.6.12 reconsult_global_tactic_file(File)

Reread a previously loaded global tactic file (typically after editing it).

This command only works for tactics which have not been loaded as compiled files.

mode reconsult_global_tactic_file(@filename).

4.6.13 reconsult_tactic_file(File) reconsult_tactic_file(File, Thy)

Reread a previously loaded theory−specific tactic file after editing it.

This command only works for tactics which have not been loaded as compiled files.

mode reconsult_tactic_file(@filename).
mode reconsult_tactic_file(@filename, @theory).

4.6.14 require_global_tactic_file(File)

Add File, if it has not already been added. File should contain global tactics (that is, tactics that are not theory−specific). File will be loaded automatically in future sessions.

If a global tactic file called File has already been loaded, this command succeeds silently. To fail in such a case, use the command add_global_tactic_file (p58).

mode require_global_tactic_file(@filename).

4.6.15 require_tactic_file(File) require_tactic_file(File, Thy)

Add File, if it is has not already been added. File should contain tactics that are related to the current theory. In future sessions, File will be added automatically whenever this theory is loaded.

If a tactic file called File is already loaded into the current, theory, this command succeeds silently. To fail in such a case, use the command add_tactic_file (p58).

mode require_tactic_file(@filename).
mode require_tactic_file(@filename, @theory).

4.7 Theory Construction Commands.

This set of commands allows new theories and subtheories to be created. Each theory can inherit from one or more other theories, provided that the resulting theory graph does not contain cycles.

Each subtheory is associated with some 'owner' theory. Conceptually, all the facts in the subtheory can be regarded as belonging to its owner theory. The only difference is that the subtheory has its own name space.

4.7.1 add_subtheory(S)

Create a new (empty) subtheory of the current theory called S. Subtheories are typically created as the result of a theory interpretation. However, they can also be used to group a set of theorems under the new name S, which can then be used to restrict searching later.

mode add_subtheory(@theory).

4.7.2 add_theory(T)

Create a new empty theory called T.

mode add_theory(@theory).

4.7.3 close_theory

Marks the current theory as being "closed". Once a theory is closed, its axiomatic basis cannot be extended. That is, no further constants or fundamental inference rules (axioms) can be added to it, and no additional theories can be inherited into it. However, commands that extend the theory in a conservative way, such as definitions and proofs of theorems, are still allowed.

mode close_theory.

4.7.4 include_theory(Thys) include_theory(Thys, Syntax)

Thys may be a theory name, or a list of theory names. This command ensures that the current theory inherits (all of) Thys and that they are all visible in the current theory. Note that this does not automatically make all ancestors of Thys visible.

The Syntax argument determines how much sugared syntax is copied into the current theory from Thys.

Syntax=none means that no operators or abbreviations are copied and Thys does not become visible within the current theory. (this means that full 'Thy.Name' names must be used for all symbols from Thys).

Syntax=local means that operators and abbreviations that were defined within Thys are copied into the current theory and Thys becomes visible within the current theory (which allows 'Thys.' prefixes to be dropped, when no ambiguity arises).

Syntax=all means that all operators and abbreviations from Thys (both those that were defined within Thys and those that were copied into it from other theories) are copied into the current theory and all theories that were visible in Thys become visible in the current theory.

Syntax=share means that the current theory will SHARE all the syntax (operators, abbreviations and visible theory facts) of Thys. This sharing is not transitive. The effect of Syntax=share is similar to Syntax=all, except that future changes to the syntax of Thys will affect the current theory and sharing gives less freedom to customize the syntax of the current theory, because inherited operators etc. cannot be deleted.

For backwards compatibility, include_theory/1 sets Syntax to 'local'.

mode include_theory(@theory).
mode include_theory(@list(theory)).
mode include_theory(@theory, @atom).
mode include_theory(@list(theory), @atom).

4.7.5 inherit_theory(Thy)

Inherit theory Thy into the current theory, which must be open.

This command does not copy abbreviations or operators of Thy into the current theory or even make Thy visible. This means that full names (eg. 'Thy.Name') must be used for all symbols that come from Thy.

The add_abbrevs, add_operators and visible_theory commands can be used to define more readable notation for Thy terms. Alternatively, use the include_theory (p63) command, which provides a higher level interface for inheriting theories, complete with all their abbreviations and operators etc.

mode inherit_theory(@theory).

4.7.6 instantiate(Abs, Sub, Inst, Repar, Subint)

Instantiate abstract theory Abs into the current theory. The instantiated rules go into a new subtheory called Sub. Inst is a list of AbsTerm−−−>ConTerm instantiations. Repar is a list of Theory−−−>SubTheory reparentings −− each Theory must have already been interpreted into SubTheory. Subint is a list of AbsSub−−−>ConSub subinterpretations, describing how subtheories of Abs map to new subtheories of the current theory.

mode instantiate(@theory, @theory, @list(compound), @list(compound), @list(compound)).

4.7.7 interpret(Abs, Sub, Inst, Repar, Subint)

Interpret abstract theory Abs into the current theory. The interpreted rules go into a new subtheory called Sub. Inst is a list of AbsTerm−−−>ConTerm instantiations. Repar is a list of Theory−−−>SubTheory reparentings −− each Theory must have already been interpreted into SubTheory. Subint is a list of AbsSub−−−>ConSub subinterpretations, describing how subtheories of Abs map to new subtheories of the current theory.

mode interpret(@theory, @theory, @list(compound), @list(compound), @list(compound)).

4.8 Theory Deletion Commands.

This set of commands allows theories and subtheories to be deleted. However, no theory or subtheory can be deleted if other non−deleted theories would still depend upon it.

4.8.1 delete_subtheory(S)

Delete a subtheory of the current theory called S.

mode delete_subtheory(@theory).

4.8.2 delete_theory(T)

Delete a theory called T and all its subtheories. This will fail if it would cause the current theory to be deleted.

mode delete_theory(@theory).

4.8.3 delete_theory_and_dependents(T)

Delete T and all the dependent theories of T. This will fail if it would cause the current theory to be deleted.

mode delete_theory_and_dependents(@theory).

4.9 Theory Extension Commands.

This set of commands allow new facts to be added to the current theory.

4.9.1 Name doc DocStrings extends Nodes.

Create a doc concept node called Name. The DocStrings are as described in writelist (p89). Nodes is a comma seperated list of doc nodes.

An optional 'extends' clause adds cross−references in the named Nodes, pointing to this one. It is also possible to include 'header' clauses to generate a heading for the resulting doc node. In fact, 'header' can be used as the first infix keyword, as an alternative to 'doc'.

For example:

```
modes_help_ergo4
doc
  'Types specific to Ergo4 are:'
header
  'Ergo4 types'
extends
  modes_help.
```

mode doc(@atom, @compound).

4.9.2 add_context_name(Name)

Defines Name as a new kind of context. The semantics of that context is determined purely by the inference rules that manipulate it.

mode add_context_name(@atom).

4.9.3 axiom Name === RHS

Adds a fundamental inference rule called Name to the current theory.

RHS can be just a predicate, or an inference rule of the form:

```
  Seq1, Seq2, ... SeqN
  --------------------
  ConclSeq
provided
  Constraints.
```

The meaning of the rule is that any instance of ConclSeq that satisfies Constraints can be proved by proving the corresponding instances of Seq1...SeqN. Operationally, when this rule is applied to an open proof node P, ConclSeq is unified with that proof node, any Constraints are evaluated, then new proof nodes corresponding to Seq1..SeqN are created as children of P (the 'parent').

Now we shall describe each of the components in more detail.

Name must be an atom, or a compound term whose functor will be treated as the name of the rule and whose arguments can be used as parameters to the rule, typically to control how the rule will be instantiated when it is applied.

ConclSeq must be of the form: IdVar ::: Term. However, the 'IdVar :::' part is optional -- it is mainly used when one of the constraint predicates needs to refer to the name of the proof node to which the rule is being applied.

Each of the Seq_i sequents before the line must be of the form: IdVar ::: (CName===CSpec) &&& ... ---> Pred. However, the 'IdVar :::' part is optional, and so is everything before the '--->' part.

When the rule is applied each of these sequents becomes a node in the proof tree. IdVar is instantiated to the name of the node, (it must be a variable when the rule is defined), Pred becomes the predicate of the node and CSpec determines what CName context terms (eg. hypotheses) are available at the node.

Some logics may define several different classes of context terms. Each class of context terms has a name (eg. CName) and contains a sequence of theory terms (with optional names). Rules that manipulate multiple context lists will have several (CName===CSpec) clauses, separated by '&&&'. Each of the 'CName's must have been declared using add_context_name (p66). Note that parentheses are needed around each === expression.

The syntax of each CSpec is:

```
CSpec ::= [Id:::Term,...]
        |  CName(Node)
        |  CSpec +++ [Id:::Term,...]
        |  CSpec --- [Id:::Term,...]
        |  CSpec ??? Filter.
        |  CSpec >>> Transformer.
```

The semantics of most of these should be fairly obvious. An empty list can be used to specify an empty context, or [Id ::: f=2] specifies a context containing only f=2. The CName(Node) specification returns the CName context from Node (which must be the identity of the conclusion of the rule). The +++ and --- modifiers add and remove context terms. The ??? modifier returns the subset of the terms from CSpec that satisfy Filter. The most common Filter function is nfi(ObjectVar), which returns only those terms that do not contain free occurences of ObjectVar. It adds not_free_in constraints to metavariables to ensure this if necessary. The >>> modifier can be used to transform every context term from CSpec, but suitable Transformer functions must be defined first, and there is currently little support for doing this.

Most sequents set their context lists to be identical or similar to the context lists of the rules conclusion sequent. Several shorthands are provided to make this more convenient. Firstly, omitting a CName===CSpec entry is equivalent to CName===CName(ConclId), where ConclId is the identity variable of the conclusion sequent. Secondly, the ConclId defaults to the conclusion sequent, so the above example can be written as just CName===CName. Thirdly, 'CName === CName(ConclId) Modifiers', where Modifiers is one or more +++, --- or >>> clauses, can be abbreviated to just CName Modifiers. Finally, context term identities (Id:::) can be omitted if desired.

With these shorthands, hyp+++[A,B] is equivalent to:

```
(hyp === hyp(ConclId) +++ [_ ::: A, _ ::: B]).
```

This concrete syntax is unfolded into the lower−level syntax used by rule_add (p71).

The line must contain at least 4 and at most 60 minus characters. If there are no Seq_i sequents (N=0), then the line must be omitted.

Constraints is a comma−separated series of constraint predicates (p77) which will be evaluated when the rule is used to extend a proof. All of them must succeed (possibly by adding delayed constraints) for an application of the rule to succeed.

Examples:

```
axiom imp_reflex === A => A.
axiom imp_trans ===   A => B,
                      B => C
                    ----------
                      A => C.
axiom or_elim ===  I1:::               A or B,
                   I2::: hyp+++[A] ---> C,
                   I3::: hyp+++[B] ---> C,
                   ------------------------------
                   I :::               C
    provided
        annotate(I,cases(A or B)).
axiom all_intro ===   hyp ??? nfi(x) +++ [x:T] ---> C
                     ---------------------------------
                              (all x:T C).
axiom assump === Id::: C
    provided
        context_search(hyp, Id, _, C).
```

mode axiom(@compound).

4.9.4 declare Symbol.

Declare Symbol as a function/quantifier.

Eg 1. declare a. declares the constant a.

Eg 2. declare f(A,B). declares the function f with arity [2].

Eg 3. declare (!!q [x:_,x_2] _). declares the quantifier q with arity [quant([typed,untyped])].

mode declare(@term).

4.9.5 declare(Name,Arity).

Declare Name as a function/quantifier with the given Arity.

mode declare(@atom, @arity).

4.9.6 declare_param_const(Name, Value, Check)

Declare a set of constants called Name, parameterized by Value. Check is Prolog code that determines allowable parameter values.

Eg: declare_param_const(intconst,V,integer(V)).

mode declare_param_const(@atom, @term, @code).

4.9.7 define F === Term.

Define a new constant/function/quantifier F to be equal to Term.

Eg 1. define a === b+1.

Eg 2. define f(X,Y) === X + g(Y).

Eg 3. define (!!ex [x] B) === not (all [x] (not B)).

The rule name given by default is "def_"F, a convention which allows tactics such as unfold (p**??**) and fold (p**??**) to find the relevant definition.

mode define(@compound).

4.9.8 proof_start(Formula, Constraints, Children)

This starts a proof, with Formula as the term at the root node of the proof and Constraints as the initial set of constraints. Children is the list of unproved nodes in the new proof.

mode proof_start(?thy(term), +rule_constraint, ?list(nodeid)).

4.9.9 proof_start_postulate(Name, Children)

This starts a proof of the postulate called Name.

Name must be the name (may include parameters) of an existing postulate in the current theory.

Children will be a singleton list containing the name of the root node of the new proof. This node corresponds to the bottom sequent of the postulate. Tactics and/or a person must then build a proof tree whose only unproved nodes are the same as the upper sequents (if any) of the postulated inference rule. Once this has been achieved, the proof_complete_postulate command can be called to mark the postulate as proved.

mode proof_start_postulate(@rulename, ?list(nodeid)).

4.9.10 prove(Pred)

Starts a proof, using the Ergo 5 tree−structured prover and the 'gumtree' user interface. See Proof Commands (p12).

mode prove(?thy(term)).

4.9.11 prove_rule(Name)
prove_rule(Name, Concl)

Starts a proof of a postulated rule, using the Ergo 5 tree−structured prover and the 'gumtree' user interface. See Proof Commands (p12). The two argument form unifies Concl with the conclusion of the postulate. This is typically used to choose meaningful names for the proof variables. It should not be used to instantiate the postulate, since that would cause the proof to fail later.

mode prove_rule(+rulename).
mode prove_rule(+rulename, ?thy(term)).

4.9.12 rerun_all_postulate_proofs

Reruns all proofs for postulates in the current theory.

mode rerun_all_postulate_proofs.

4.9.13 rerun_postulate_proof(RuleName)

Reruns the proof file determined by RuleName.

RuleName must be an existing postulated inference rule. RuleName may be of the form Thy:Name, or just Name, in which case Thy defaults to the current theory. In both cases, a proof is read from the file called 'Thy/Name.proof' and the load_path (p73) is used to determine the directories in which this file may appear.

mode rerun_postulate_proof(@rulename).

4.9.14 rerun_proof(RuleName)

Reruns the proof file determined by RuleName.

RuleName may be of the form Thy:Name, or just Name, in which case Thy defaults to the current theory. In both cases, a proof is read from the file called 'Thy/Name.proof' and the load_path (p73) is used to determine the directories in which this file may appear.

mode rerun_proof(@rulename).

4.9.15 rule_add(Name, Concl, ConclId, SeqList, Constraints)

Adds an inference rule called Name to the current theory. The rule is added as a postulate. It can be proved, or marked as being a 'fundamental' rule later.

Name may be an atom or a compound term. However, in both cases the outermost functor of Name must be unique within the current theory.

Concl is the conclusion (succedent) of the bottom sequent of the inference rule and ConclId is a variable that will be instantiated to the name of the proof node that the inference rule is applied to.

SeqList must be a closed list of seq(Id,Contexts,Concl) triples. These triples are the upper sequents of the inference rule. Contexts is a list of ContextName=CSpec terms. See CSpec (p88).

Constraints is a comma−separated series of procedures which will be evaluated when the rule is used to extend a proof. All of them must succeed (possibly by adding delayed constraints) for an application of the rule to succeed.

If SeqList = [S1,...,Sn], the meaning of the rule is that any instance of Concl that satisfies Constraints can be proved by proving the corresponding instances of S1...Sn.

mode rule_add(@nonvar, @thy(term), @var, @list(sequent), @rule_constraint).

4.9.16 rule_make_fundamental(Name)

Marks the postulated rule, Name, as being a fundamental rule. Fundamental rules are rules that cannot be proven, because they form the axiomatic basis of the theory.

This command is only allowed when the current theory is open, because it extends the axiomatic basis of the theory.

mode rule_make_fundamental(@std_nonvar).

4.9.17 theorem Name === RHS

Adds a theorem or inference rule called Name to the current theory as a postulate. Depending on the value of the theorem_action (p74) parameter, some further action may take place after the postulate has been added, such as an interactive proof of the postulate or a rerun of an existing proof.

The arguments are the same as for the rule (p**??**) command. However, this 'theorem' command should be used for rules that are believed to follow from the other rules in the theory. Such rules should eventually be proven, or changed into fundamental rules using the rule_make_fundamental (p72) command.

mode theorem(@compound).

5 Parameters

Ergo has several global parameters that control various aspects of its behaviour, such as how terms are printed, how verbose the output should be etc.

Each parameter has a name and a value. The show(ParamName) (p8) command can be used to display the current value of a parameter. The set ParamName === Value (p8) command can be used to set a parameter to a new value.

The parameters are:

batch_mode (p73)
elves_type (p73)
gumtree_display (p73)
load_path (p73)
simplify_substitutions (p74)
theorem_action (p74)
verbosity (p75)
write_depth (p75)

5.1 batch_mode

This determines whether interactive questions asked via question/2 should be answered automatically or not. If batch_mode===true, all questions will automatically be answered with their default answer (which should be the 'safe' option, in some sense). If batch_mode===false, the usual interactive behaviour will result.

5.2 elves_type

This determines if the elves tactic is run before each user interaction in the Gumtree interpreter.

If elves_type === none then no elves are run.
Otherwise the elves tactic is run, provided it is defined

See jprop/elf.gum for the default definition of the elves tactic.

5.3 gumtree_display

This parameter specifies what should be displayed each time the Gumtree proof interface prompts for a command.

The parameter must be one of the following terms:

0 silent −− produces no output,

1 curr(Proc) −− calls Proc(CurrentUnprovedNodes),

2 open(Proc) −− calls Proc(AllUnprovedNodes),

Currently there is a restriction that Proc is either an atom, or the term 'application(P)'; later this will be relaxed to allow an arbitrary higher−order goal. The parameter value is saved in the global.erg file.

5.4 load_path

The 'load_path' parameter determines which directories will be searched when theories and tactic files are being loaded.

It is a list of directory specifiers. Each directory specifier must be one of the following:

***** a directory name (eg. '/homes/prover/theories'). Note that directories containing non−alphanumeric characters must be quoted. Eg. '.' means the current directory.

* a term of the form 'lib(RelPath)', where RelPath is an atom. This specifies the directory ERGOHOME/RelPath, where ERGOHOME is the 'library directory' of Ergo, which is set up during the installation process, and passed to the Ergo executable using the −lib argument. Eg. 'lib('thy/obj')'.

When the load_path is being set, the special atom '...' appearing in the new load_path list is an abbreviation for all the directories that are in the current load_path list and do not appear explicitly in the new load_path list.

An Example: If load_path is ['.','/homes/prt/thys',lib(thy)], the current directory is /homes/marku/ergo, and the Ergo installation directory is /usr/ergo, then the following directories will be searched for theory files and global tactic files:

'/homes/marku/ergo', '/homes/prt/thys', '/usr/ergo/thy'.

Theory−specific tactic files (*.ql) are expected to be within a theory subdirectory, so when trying to load a tactic file called 'tac2' in a theory called 'arith', with the above load_path, the following paths to the tactic file would be tried:

'/homes/marku/ergo/arith/tac2.ql',

'/homes/prt/thys/arith/tac2.ql',

'/usr/ergo/thy/arith/tac2.ql'.

(various alternative extensions, such as *.qo, would also be tried in each of these directories).

5.5 simplify_substitutions

The simplify_substitutions parameter determines how substitutions are displayed when theory terms are printed. If it is set to false, then all substitutions are displayed exactly where they appear in the term that is being output. If it is set to true, then substitutions are pushed into the term and evaluated as much as possible. This usually results in more readable output. The false setting is mostly used for analysing difficult problems involving substitutions.

5.6 theorem_action

This parameter allows a user−specified action to be triggered each time a theorem is added to a theory via the 'theorem' command.

If the parameter is set to 'none', then no action is performed. Otherwise, the parameter must be set to a pattern that matches a 'database' or 'general' command whose mode is one of the following:

0 Name -- no arguments

1 Name(M rulename) -- where M is @, + or ?.

2 Name(M1 rulename, M2 thy(term)) -- where M1 and M2 are each @, + or ?.

The Name command will be called immediately after the theorem has been added. The short name of the new theorem will be passed as its first argument (if required) and the conclusion of the theorem will be passed as its second argument (if required).

A typical use is to set the parameter to 'prove_rule(_,_)', so that each theorem is proved as soon as it is added. Using prove_rule/2 rather than prove_rule/1 means that the proof will use the same variable names as the original theorem.

Why not just start up a proof directly, rather than add a postulate then prove it, you ask? Well, an advantage of the postulate approach is that the resulting theorem is guaranteed to be exactly what you specify with this command, whereas an ordinary proof allows you to instantiate variables and add constraints, which may result in a less general theorem or one with the constraints expressed in a slightly different form.

5.7 verbosity

Determines how many trace messages will be displayed by tactics. Zero means no messages. Higher numbers mean more messages.

This is typically set by the user to determine how many tracing messages he/she wants to see.

5.8 write_depth

Global parameter that controls the depth of proof terms that are displayed. The default depth is 4.

6 Concepts

This section describes major Ergo concepts that are common to several commands or user interfaces.

6.1 Annotations on proof nodes

Annotations allow additional user−defined information to be associated with each proof node.

The soundness of proofs does not depend upon annotations, but they are useful for storing additional information about how the proof was constructed. Eg. tactic calls, user commands or comments.

6.1.1 annotate(PNode, Ann)

Adds the annotation(s) Ann to the proof node PNode. Ann can be a single annotation, or a list of annotations. Each annotation in Ann must have been declared using new_annotation/2 (p7).

mode annotate(@nodeid, @annotation).
mode annotate(@nodeid, @list(annotation)).

6.1.2 usercmd(CmdNum, Node, InNodes, OutNodes, Cmd)

When this annotation is attached to a proof node, it means that the user successfully executed Cmd at that proof node.

CmdNum is a positive integer that is incremented upon each successful user command, so it provides a record of the chronological order in which the commands were executed.

Node, InNodes and OutNodes give historical information about which proof nodes were affected by Cmd. They contain proof node names from the proof tree in which Cmd was originally executed. After a proof has been rerun, these node names will typically be unrelated to the node names of the current proof tree. InNodes is the list of nodes that was passed to Cmd and OutNodes is the list of nodes that was returned by Cmd. Node is the name of the proof node to which this annotation was first attached, so it is usually a member of InNodes. It is included to provide a link from the current proof tree to the node names of the original proof tree of Cmd.

Cmd is the exact character string that was entered by the user, but with :::/2 commands stripped off, because they merely determine which proof nodes the annotation is attached to. Similarly, commands that just move around the tree, or display information, are not recorded as usercmd annotations.

6.2 Constraints on rules

The following commands can be used to add constraints on the application of inference rules (and theorems).

6.2.1 =(Term1, Term2)

True if Term1 can be unified with Term2. If the two terms are unifiable, but there is no unique unifier, then delayed unification constraints will be added to the proof. For example, [A/x]B = 1 has two possible unifiers: B=1, or B=x,A=1, so this unification would be delayed.

mode =(?thy(term), ?thy(term)).

6.2.2 annotate(PNode, Ann)

Adds the annotation(s) Ann to the proof node PNode. Ann can be a single annotation, or a list of annotations. Each annotation in Ann must have been declared using new_annotation/2 (p7).

mode annotate(@nodeid, @annotation).
mode annotate(@nodeid, @list(annotation)).

6.2.3 atom(A)

True if A is an atom, like 'abc'.

mode atom(?atom).

6.2.4 context_index(CName, PNode, Index, CId)

Succeeds when the CName context of the proof node PNode contains a term with identifier CId at index Index.

mode context_index(@cname, @nodeid, ?number, ?ctermid).

6.2.5 context_nfi(ObVar, PNode)

Adds an nfi(ObVar) filter to *all* the context lists of proof node PNode. This effectively removes all terms T in those context lists that do not satisfy ObVar not_free_in T.

mode context_nfi(@obvar, @nodeid).

6.2.6 context_search(CName, PNode, CId, CTerm)

Succeeds when the CName context of the proof node PNode contains CTerm (CId is an identifier for that occurence of CTerm).

mode context_search(@cname, @nodeid, ?ctermid, ?thy(term)).

6.2.7 defined(D, E)

True if D is defined to be equal to E in an ancestor theory.

mode defined(?thy(term), ?thy(term)).

6.2.8 delay_until(Condition, Constraints)

True iff Constraints is true. Constraints can be any other sequence rule constraints. However, checking of those constraints is delayed until Condition is true. This is typically used to allow proof steps to succeed when the proof terms contain variables, but leave constraints on the future instantiations of those variables.

Here is a hypothetical example:

delay_until(ground([I,J]), (integer(I),integer(J),I<J))

checks that I<J, but delays that check until both I and J are fully instantiated to ground terms (containing no variables). This might be useful in an arithmetic theory, if you wanted to prove a theorem like param_const(intconst,I) < param_const(intconst,J), for all literal values of I and J.

The possible values of Condition are:

nonvar(X) delay until X becomes a non−variable.

ground(X) delay until X becomes ground (containing no variables)

bound(X) delay until the variable (or object variable) X is bound to any term (including another variable or object variable). (Not very useful in rule constraints)

identical_or_apart(X, Y) delay until the terms X and Y become either identical or non−unifiable.

or(A,B) delay until either of the delay conditions A or B is satisfied.

and(A,B) delay until both of the delay conditions A and B are satisfied.

mode delay_until(+nonvar, +rule_constraint).

6.2.9 integer(I)

True if I is a literal integer constant.

mode integer(?integer).

6.2.10 is_not_free_in(Obvar, Term)

True if Obvar obviously does not appear free in Term. Unlike the 'not_free_in' constraint, this constraint never adds delayed constraints, so if the not_free_in check is not decidable, this predicate will fail.

mode is_not_free_in(?thy(obvar), ?thy(term)).

6.2.11 not_free_in(Obvar, Term)

True if Obvar does not appear free in Term. If this is not decidable, delayed constraints will be added to ensure that Obvar can never appear free in Term.

mode not_free_in(?thy(obvar), ?thy(term)).

6.3 Gumtree Tactics

A tactic is a function that maps a sequence of 'm' open proof nodes to a new sequence of 'n' open proof nodes. We say such a tactic has 'arity m−−−>n'.

Suppose:
'T' is a tactic with arity 'm−−−>n';
'U' is a tactic with arity 'p−−−>q';
'R' is an inference rule with 'n' premises and one conclusion; and
'L' is a list of length k.

Primitive gumtree_tactics are:

Tactic: rule R (p23)

> **Arity:** 1−−−>n
>
> **Meaning:** map conclusion to premises.

Tactic: srule R (p23)

> **Arity:** 1−−−>n
>
> **Meaning:** map conclusion to premises.

Tactic: zero (p26)

> **Arity:** 0−−−>0
>
> **Meaning:** do nothing −− zero === [].

Tactic: one (p21)

> **Arity:** 1−−−>1
>
> **Meaning:** do nothing −− one === term(_).

Tactic: two (p25)

> **Arity:** 2−−−>2
>
> **Meaning:** do nothing −− two === [one,one].

Tactic: fix(N) (p19)

Arity: N−−−>N

Meaning: do nothing −− fix(N) === [one,...,one] N times.

Tactic: skip (p23)

Arity: N−−−>N

Meaning: do nothing −− skip === !([one,skip] | zero).

Tactic: fail (p19)

Arity: N−−−>N

Meaning: fail

Tactic: T;U (p35)

Arity: m−−−>q (n=p)

Meaning: do T then U

Tactic: T | U (p36)

Arity: m−−−>n or p−−−>q

Meaning: do T or U

Tactic: [T,U] (p35)

Arity: m+p−−−>n+q

Meaning: do T and U to parallel sets of subgoals (generalized to an arbitrary list of parallel tactics)

Tactic: !(T) (p35)

Arity: m−−−>n

Meaning: do T but discard alternatives.

Tactic: fails(T) (p36)

Arity: n−−−>n

Meaning: The tactic T would fail.

Tactic: succs(T) (p36)

Arity: n−−−>n

Meaning: The tactic T would succeed.

Tactic: if_tac(Cond, Then, Else) (p36)

Arity: n−−−>m

Meaning: If tactic Cond succeeds then remove alternatives and do tactic Then, otherwise do tactic Else. This has the same semantics as Prolog's if−then−else. The required arities are Cond : n−−−>p, Then : p −−−>m and Else : n−−−>m.

Tactic: name(N) (p20)

Arity: 1−−−>1

Meaning: The current subgoal is named N (N is a variable).

Tactic: term(T) (p24)

Arity: 1−−−>1

Meaning: The term of the current subgoal is T.

Tactic: premises(T) (p21)

Arity: N−−−>N

Meaning: The premises of the rule being proved.

Tactic: constraints(V,Dist,NFI,Other) (p17)

Arity: N−−−>N

Meaning: The constraints of the rule being proved − V is the list of variables in the rule, Dist is the list of disjointness constraints, NFI is the list of not−free−in constraints and Other is the list of other constraints.

Tactic: context(K,N,T) (p18)

Arity: 1−−−>1

Meaning: The current subgoal has a context of kind K, number N, term T.

Tactic: context_terms(T) (p19)

Arity: 1−−−>1

Meaning: The current subgoal has context terms T.

Tactic: context_index(K,I,CId) (p19)

Arity: 1−−−>1

Meaning: The current subgoal has a context of kind K, index I and ID CId.

Tactic: context_terms(T) (p19)

Arity: 1−−−>1

Meaning: The current subgoal has context terms T.

Tactic: names(L) (p20)

Arity: k−−−>k

Meaning: The first k subgoals are named L.

Tactic: terms(L) (p24)

Arity: k--->k

Meaning: The terms of the first k subgoals are L.

Tactic: annotate(L) (p16)

Arity: 1--->1

Meaning: add the annotation(s) L to the first open subproof

Tactic: {G} (p26)

Arity: N--->N

Meaning: execute Prolog goal G (must not affect proof!)

6.4 Modes and Types

The modes documentation is based on the Qu-Prolog modes. However, in addition to the Qu-Prolog meta-types, Ergo Prolog code may use synonyms defined by the add_type/3 command (eg. theory, rulename, writelist *etc.*). Use of these synonyms aids readability and may also help to hide changes to the underlying representations of such data.

To aid in documentation and to allow 'type checking' programs to be run on Prolog source code, mode declarations for every procedure should be given. These mode declarations are included in the declaration, or comment, that describes the procedure.

Each mode declaration is of the form *mode* followed by the procedure name with mode/type arguments (described below) and terminated with a '.'.

The allowable modes follow the ISO Draft Prolog standard and are summarised in the

```
mode         description
---------    -----------
-termtype    A {\em var\/} on call, at most termtype on success
+termtype    A {\em nonvar\/} on call, at most termtype on success
?termtype    A {\em term\/} on call, at most termtype on success
@termtype    A termtype on call, unchanged on success
```

table below.
The allowable types for terms are classed as follows:

Ergo5 types (p84)

6.4.1 Ergo5 types

Types specific to Ergo5 are:

anyvar Any (meta or object) variable.

atom Any Qu−Prolog name. Syntax is [a−z][A−Za−z0−9_]*, or '...'.

atomic Any atom or number.

bool must be 'true' or 'false'.

cdiff Context difference between two proofnodes.

chars Any list of chars.

cmd Something that satisfies is_legal_cmd/1. That is, an atom or a compound term whose functor is an atom.

cmdtab cmdtab is the name of a command table.

cname The name of a context list.

code Prolog code (not containing uninstantiated variables).

compound Any structured term + [].

cspec Specification of context list difference.

cterm A context 'term' (ie. an (id,term) pair).

ctermid Identity of a term in a context list.

docnode The name of a doc_node.

emptylist The empty list ([]).

filename The name of a file.

gcomp Any ground compound.

ground Any term not containing any variables.

integer Any integer.

list(Type), openlist(Type) Any list whose elements are of type Type.

nodeid Obsolete equivalent to proofnode (p86).

nonvar Any term other than variables.

number Any number.

obvar Any Qu−Prolog object variable.

proofnode Proof node identifier/name.

quant Any quantified term.

real Any real number.

rootnode Proof node identifier for root of a proof.

rulename rulename specifies a database rule. Eg. thmname, thy:thmname, thmname(args).

stream Any stream.

tactic Tactic code, possibly containing uninstantiated variables that will be instantiated by call_tac.

term Any Qu−Prolog term.

theory theory is the name of a theory. Eg. 'ergo'.

thy(Type) thy(Type) specifies some term from the 'current' theory, whose type is Type.

var Any Qu−Prolog meta variable.

writelist List of atoms and output terms for printing by writelist/1.

6.5 Proof File Properties

It is often useful to store additional information when a proof is saved into a disk file. For this reason, the read and write procedures for proof files allow a 'property list' to be saved with the proof. The property list is a closed list of PropertyName=Value pairs.

Callers of these procedures may define new 'PropertyName's, but the following PropertyName's have predefined meanings:

annotations=Bool. Annotations are saved iff Bool=true. If this property is not specified explicitly, the default is true.

rulename=Term. If this proof has been saved into a theory, then Term is the name of the resulting rule. The outermost functor of Term is usually the same as the name of the file in which this proof is saved.

premises=List. List is a list of proof node names. When the proof file is read back in again, List will be a list of variables, which will be instantiated to the corresponding node names when the proof tactic is rerun. This is usually used to record the unproved nodes of the proof.

concl=Term. Term is the conclusion of PNode. This is not essential for rerun, but is useful documentation.

constraints=Code. Code is a comma−separated list of user−supplied constraints on the proof. These are in addition to the constraints that are generated by the proof itself.

6.6 Query Language for Rules

Several commands take a 'query' argument, which specifies a rule of a set of rules. For example, the 'use' command attempts to find a rule that satisfies the query and then applies it. The 'rule' command displays all rules that satisfy the query.

A query may be a single constraint or a set of constraints (in a list or combined with and/2) that specifies a set of rules.. For example, if the query is 'imp_reflex', then all ancestor theories will be searched for a rule called 'imp_reflex'. If the query is '[jprop,iprop]:_', then all rules in the jprop and iprop theories will be tried.

The following constraints can be used within queries:

Thys:Name -- short for thys(Thys) and name(Name).

Name -- short for name(Name).

thys(Thys) -- Thys specifies the theory search list (p89) for the rule.

name(Atom) -- the name of the rule must be Atom.

name("Str") -- the name of the rule must contain Str.

frozen(Flag) -- Flag=true means freeze all variables in the rule, thus preventing instantiations.

iu_depth(Number) -- Set the incomplete unification depth (default 3).

premises(Number) -- Constrains how many premises the rule produces.

concl(Term) -- the conclusion of the rule must unify with Term.

goal -- the conclusion of the rule must unify with the goal.

context(C, T) -- T must unify with a C context entry in the conclusion of the rule.

prem(Prem) -- Prem must unify with some premise of the rule.

subterm(Term) -- some subterm of the rule must unify with Term.

6.7 Rule context specification

A 'cspec' (Context Specification) is an expression that describes the construction of a context list. The syntax of cspecs is:

```
CSpec ::= empty
        | use(CName,ProofNode)
        | add(CIdVar,Term,CSpec)
        | delete(CIdVar,Term,CSpec)
        | filter(Func,CSpec).
```

Note that each CIdVar must be a metavariable. This metavariable will be instantiated to the identity of a context term when the cspec is used to construct a context list. This typically happens whenever an inference rule is applied to a proof.

'cdiff's (Context Differences) are similar to cspecs, but are used to describe the *difference* between the context of two proof nodes in a given proof, rather than as a specification for constructing new context lists. So 'cdiff's have exactly the same syntax as cspecs, except that the CIdVar components are replaced by the actual context term identities from a current proof.

6.8 Theory Search List

One component of a query (p87) for a rule can be a ‘theory search list’. A theory search list specifies a list of theories which should be searched for a rule.

A theory search list may be a single theory name, or a list of the following terms, which specify alternative theories to search.

curr −− search the current theory.

Thy −− search the theory called Thy.

***** −− search all ancestor theories.

upto(Thy) −− search between the current theory and Thy.

i(N) −− search the theories associated with the symbols of the input term (N levels deep). For example, if the input term is
'TA.a'('TB.b'('TC.c',X),'TD.d'),
then
i(1) gives ['TA'],
i(2) gives ['TA','TB','TD'],
i(3) gives ['TA','TB','TC','TD'].

o(N) −− similar to i(N), but using the output term. currently being discharged (if any).

NB. the i(N) o(N) specs are not implemented yet.

6.9 Writelists

Most Ergo output is produced by the writelist/1,2 or msg/2,3 commands. These commands take a 'writelist' argument.

A writelist is a list of atoms and terms, to be written in turn, with the following interpretation of each element:

'nl': begin a new line, then write current indentation.

'tab': write a tab character.

tab(N): write N spaces.

w(T): write T using the Qu−Prolog write command and default operators.

q(T): write T using the Qu−Prolog writeq command and default operators.

canon(T): write T using the Qu−Prolog write_canonical command.

thy(T): write T, viewed as a term in the current theory.

thy(Thy,T): write T, viewed as a term in theory Thy.

code(T): write T, viewed as Prolog code in the current theory.

code(Thy,T): write T, viewed as Prolog code in theory Thy.

chars(T): write the chars list T (eg. ”abc”) as an unquoted string.

qchars(T): write the chars list T (eg. ”abc”) as a quoted string.

begin(Environment): begins a new Environment, within which certain commands behave specially. At present, the only defined values for Environment are ‘list’ and ‘verb’. Within a ’list’ environment the command ‘item(Tag)’ produces a line break and text for Tag. Tag may be a number, an atom, or a writelist.

end(Environment): ends the most recently started Environment.

xref(WL,Thy,Name): this writes out the atom/writelist WL and marks it as a cross reference to doc node Name in theory Thy (Thy may be ’global’). Name may also be cmd(CmdName/Arity) or ’parameter:ParamName’. WL must not contain any newlines and should not begin or end with whitespace (if whitespace is desired, it should be put into the adjacent elements of the surrounding writelist).

Typical examples are:

[’See ’,xref(’fred/2’,global,cmd(fred/2)),’.’]

[’See ’,xref(’fred/2’,global,’proof:fred/2’),’.’]

These may print out just ’See fred/2.’, or may underline ’fred/2’, or may (in a printed manual) produce something like ’See fred/2 (page 89).’

xinc(Thy, Frag(Args)): this writes the instantiated doc fragment from Thy.

wl(L): writes the writelist L.

emph(L): writes the writelist L, *emphasizing* it in some way.

verb(L): writes the writelist L in verabtim mode in some way.

left(W,L): left justify writelist(L) in a column of width W.

right(W,L): right justify writelist(L) in a column of width W.

centre(W,L): centre writelist(L) in a column of width W.

para: Write a paragraph break.

br: Write a hard line break.

Note: the q(T), canon(T), qchars(T), thy(T), thy(Thy,T), code(T) and code(Thy,T) items all write T in a way which can be parsed back in, whereas the other ways of writing terms may not.

References

Index