London Mathematical Society ISSN 1461–1570

ERGO 6: A GENERIC PROOF ENGINE THAT USES PROLOG PROOF TECHNOLOGY

MARK UTTING, PETER ROBINSON AND RAY NICKSON

Abstract

To support formal reasoning in mathematical and software engineering applications, it is desirable to have a *generic* prover that can be instantiated with a range of logics. This allows the prover to be applied to a wider variety of reasoning tasks than a fixed-logic prover. This paper describes the design principles and the architecture of the latest version of the Ergo proof engine, Ergo 6. Ergo 6 is a generic interactive theorem prover, similar to Isabelle, but with better support for proving schematic theorems with user-defined constraints, and with a different approach to handling variable scoping. A major theme of the paper is that Prolog implementation technology can be generalized to obtain efficient implementations of generic proof engines. This is demonstrated via a Qu-Prolog implementation of Ergo 6.

1. *Introduction*

Theorem provers are increasingly being used to assist with aspects of software development and hardware design [25]. In many of these applications, fully formal machine-assisted proof is essential, because the proofs are often large, and it is impractical to check such proofs by hand. Also, many of the systems being developed are proprietary or of interest only to a small community, so the usual quality-checking of proofs by peer review is not appropriate.

However, there are many different logics and theories that are being applied to software and hardware design. Theorem provers support this variety of logics in two main ways.

1. Some systems provide a prover with a single fixed logic, and encode the other logics within that logic. Some of the major interactive theorem provers that take this approach are HOL [6], PVS [16] (both based on higher-order logic) and ACL2 [12] (a successor to the well-known Boyer–Moore prover [2]).
2. Others provide a generic prover, which can be *instantiated* to work on many different logics. Examples of provers that take this approach are Isabelle [20], **mural** [9], JAPE [1] and ELF [21].

Both approaches have merit. Provers that use the first approach can provide a higher degree of automatic proof assistance, because the logic is known in advance.

In contrast, the generic provers typically provide a more direct embedding of a new logic, but cannot provide much automation until the logic is known. Instead, they typically provide a *tactic* language, which is a programming language that can be used to control the

Received 15 December 1999, revised 21 December 2001; *published* 29 November 2002.
2000 Mathematics Subject Classification 68T15
© 2002, Mark Utting, Peter Robinson and Ray Nickson

prover as it builds proofs. Thus, after instantiating the generic prover with a new logic, one can write tactics that automate common steps in that logic.

Another important factor in the design of modern theorem provers is the need for speed. Because many proofs are large, and a single tactic may perform several thousand proof steps, a prover must be fast in order to provide good interactive response times.

This paper describes a design for a generic interactive theorem prover that endeavours to maximise two (conflicting) goals: *inference speed* and *flexibility* (by which we mean allowing users to define their own custom logics, tactics, proof strategies and even side-conditions). We now expand on these two points and explain the key goals of the prover.

1.1. *Flexibility*

The primary goal of Ergo 6 is to allow an interactive user (or a tactic) to explore high-level *schematic* theorems in a flexible way, possibly even modifying a theorem as its proof is being developed. To this end, our design goals for Ergo include the following.

- Support for *schematic theorems*, with constraints on the schematic variables, including user-defined constraints.
  For example, we want to prove theorems like:

  $$(\forall x\ (A \Rightarrow B)) \iff (A \Rightarrow (\forall x\ B)).$$

  Here, A and B are schematic variables, so this theorem holds for all instantiations of A and B. In fact, the observant reader will notice that it does *not* hold for all instantiations of A, but only for those that do not contain x free. We want to be able to discover constraints like this *during* the proof, and constrain the theorem so that it becomes valid. In this case, we must add the constraint x `not_free_in` A.

- Proof and use of derived inference rules.
  For example, after defining a new operator in some theory that we are developing, we should be able to derive some basic results about that operator, and then use those results as inference rules in subsequent proofs. If the prover is to scale up to handle large proofs, it is essential that using a derived result must comprise a single proof step – it must not require the proof of the derived inference rule to be rerun. Derived inference rules are essential for scalability, and for providing high-level reasoning support for user-defined concepts. But ensuring the soundness of derived rules that include user-defined side-conditions is challenging. Note that a theorem is simply a special case of a derived inference rule: one that has no premises.

- Support for interactive proof, with users being able to focus on any part of the proof that is interesting.
  On the other hand, it is also necessary to support automated proof via tactics.

- Support for answer extraction and theorem specialization.
  Theorems can be specialized during their proof, by applying inference rules that constrain the generality of the proof. For example, in refinement proofs, we should not have to specify the desired program in advance, but should be able to extract it during the refinement process. We might start with the theorem:

  $$Sqrt \Rightarrow out^2 = in,$$

  where '$out^2 = in$' is a specification of some unknown program, $Sqrt$, and then gradually instantiate $Sqrt$ to some formula that corresponds to a square-root program.

Developing programs hand-in-hand with their proofs in this way typically results in simpler developments [4].

- Construction of an explicit record of each proof, including all the inference rules used, what tactics were applied, and user-comments about the proof.
  This allows proofs to be redisplayed in various formats, checked by independent proof-checker programs, transformed into tactics, edited and reused within other proofs.

1.2. *Inference speed*

Several decades of research into implementing logic programming languages such as Prolog [3, 24] have resulted in quite efficient implementation techniques for this restricted form of theorem proving (SLD-resolution). The prover described in this paper attempts to maximize prover speed by taking advantage of this Prolog technology as much as possible, while relaxing its restrictions.

The The Software Verification Research Centre (SVRC) at the University of Queensland, Australia, where Ergo was developed, has invested about 12 person-years into efficient implementation techniques for schematic reasoning. It has produced the Qu-Prolog language [26], an extended Prolog that incorporates this technology. We demonstrate in this paper that by using Qu-Prolog it is possible to build a prover that compares favourably with other generic schematic provers.

1.3. *Structure of the paper*

The rest of the paper is structured as follows. Section 2 describes the architecture of the Ergo 6 proof engine, and then Section 3 outlines its semantics. Section 4 describes the implementation technology that was used to implement this architecture, as well as its advantages and disadvantages. Section 5 describes the implementation of the core proof engine. Other aspects of Ergo 6, such as the implementation of the theory database, support for theory interpretation and instantiation [7, 30], details of the textual and graphical user interfaces and the Gumtree tactic language for proof construction [14] (based on ANGEL [13]), are beyond the scope of this paper.

2. *Ergo 6 proof engine architecture*

The following subsections describe the Ergo 6 proof engine in stages, starting with how object-level terms are represented, then showing how proofs are constructed, and finally explaining its support for constraints, context and annotations.

2.1. *Encoding of object-logic terms*

Ergo 6 requires all object-logic constants to be declared with an arity, and performs basic syntactic well-formedness checks, but it is otherwise untyped, with no distinction between terms and formulae.

To understand how object-level terms are encoded, it is necessary to explain the features of the meta-language, Qu-Prolog. Object-level constants, functions and predicates are encoded by meta-level (Qu-Prolog) constants; object-level variables are encoded by Qu-Prolog object variables (for example, `x`); object-level quantifiers are encoded by Qu-Prolog quantifiers (for example `all x A`), and substitutions are encoded as explicit Qu-Prolog substitution terms (for example, `[2/x]B`; the Qu-Prolog substitution notation `[T/x,U/y]B`

stands for the term `B` with all free occurrences of the object variable `y` replaced by `U` and all free occurrences of `x` replaced by `T`.). Thus the scopes of object-level variables are expressed by the Qu-Prolog quantifiers. This is similar to the *higher-order abstract syntax* approach used in many logical frameworks (for example, Isabelle [19], ELF and λProlog [21]), but with multiple meta-level binders rather than just one (λ). Another difference is that Qu-Prolog's substitution terms are used to represent object-level substitutions, whereas the above systems use *β-reduction* [21]. An advantage of the Ergo approach is that encoding an object-logic into the meta-level is often more direct and natural, but a disadvantage is that implementation of the meta-level (explicit substitutions in Qu-Prolog) is more complex.

The Qu-Prolog unification algorithm [15] extends that of standard Prolog, with syntactic equality replaced by alpha-equivalence (equality up to change of bound variable). For example, if `all` is declared as a quantifier, then unifying `(all x A)` with `[2/x]B` succeeds by instantiating `A` to `[2/x,x/y]C` and `B` to `(all y C)`, for some new `C` and `y`. Unification problems that do not have a unique solution are delayed, and are then retried when their variables are instantiated. Such unification problems can also be inspected and solved by user-defined predicates. For logical correctness, Qu-Prolog carries out 'occurs' checks during unification, but for efficiency minimises the impact of occurs checking by avoiding it when it is clearly not required.

In order to distinguish between the type of substitution discussed above, and meta-level substitution (binding of meta-variables to meta-level terms), we refer to the latter as *instantiation*. Note that instantiations 'pass through' quantified terms (no change of bound variable is required) because, at the meta-level, these quantified terms are simply structures.

Note that schematic variables (Qu-Prolog meta-variables) represent arbitrary object-level terms or predicates. When a schematic variable occurs within the scope of a quantifier, `not_free_in` constraints can be used to express holes in the scope. For example, in $(\forall x\ (A \Rightarrow B))$ we might add x `not_free_in` A.

Qu-Prolog's object variables are not standard Prolog meta-variables. Instead, they are a special class of meta-level variables that are syntactically distinguished by starting with a single `!` character, or (in this paper) a lower-case `x` or `y` (Qu-Prolog and Ergo provide commands for declaring user-defined prefixes of object variable names). They range over object-level variables of the object-logic, so cannot be instantiated to arbitrary terms, and are unifiable only with other Qu-Prolog object variables. With these restrictions, they behave almost like constants, except that they have two special properties: *substitutability* and *distinctness/unifiability*.

The *substitutability* property means that Qu-Prolog object variables can appear in the domain of substitutions (indeed, they are the only terms that are allowed in the domain of substitutions!). This provides a way of replacing free object variables by some other term. For example, the following ∃-introduction rule allows us to prove $(\exists\ y\ A)$ by proving $[T/y]A$ instead, typically by instantiating the new meta-variable T to some witness term:

```
rule ex_intro ===   [T/y]A
                   ---------
                   (ex y A).
```

The *distinctness/unifiability* property means that two Qu-Prolog object variables, `x` and `y`, may represent the *same* object-level variable, or two *different* object-level variables. Unifying `x` and `y` by writing `(x = y)` constrains them so that they represent the same

object-level variable, whereas making them *distinct* (for example, `x not_free_in y`) makes them represent different object-level variables. As long as two object variables are neither unified nor made distinct, the question as to whether they represent the same object-level variable is undecided. If an object variable is distinct from all other object variables in the proof, it is effectively a constant. For example, the constraint for the $\forall x2\ A$ rule shown later (Figure 3) forces $x2$ to be distinct from all object variables in the hypotheses during the proof of A, so that $x2$ behaves like a constant and cannot be instantiated.

2.2. *Proof construction*

Proof construction in Ergo is similar to that of the Isabelle theorem prover [17]. Isabelle is a generic successor to the LCF family of tactic-based theorem provers [18]. Experience with Isabelle has demonstrated that it is sufficiently generic to model many different reasoning styles, such as natural deduction, sequent calculi [20] and window inference [23, 27].

One of the main advances made by Isabelle was its representation of inference rules as *data*, rather than as ML functions like the LCF provers. This gives more flexibility in the way that inference rules can be used, allows a nice symmetry between forward proof and backward (or *goal-directed*) proof, and enables derived inference rules to be proved. Indeed, proof (forward or backward) in Isabelle is simply the composition of inference rules to produce derived inference rules.

Similarly, in Ergo, inference rules are data, with each inference rule containing a *conclusion* C and a sequence of *premises* $A_1, \ldots, A_n$. This is written as:

$$\frac{A_1, \ldots, A_n}{C}.$$

Proofs are trees. Each proof tree corresponds to a valid inference rule, where the root of the tree is the conclusion of the rule, and any unproved (*open*) nodes of the tree are the premises. A new proof begins with the trivially correct inference rule, T/T, and proceeds by *resolving* a given open node of the proof tree with an inference rule. It would also be possible to resolve two previously constructed proof trees together, but the current implementation supports only backward proof.

In Isabelle, proofs are an ML abstract data type, and the soundness of proofs is guaranteed by the type system of ML, because it allows proof objects to be constructed only by validity-preserving operations. In Ergo, because Qu-Prolog is untyped (like most Prolog systems), we provide this security by hiding all proof trees within a trusted module, and allowing clients to refer to the nodes of the tree only indirectly, via *node names* (integers). These node names are similar to the numbering of subgoals in the Isabelle subgoal package, except that our node names are unique across the entire proof (even if a node is deleted, its name is never reused within that proof), so that it is easier for user interfaces to track changes to the proof structure. It is occasionally useful for an inference rule to refer to the node names that will be allocated when the rule is applied to a proof tree. This can be done by prefixing the conclusion or premises of a rule by `NodeName :::`, where `NodeName` is a metavariable which will be instantiated to the corresponding node name whenever the rule is applied. (Note that most metalevel symbols in Ergo contain three repeated characters, like `:::`, `===` or `+++`. Reuse of these operators in object-level theories should be avoided, since care would be needed to avoid ambiguity.)

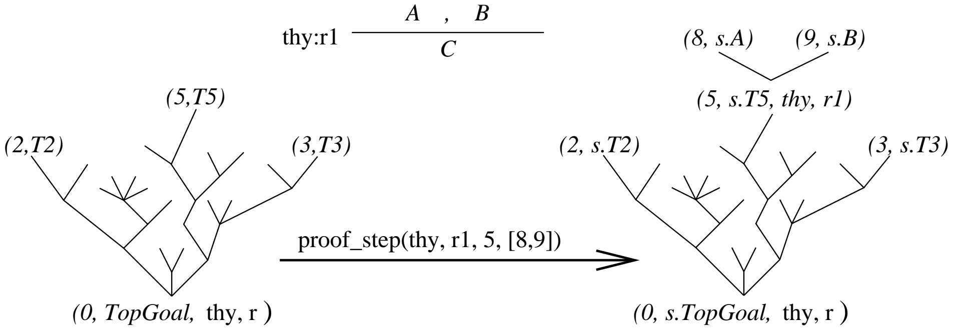

Figure 1: The effect of a single proof step.
Open nodes are represented by (*Node*, *Term*) pairs, proved nodes are represented by (*Node*, *Term*, *Thy*, *Rule*) tuples, and $s.T$ is the application of the instantiation s to the term T.

The basic operations provided by the proof module are as follows.

- `proof_start`(*Formula*, *Constraints*, *Children*).
  This creates a new proof node, and marks it as being the root node of a proof. It returns the name of this proof node in the singleton list *Children*, so that following commands can refer to the new proof node. (Note that the proof engine supports multiple proof trees, but the current user interfaces support only one proof tree.) It also executes *Constraints*, which are the initial constraints of the proof.
- `proof_step`(*Theory*, *Rule*, *Node*, *Children*).
  This resolves the inference rule *Theory* : *Rule* against the open node *Node* (see Figure 1; in Ergo, each theory has a separate name space, so the full name of a rule comprises a theory name and a rule name within that theory). This extends the proof tree with the premises of the rule, checks the rule's constraints and adds them to the proof and unifies the conclusion of the rule with the predicate of proof node *Node*. Note that this unification applies an instantiation to the entire proof, which allows schematic variables in the root of the proof to be instantiated, supporting answer extraction. *Children* is the list of new node names corresponding to the open subproofs generated by the premises of the rule.
- `proof_complete`(*Name*, *Root*, *OpenNodes*, *Constraints*).
  This checks that *Root* is the root node of a proof tree whose open nodes are exactly *OpenNodes*. It then extracts a derived inference rule from that proof tree, and adds it as a new rule (called *Name*) in the current theory. *Constraints* can be used to restrict the generality of the rule by specifying additional constraints to those implicit in the proof.

There are also several other predicates for extracting information from a given proof node, such as its parent node, its children, whether or not it has been proved, and so on.

By repeated `proof_step` commands, a goal-directed proof tree is built. At all times, the proof tree represents a valid derived inference rule. At any point during a proof, the derived inference rule defined by the proof tree can be saved into the current theory, using the `proof_complete` command. Other commands are available to convert a proof tree into a tactic, and to save a proof tree into a file for later rerunning, browsing or pretty printing.

2.3. *Rule constraints*

One of the major advances in theorem-proving technology was the introduction of schematic meta-level variables into proofs. This allows the right instantiations to be found gradually during proof [11], as in the square root example in Section 1.1. Ergo 6 extends this schematic proof style by allowing user-defined *constraints* to be placed on some of the schematic variables, to restrict their possible instantiations. This increases the expressiveness of the theorems that we can prove (just as constraint logic programming [8] is more expressive than standard logic programming).

For example, in a theory about Z schemas [29] in which signatures are ordered lists of name declarations, we might define a constraint $zsig(S)$ that checks that names in a signature are sorted, and another constraint $mergesig(S1, S2, S)$ that merges two sorted signatures to give S. This gives efficient linear-time manipulation of signatures, but still allows schematic theorems about all signatures to be proved. A similar effect can be obtained by defining the concepts of sorted signatures and merging in the object-level theory, but defining them as constraints can be more efficient and allows the proof steps to match hand proofs more closely [29]. These kinds of constraints are often referred to as *oracles*.

Ergo 6 associates a set of constraints with each proof tree, and allows constraints to be attached to inference rules. Whenever an open node of the proof is resolved with a rule, the rule's constraints are combined with the constraints of the proof. Thus the proof tree always includes the constraints of the rules that have been used to build it, and these constraints become side-conditions of the resulting inference rule or theorem.

To ensure that constraints can be managed efficiently, we require constraints to be side-effect-free Prolog predicates whose truth depends only upon their arguments. Whenever a proof step instantiates a variable, the constraints that involve that variable may be rechecked. The check may return either '*false*', in which case the proof step fails, or a new set of constraints, which replace the original constraint. The new set of constraints may contain just the original constraint unchanged, or some simplified constraints, or may be empty (which means that the original constraint has been fully satisfied). For soundness, constraints must also satisfy a weak commutativity property with respect to instantiation. That is, given a constraint $c(A)$ with a parameter A, the following condition must hold for all terms t (comma is left-to-right Prolog conjunction here):

$$c(A), A = t \quad \Rightarrow \quad A = t, c(A).$$

For example, $\texttt{nonvar}(A)$ is a legal constraint, but $\texttt{var}(A)$ is not, because $\texttt{var}(A)$, $A = 1$ succeeds, while $A = 1$, $\texttt{var}(A)$ fails. This property ensures that it is always sound to delay the checking of constraints until later in the proof. In fact, checking a constraint too early (before its arguments are instantiated) may cause it to fail – this reduces completeness, but not soundness.

A third restriction on constraints is that when all the arguments of a constraint are fully instantiated terms, then the constraint must evaluate to either false or true (that is, it must either fail, or return the empty set of constraints). In standard Prolog terminology, such fully instantiated terms are called *ground* terms. We also require all pairs of object variables in the terms to be either identical or distinct. Thus a fully instantiated term corresponds to exactly one term of the object logic.

The kinds of constraints needed to express rules vary between logics – we provide some support for logic designers to define new constraints. Several standard constraints are supplied: $\texttt{nonvar}(T)$; $\texttt{ground}(T)$; $\texttt{delay_until}(\textit{Cond}, \textit{Constr})$, which delays

the checking of constraint *Constr* until condition *Cond* (an instantiation condition such as `nonvar`(T) or `ground`(T)) is satisfied; and x `not_free_in` T, which ensures that object variable x cannot appear free in term T.

Other examples of the use of constraints are to implement arithmetic of constants using Qu-Prolog or infinite-precision integers, and to define the properties of a class of constants, such as a lexicographical ordering relation.

2.4. *Special support for context*

Since many logics have some notion of *context*, it is useful to support context directly in logical frameworks [21]. This section briefly describes the theory designer's view of how context is supported by Ergo. An efficient implementation of these context facilities is discussed in Section 5.

The logical framework of Ergo is based on sequent calculus, with each proof node conceptually containing a sequent, $C \longrightarrow F$, where C is the *context* of the node and F is a formula of the current object-logic. In sequent calculi based on Gentzen's LJ calculus [5], C is simply a list of formulae. Ergo extends this by allowing each node of a proof tree to contain *several* named contexts, each of which is a list of uniquely labelled terms. Multiple contexts are useful when a logic requires that different kinds of contexts be manipulated in different ways. For example, an Ergo implementation of the Z_c logic for Z schemas used one context for typing information and another for general assumptions. This allows the inference rules that discard all general assumptions to execute in constant time, rather than in a time proportional to the number of assumptions [29].

As is the case for the normal symbols of a logic, the meaning of contexts is embodied entirely in the inference rules. To support goal-directed proof efficiently, inference rules specify the contexts of their premises by describing how they differ from the context of their conclusion. The full syntax of each premise of an inference rule is:

$$NAME \; \texttt{:::} \; (n_1 \; \texttt{===} \; s_1 \; \texttt{\&\&\&} \ldots \texttt{\&\&\&} \; n_k \; \texttt{===} \; s_k) \; \texttt{--->} \; F \; .$$

where each n_i is the name of a context list and each s_i is a *context specification*.

The following kinds of context specifications are supported.

- *Initialise*. Set the context list to be empty.
- *Add*. Add a term to the context list.
- *Delete*. Delete a term from the context list.
- *Filter*. Apply a filter function to the context list. The only predefined filter is `nfi`(x), which succeeds on terms that do not contain free occurrences of the variable x (for schematic terms, it adds constraints to ensure that those terms cannot be instantiated to contain a free x).

As examples, Figure 2 shows three rules from the Ergo standard theory library, in Ergo syntax. These rules are part of an LJ-like logic that uses the context named `hyp` to represent antecedents. The first is an $\vee$-elimination rule, the second is a form of $\exists$-elimination, and the third is the assumption axiom.

In the first rule, the first premise has the same context as the conclusion, while the second and third premises are each augmented with a new hypothesis.

In the second rule, the existentially quantified term is first removed from the hypotheses, then the remaining hypotheses are filtered with a not-free-in constraint, and then the new

```
rule or_elim ===
                          A or B,
    hyp+++[A]   --->   C,
    hyp+++[B]   --->   C
    ------------------------
                          C.

theorem ex_elim_c(x,A) ===
    hyp --- [ex x A] ??? nfi(x) +++ [A] ---> C
    ------------------------------------------
                          C
  provided
    x not_free_in C.

rule assump(ID) ===
     Node:::C
  provided
     context_search(hyp, Node, ID, C).
```

Figure 2: Example rules from the Ergo standard theories.

hypothesis is added. The conclusion of the rule is also constrained with the appropriate not-free-in constraint.

In the third rule, the built-in constraint `context_search` is used to search for the conclusion in the hypotheses. `Node` is the identifier of the conclusion, and `ID` is the identifier of the corresponding hypothesis (if `ID` is not specified when the `assump` rule is applied, then any context term that unifies with the conclusion may be used).

2.5. *Annotations*

Finally, we describe user-defined *annotations*, which may be attached to any node of the proof tree. Although annotations do not affect the soundness of proofs, it is useful to provide support for them in the core prover so that operations that modify the proof can preserve the annotations that are attached to proof nodes. For example, the `proof_tactic` command that converts a proof tree into a tactic offers the option of recording and regenerating annotations. Annotations are useful for attaching to nodes and subtrees of the proof [**10**], such higher-level information as: tactic invocations, user commands, comments describing proof strategies and rationales, user interface information, and instructions for the presentation of printed proofs.

For example, one of the Ergo user interfaces uses annotations to record the exact text of each successful proof-construction command entered by the user, including comments and whitespace. These annotations are stored as part of the proof record, and are used to recreate the original proof script when required. This proof script can be used as a basis for reuse by editing, if the proof fails to rerun in a modified theory at some later date.

3. *A meta-logic for Ergo*

Since Ergo is a generic prover, it is important that it have a clear semantics, so that object-logic designers can check that they have correctly encoded an object logic. This section describes a meta-logic for Ergo.

Because Ergo is implemented in Qu-Prolog, it relies heavily on the unification algorithm of Qu-Prolog and the built-in not-free-in constraints that relate object-variables to Qu-Prolog terms. For this reason, we have chosen to explain the semantics of the meta-logic in the same way as is used to prove the correctness of the Qu-Prolog unification algorithm [15]. In fact, we consider the meta-logic for Ergo to be an extension of the logic used in the Qu-Prolog unification paper. In other words, the Ergo meta-logic includes all the constants and theorems of that paper.

The semantics is based on functions that map terms (and formulae) at the meta-level to terms (and formulae) at the object-level via a *meaning function*, m, that maps constants at the meta-level to constants at the object-level, and *variable valuations*, v, that map variables at the meta-level to terms at the object-level. For a fixed meaning function m and each variable valuation v, we define the function m_v^* that maps terms at the meta-level to terms at the object-level. The definition for m_v^* is given in [15] for abstract Qu-Prolog terms, and follows the recursive structure of terms in a straightforward manner. The value of each constant is determined by m, and the value for each meta-level variable is determined by v.

Note that Qu-Prolog object-variables map to variables under v (since object-variables range over variables at the object-level), and substitution terms and not-free-in constraints at the meta-level can be fully evaluated after mapping them to the object level. This is because, at the object-level, we have complete information about occurrences of variables in terms.

3.1. *Formulae and terms of the meta-logic*

The formulae of the meta-logic can be divided into four categories:

1. formulae representing the valid rules of Ergo;
2. constraint predicates that describe when rules are applicable;
3. formulae for processing structural subterms of rules, such as premise lists; and
4. formulae constructed from other formulae using logical connectives ($\wedge$, $\Rightarrow$, $\neg$ and *true*).

We now discuss each of the first three categories in more detail.

The valid rules of Ergo include all the fundamental inference rules supplied by a theory designer, plus all the derived inference rules that can be proved using Ergo. The meta-logic formula representing such a rule is called an *Ergo-rule*, and has the form *rule(Concl*, *Prems*, *Constr*) where *Concl* is the conclusion of the rule, *Prems* is a list of premises and *Constr* is a conjunction of constraints representing the 'side conditions' of the rule.

Proofs in Ergo are explicitly represented by their proof trees. For the meta-logic, however, we take a similar approach to that of Isabelle, and consider each step of an Ergo proof as a proof of an Ergo-rule, with the conclusion of the rule being the root of the proof tree and the premises being the list of open (unproven) nodes.

The conclusion and each premise of an Ergo-rule is a generalised sequent represented by a meta-logic term of the form *seq*(*Succ*, *Contexts*), where *Succ* is the succedent of the generalised sequent (a theory-specific term), and *Contexts* is a list of contexts. Each context is a term of the form *context*(*Name*, *CSpec*), where *Name* is an atom that names the context and *CSpec* is a term representing the context specification of the context. The terms used to represent context specifications are of the form

$$nil, \quad add(ID, Formula, CSpec), \quad del(ID, CSpec), \quad nfi(ObjVar, CSpec),$$

where *nil* represents the empty context, *add* and *del* respectively add and remove formulae from a context, and *nfi* is used to add a not-free-in constraint to variables in a context. *ID* is an identifier for the context formula that is added/deleted.

In the implementation, for efficiency reasons, context is managed by reference to node identifiers and context names. If a context is not mentioned in the rule, then it is unchanged by the application of the rule. In order to simplify the discussion of the meta-logic, we take the view that each context list in each sequent is a list of named contexts, with each possible context name of the theory appearing exactly once. Furthermore, to simplify the unification of sequents we assume that the named contexts appear in the same order in each context list.

Each context specification of the conclusion of each Ergo-rule is typically a variable that is instantiated to the corresponding context specification of the sequent to which the rule is applied. Rules that access context will have a context-search constraint on one or more of these variables. The context specifications of each premise are modifications of the corresponding context specifications of the conclusion.

The three rules of Section 2.4 are represented in the meta-logic as follows:

- the `or_elim` rule:

$$\begin{aligned}&\mathit{rule}(\mathit{seq}(C, [\mathit{context}(\mathit{hyp}, H)]),\\&\quad [\mathit{seq}(A \; \mathit{or} \; B, [\mathit{context}(\mathit{hyp}, H)]),\\&\quad\; \mathit{seq}(C, [\mathit{context}(\mathit{hyp}, \mathit{add}(\mathit{ID}1, A, H))]),\\&\quad\; \mathit{seq}(C, [\mathit{context}(\mathit{hyp}, \mathit{add}(\mathit{ID}2, B, H))])],\\&\quad \mathit{true});\end{aligned}$$

- the `ex_elim_c` rule:

$$\begin{aligned}&\mathit{rule}(\mathit{seq}(C, [\mathit{context}(\mathit{hyp}, H)]),\\&\quad [\mathit{seq}(C, [\mathit{context}(\mathit{hyp}, \mathit{add}(\mathit{ID}1, A, \mathit{nfi}(x, \mathit{del}(\mathit{ID}2, \mathit{ex}\ x\ A, H))))])],\\&\quad x\ \mathit{not_free_in}\ C \wedge \mathit{context_search}(\mathit{ID}2, \mathit{ex}\ x\ A, H));\end{aligned}$$

- the `assump` rule:

$$\mathit{rule}(\mathit{seq}(C, [\mathit{context}(\mathit{hyp}, H)]), [], \mathit{context_search}(\mathit{ID}, C, H)).$$

The constraint part of an Ergo-rule is used to describe when the rule is applicable. It is a conjunction of constraint formulae, which comprise the second category of meta-logic formulae, and have the form

$$\begin{aligned}&\mathit{ObjVar}\ \mathit{not_free_in}\ \mathit{Term},\\&\mathit{Term}_1 = \mathit{Term}_2,\\&\mathit{context_search}(\mathit{ID}, \mathit{Formula}, \mathit{CSpec}).\end{aligned}$$

These represent not-free-in constraints, unification constraints, and context-searching constraints, respectively, which are the main built-in constraints of Ergo. Note that each context search constraint of the meta-logic refers to the context specification of the relevant context directly rather than through its name, as is done in the implementation.

Users designing logics for Ergo can also define their own oracles as constraints on rules, and these become additional meta-level constraint formulae. For example, a theory about arithmetic includes a rule with the constraint

```
delay_until(ground(Exp), arith_oracle(Exp, Result))
```

for evaluating ground expressions. The correctness of the rule depends on the correctness of the code that implements the oracle.

We note that these constraints are Qu-Prolog goals, and are called when encountered. On the other hand, constraints can, in principle, be generated and solved in any order. A consequence of this is that, in order to preserve correctness, constraints cannot rely on any non-logical features of Qu-Prolog. More precisely, user-defined constraints must satisfy the three properties described in Section 2.3: they must be side-effect-free Qu-Prolog predicates whose truth depends only upon their arguments, they must satisfy the weak commutativity property, and they must be decidable when applied to fully instantiated terms. The built-in constraints satisfy these requirements, and it is the theory designer's responsibility to make sure that user-defined constraints also do so. Thanks to the delay, the `arith_oracle` constraint described above also satisfies this condition.

The third category of meta-level formulae is concerned with the change to a proof tree (as represented by an Ergo-rule) when a rule is applied. The only formula in this group is $replace(T, L_1, L_2, L_3)$, which describes how the list of premises (open nodes of the proof tree), L_2, becomes the new list L_3 by replacing the element T (the conclusion of the rule) by the list L_1 (the premises of the rule).

3.2. *Semantics of the meta-logic*

Our main objective is to describe what `rule` formulae are theorems of the meta-logic, and to write down axioms that allow us to deduce new `rule` theorems. We first extend the meaning function m to the constants of the meta-logic (such as `seq`, `context`, list constructors, $\wedge$ and $\Rightarrow$) in the obvious way. Note that any constraint C evaluates to true or false when applied to fully instantiated terms, so $m_v^*(C)$ is either true or false. Next, we say what it means for a formula of the meta-logic to be a theorem.

Definition 1. A formula, T, of the meta-logic is a *theorem* if and only if $m_v^*(T)$ is true for all variable valuations v.

We note that, as in [15], the $\Rightarrow$, $\wedge$ and $\neg$ connectives are the pointwise lifted versions of the usual propositional connectives, and a consequence of this is that the meta-logic is an instance of a standard propositional calculus in which, for example, we can carry out Hilbert-style reasoning.

Definition 2. The meta-logic formula, $replace(T, L_1, L_2, L_3)$, is true in a valuation v if and only if there are some L_a and L_b (at the object-level) such that

$$m_v^*(L_2) = L_a \frown [m_v^*(T)] \frown L_b m_v^*(L_3) = L_a \frown m_v^*(L_1) \frown L_b,$$

where '$\frown$' is list concatenation.

For defining the meaning of *rule*(_, _, _), it is convenient to be able to extract the constraints from the rest of the rule, so we first define a variant of *rule* without constraints.

Definition 3. The meta-logic formula, $rule(Concl, Prems)$, is true in a valuation v if and only if

$$\frac{m_v^*(Prems)}{m_v^*(Concl)}$$

is a valid object-level inference rule.

Definition 4. $rule(Concl, Prems, Constrs)$ is defined as $Constrs \Rightarrow rule(Concl, Prems)$.

The validity of an Ergo-rule follows directly from these definitions and the definition of theoremhood.

COROLLARY 1. *The rule rule(Concl, Prems, Constrs) is a theorem if and only if*

$$\frac{m_v^*(\mathit{Prems})}{m_v^*(\mathit{Concl})}$$

is a valid object-level inference rule for all valuations v such that $m_v^*(\mathit{Constrs})$ is true.

It is the responsibility of the designer of each Ergo theory to check that the fundamental inference rules of the theory represent valid object-level inference rules. The meta-logic formulae that represent fundamental inference rules of the Ergo theory are axioms of the meta-logic.

Next we present the axioms of the meta-logic used to derive new Ergo-rules. We use the semantics to justify the soundness of these axioms. It then follows that any derived inference rule proved in Ergo is valid, since Ergo is an implementation of this meta-logic.

3.3. *Axioms of the meta-logic*

For a given Ergo theory, the meta-logic contains axioms that represent the fundamental inference rules of the Ergo theory, and also the general axioms that state how Ergo derives new inference rules for the theory. We now consider the latter set, which are applicable to all theories.

The following axiom describes how two rules can be joined together to produce a new rule. This axiom is justified by noting that the semantics of `replace` describes how the proof tree described by the first rule is modified by the application of the second rule to produce the proof tree described by the third rule.

AXIOM 1.

$$\begin{array}{l} \mathit{rule}(\mathit{Root}, \mathit{Prems}) \wedge \mathit{rule}(\mathit{RC}, \mathit{RP}) \wedge \mathit{replace}(\mathit{RC}, \mathit{RP}, \mathit{Prems}, \mathit{NPrems}) \\ \Rightarrow\ \mathit{rule}(\mathit{Root}, \mathit{NPrems}). \end{array}$$

The following theorem justifies the mechanism for rule application in Ergo. This corresponds to the `proof_step` command introduced in Section 2.2.

THEOREM 1 (APPLY RULE).

$$\begin{array}{l} \mathit{rule}(\mathit{Root}, \mathit{Prems}, \mathit{Constraints}) \wedge \\ \mathit{rule}(\mathit{RConcl}, \mathit{RPrems}, \mathit{RConstraints}) \wedge \\ \mathit{replace}(\mathit{Concl}, \mathit{RPrems}, \mathit{Prems}, \mathit{NPrems}) \wedge \\ \Rightarrow \\ \mathit{rule}(\mathit{Root}, \mathit{NPrems}, \mathit{Constraints} \wedge \mathit{RConstraints} \wedge (\mathit{Concl} = \mathit{RConcl})). \end{array}$$

The first rule formula represents the current state of the proof, while the second rule formula represents the rule being applied.

Proof. We proceed by straightforward propositional reasoning. First, it is enough to assume the antecedents, and to show that

$$\mathit{rule}(\mathit{Root}, \mathit{NPrems}, \mathit{Constraints} \wedge \mathit{RConstraints} \wedge (\mathit{Concl} = \mathit{RConcl}))\ .$$

Secondly, by using Definition 4, we can assume that

$$\mathit{Constraints} \wedge \mathit{RConstraints} \wedge \mathit{Concl} = \mathit{RConcl},$$

and show *rule*(*Root*, *NPrems*). This follows using Axiom 1, replacing *RConcl* by *Concl* in the second assumption (which is justified by the equality assumption). □

Next we look at a theorem that allows us to strengthen or simplify constraints.

THEOREM 2 (TRANSFORM CONSTRAINTS).

$$\begin{aligned}&rule(Root, Premises, Constraints_1) \wedge \\ &(Constraints_2 \Rightarrow Constraints_1) \\ &\Rightarrow \\ &rule(Root, Premises, Constraints_2).\end{aligned}$$

Proof. This is achieved by propositional reasoning and Definition 4. □

An example of strengthening constraints in Ergo occurs when we have a delayed unification constraint that has more than one solution. Such constraints are typically strengthened by choosing one of the solutions. This is equivalent to adding new unification constraints and new not-free-in constraints. For simplifying constraints, we are typically using some theorem of the form

$$Constraints_1 \iff Constraints_2$$

which can be used in conjunction with the above theorem to apply the simplification. Simplifications of unification or not-free-in constraints are examples of the above.

The next theorem tells us that the use of Qu-Prolog's variable instantiation preserves the correctness of rules. Note that the application of an instantiation, σ, to a term T, written σT, gives the term obtained from T by replacing each variable in the domain of σ by its corresponding instantiation.

THEOREM 3 (INSTANTIATE VARIABLE).

$$\begin{aligned}&rule(Root, Premises, Constraints) \wedge \\ &(Constraints \Rightarrow X = T) \\ &\Rightarrow \\ &rule(\sigma Root, \sigma Premises, \sigma Constraints),\end{aligned}$$

where X is a variable and T is a term not containing X, or X and T are both object-variables. Here, σ is the instantiation that instantiates X to T.

Proof. This theorem follows from theorems about Qu-Prolog unification. □

Finally, we state the axioms that give the semantics of searching contexts.

AXIOM 2 (CONTEXT HEAD).

$$F_1 = F_2 \Rightarrow context_search(ID, F_1, add(ID, F_2, Context)).$$

AXIOM 3 (CONTEXT TAIL).

$$\begin{aligned}&context_search(ID, F_1, Context) \\ &\Rightarrow context_search(ID, F_1, add(ID_2, F_2, Context)).\end{aligned}$$

AXIOM 4 (CONTEXT DELETE).

$$\begin{aligned}&context_search(ID_1, F, Context) \wedge \neg(ID_1 = ID_2) \\ &\Rightarrow context_search(ID_1, F, del(ID_2, Context)).\end{aligned}$$

AXIOM 5 (CONTEXT NFI).

$$context_search(ID, F, Context) \wedge ObjVar\ not_free_in\ F \Rightarrow context_search(ID, F, nfi(ObjVar, Context)).$$

The Ergo implementation of context searching has the above semantics.

4. *Implementation technology*

Proof construction in the proposed architecture, as in Isabelle, has many similarities to Prolog. Indeed, Paulson shows that, with a depth-first control tactic, Isabelle is a higher-order Prolog interpreter [17]: inference rules are Horn clauses, usually containing schematic metavariables; inference rules are composed using resolution, with the conclusion of one rule unifying against a premise of another rule to produce the compound rule; and higher-order unification is used to handle quantifiers and unknown function variables.

This section discusses the degree to which it is possible to implement the Ergo proof engine using Prolog resolution, and the advantages and disadvantages of doing this. This is not just a matter of using Prolog as the implementation language for the proof engine. Rather, can we use the Prolog proof engine (SLD-resolution) to implement the Ergo proof engine directly?

4.1. *Advantages and disadvantages of using Prolog*

The main advantage of using the Prolog resolution mechanism directly is speed. Stickel identifies three main aspects of Prolog implementation technology that make it a highly efficient theorem prover [28].

1. *Efficient representation of derived clauses.* The Prolog SLD-resolution mechanism (*input resolution with depth-first search*) means that the current state of a Prolog proof (the *derived clause*) is simply the set of goals that remain to be proved. Since Prolog uses depth-first search, this derived clause can be represented by a *stack* of goals, which makes backtracking very efficient.
2. *Compilation of clauses.* Prolog uses *input* resolution, where one of the clauses being resolved is always derived from the original query and the other comes from a database of clauses. Since the clause database is relatively fixed throughout a proof (except for asserting new theorems/lemmas), its clauses can be compiled into efficient code (either interpreted bytecodes or machine code) that is customized according to the structure of each clause. In contrast, compilation is not so effective for other forms of resolution, because there is no fixed set of clauses to compile.
3. *Efficient representation of variable instantiations.* Prolog unifies two terms by destructively updating their variables to satisfy the unifying instantiation. The destructive updates are recorded on a stack so that they can be undone upon backtracking. This scheme is simple, and highly efficient for depth-first search. Alternative schemes are much less efficient: they typically require a new term to be created for the result of each unification (lots of copying), or they use *structure sharing* where the result of each unification is a (*instantiation*, *term*) pair (processing of such results is more complex than in the Prolog scheme).

For comparison, note that Isabelle uses multiple proof trees represented by ML data structures, so none of the above advantages apply. The derived clauses (proof trees) are

in the ML heap, so must be garbage collected; they cannot be compiled; and unification creates new terms.

By modifying Prolog systems in ways that preserve the above three properties, Stickel built high-speed theorem provers that are capable of thousands of inference steps per second. The work described in this paper can be viewed as an extension of his approach, to allow even more flexible styles of proof, while retaining most of the advantages of the Prolog technology.

Disadvantages of using Prolog to directly implement Ergo include the following.

1. We want to produce an explicit record of the whole proof, whereas the Prolog stack-based representation of the proof tree discards each subtree of the proof as soon as it is completed.
2. We want to be able to construct incomplete proofs (for example, derived inference rules), whereas Prolog always either completes an entire proof, or fails.
3. Prolog uses a fixed (and incomplete) search strategy (depth-first, left-first), whereas we want to give the tactics and/or the user control over the search strategy.
4. Prolog uses quantifier-free first-order unification, whereas we want to unify terms involving quantifiers and higher-order terms.
5. Most Prolog implementations omit occurs checks, but occurs checks are necessary for avoiding faulty inferences in theorem proving [28].

We avoid the last two difficulties by using Qu-Prolog, which supports quantifiers, delays difficult unifications (and allows user-defined heuristics to solve such delayed problems) and does include occurs checks. The next subsection discusses how the first three problems can be solved.

4.2. *A design that uses Prolog*

This section develops a design for a prover that has much of the proof efficiency of Prolog, yet builds explicit proof trees (possibly incomplete trees) and supports user-defined search strategies. The actual Ergo 6 implementation (Section 5) is similar to this design, but with more sophisticated data structures, and with support for context lists.

The key insight is that we can transform each inference rule into a Prolog clause that not only performs the desired resolution step, but also extends a data structure that represents the explicit proof tree.

For example, the obvious Prolog translation of a conjunction introduction rule is:

```
prove1(A and B)  :-  prove1(A), prove1(B).
```

We modify this Prolog clause so that it builds an explicit proof tree. We add an extra parameter that builds a structure to record the term being proved and the name of the rule used to prove it. This structure has the form `node(`*Thy*, *Rule*, *Term*, *SubTrees*`)`, where *Term* is the term that is being proved, *Thy* : *Rule* records the theory and name of the inference rule, and *SubTrees* is a list of the subtrees (also `node/4` structures) that result from applying that inference rule. In unproved nodes, *Thy*, *Rule* and *SubTrees* are variables. The clause representation of the `and_intro` rule is now:

```
prove(node(fol, and_intro, A and B, [Sub1,Sub2]))
    Sub1 = node(_, _, A, _),
    Sub2 = node(_, _, B, _),
    prove(Sub1),
    prove(Sub2).
```

The two recursive calls to `prove` in this clause correspond to the Prolog depth-first search strategy, so we remove them to allow more flexible search strategies. This reduces our `prove` predicate to just a table of inference rules. However, we still retain much of the Prolog feature of *clause compilation*, because each clause in the table is fixed (even though new rules can be added to the table), and can thus be compiled into efficient code that is optimized for the particular structure of that inference rule.

A proof is basically just a sequence of calls to this table. Each call proves one node of the tree, by instantiating its *Thy*, *Rule* and *SubTrees* components.

It is simple to write a recursive tactic that implements the original Prolog depth-first search. For interactive proof, we can define a `proof_step` command, similar to the one described in Section 2. (The only difference is that this one passes an entire proof node data structure as the third argument, whereas the real Ergo `proof_step` command passes just the *name* of that proof node.) A sequence of calls to this `proof_step` predicate builds a goal-directed proof tree.

```
proof_step(Thy, Rule, node(Term,Thy,Rule,SubProofs), SubProofs) :-
    prove(node(Thy, Rule, Term, SubProofs)).
```

4.3. *Analysis of this design*

This style of using Prolog to build proof trees overcomes the first three disadvantages of standard Prolog mentioned above. It creates explicit proof trees, and allows incomplete proofs and flexible search strategies. It also retains the Prolog efficiency advantages.

1. *Efficient representation of derived clauses.* In our architecture, a proof is represented by a tree data structure, rather than a stack like the derived clause of Prolog. However, this proof tree data structure is built within the Prolog heap, which is organised as a stack. Furthermore, the sequence of proof steps that builds a proof tree is recorded in a stack-like fashion on the environment stack of the underlying Prolog. This means that when the underlying Prolog backtracks, proof steps are undone in reverse chronological order. This gives us a chronological 'undo' facility for free at the user interface level.
   It also allows us to write efficient Prolog tactics that call `proof_step` using various search strategies, and automatically undo those proof steps upon failure. If failing did *not* undo proof steps automatically, tactics would have to contain explicit code to undo unwanted steps, and our experience with earlier versions of Ergo showed that this made it far harder to write searching tactics.
2. *Compilation of clauses.* The `proof_step` command performs a backward proof step using input resolution, where an open subproof of the proof tree is resolved against an inference rule from a (relatively) fixed theory database. Thus mature Prolog compilation technology can be used to compile the inference rules. This efficiency advantage is the main reason for preferring the goal-directed `proof_step` command over a more general command that resolves two proof trees together.
3. *Efficient representation of variable instantiations.* Our proof engine uses the underlying Prolog representation of variables and variable instantions directly. Schematic proof variables are simply Prolog variables (meta-level variables in logical and in Qu-Prolog [26] terminology). They are instantiated by Prolog unification (for example, when the conclusion of an inference rule is unified with a proof node). This is convenient and efficient for depth-first searching tactics and for chronological 'undo'

operations. However, it does make it harder to support non-chronological undo operations.

For instance, the implementation of the architecture described in Section 5 allows subtrees of a proof to be discarded in any order. However, variable instantiations that were made by the discarded proof steps can be undone only in reverse chronological order. To implement more sophisticated undo operations, it would be necessary to undo a larger portion of the proof, and then rebuild some of its subproofs.

So, in using the underlying Prolog representation of variable instantiations we have followed the RISC principle: make common operations (goal-directed proof steps, chronological undo operations and tactic execution) fast, while less common operations (non-chronological undo operations) are slower.

5. *An efficient implementation*

This section sketches the main features and data structures of our implementation of Ergo 6. The main differences from the simplified design in the previous section are that proof trees are encoded into hash tables, and that context is manipulated more efficiently.

Qu-Prolog provides a set of global variables called *implicit parameters* [22], with a backtrackable assignment operation and clean logical semantics. It also provides *indexed* implicit parameters, which provide the functionality of hash tables. The assignment command `ip_set(`*IP*`,`*Index*`,`*Value*`)`, is equivalent to *IP*(*Index*) `:=` *Value*, but is undone upon backtracking. Similarly, `ip_lookup(`*IP*`,`*Index*`,`*Value*`)` is used to query an indexed implicit parameter. It succeeds whenever *IP*(*Index*) unifies with *Value*.

5.1. *Proof tree data structures*

The simple design outlined in Section 4 represented an entire proof tree as a single Prolog term. The actual implementation uses a more sophisticated representation.

Proof nodes are stored in an *indexed* implicit parameter, called `node`. In fact, a more obscure name is used. Since Qu-Prolog does not yet provide any module or ADT constructs, the encapsulation of the proof data structures relies on choosing names for these implicit parameters that are unlikely to clash with names used by tactics. This does not protect against malicious attempts to corrupt the proof data structure, but provides good protection against accidental corruption. This has two advantages over the single-term approach.

- It gives us constant-time access to every node of the tree.
- It allows us to destructively update any node of the tree in constant time, using backtrackable assignment. This allows proof steps to be undone in non-chronological order, except for variable instantiations.

Each proof node is allocated a unique *name* (an integer) as discussed in Section 2. Each node of the proof is represented by a term of the form:

`proofnode(`*Parent*, *Term*, *Theory*, *Rule*, *Children*`)`.

If a node is open (unproved), then the *Theory*, *Rule* and *Children* fields contain variables, otherwise *Theory* : *Rule* is the name of the inference rule that has been resolved with this node and *Children* is a list of node names corresponding to the subproofs generated by the premises of the rule. We can now describe the implementation of the main proof engine commands.

The `proof_start/3` command allocates a new node name, `RootID`, and then creates the root node of the proof by executing

```
ip_set(node, RootID, proofnode(root, Formula, _, _, _)).
```

The implementation of the `proof_step` command is

```
proof_step(Thy, Rule, Id, Children) :-
    ip_lookup(node, Id, proofnode(_,Term,Thy,Rule,Children)),
    rule_apply(Thy, Rule, Term, Id, Children),
    proof_non_circular(Thy, Rule).
```

The `rule_apply/5` predicate looks up `Thy : Rule` in the theory database (similar to the `prove4` table in Section 4.2) and `proof_non_circular/2` checks that no circularities arise involving `Thy : Rule` and the current proof, by checking that `Thy : Rule` does not appear in a precalculated table of rules.

5.2. *Context list data structures*

The data structures used for context lists are designed to meet the following objectives.

- They achieve a high degree of structure sharing (to save space, and to save time by avoiding copying when creating new proof nodes). Structure sharing may occur even between context lists that have different names.
- They allow efficient updating of any point in any context list.
- They allow changes to the context list at a given proof node to be immediately visible at all nodes of its subtree.
- They automatically undo changes on backtracking.
- They allow the differences between the context of a conclusion and a premise to be reported symbolically, as a context specification. (This allows proofs to be transformed into inference rules.)

Our current implementation never copies context lists. If a rule applies a filter function (for example, `nfi(x)`) to a given context list, that context list is not immediately filtered to obtain a new context list. Instead, the function is recorded at that point in the context list, and then whenever an access of the context list returns a term T from deeper in the context, it applies the function to it. In other words, the application of filter functions is done lazily at lookup time, rather than at rule-application time. It also means that filter functions are not applied to context terms that are never used.

Advantages of this approach are that it allows rule application to be very fast because contexts are not copied, and it allows updates of the parent's context list to flow into the filtered list automatically. A disadvantage is that if a given term in a context list is frequently accessed, then filter functions may be applied to that term repeatedly (once every access). Experiments suggest that this does not cause performance problems, but if it does in the future, then we may investigate caching the filtered context list.

To maximize sharing, all context lists are stored in a single *forest* which is represented as an indexed implicit parameter, which we shall call `context`. The indexes into `context` are integers, but these indexes are private to the proof engine, so that destructive list insertion algorithms can be used without affecting the user view of context lists.

```
rule and_intro ===
    A,
    B
    ----------
    A and B.

rule implies_intro ===
    hyp+++[A] ---> B
    ----------------
    A => B.

define A <=> B ===
    (A => B) and (B => A).
    /*
    This enters A <=> B
    into a table, so that
    defined(A<=>B,(A=>B)and(B=>A)
    is true.
    */
```

```
rule all_intro(x2) ===
    hyp ??? nfi(x2) ---> A
    ---------------------
    all x2 A.

rule all_elim(A,x,T) ===
    all x A
    --------
    [T/x]A.

rule unfold ===
    [E/x]P
    ----------
    [D/x]P
    provided defined(D,E).
```

Figure 3: Inference rules used in the example proof

5.3. *An example proof*

This section shows all the data structures for a small proof that involves context terms, quantifiers and constraints. The theorem is:

```
theorem all_intro2  ===  A <=> (all x A)
                         provided x not_free_in A.
```

The inference rules used (see Figure 3) are the usual sequent calculus rules, in the style of Gentzen's LJ calculus [5], with the assumption rule given in Section 2.4.

Figure 4 shows a graphical view of the final proof tree. To avoid clutter, the figure uses the context-specification notation from Section 2.4 to describe changes to the `hyp` context, rather than repeating the full context at every node.

Figure 5 shows how this proof tree is represented within the Ergo 6 proof engine, by the bindings of three indexed implicit parameters: `node`, `hyp` and `context`.

The set of constraints associated with this proof is maintained by Qu-Prolog, so we shall not describe its representation in detail. Once the proof tree is complete, the set contains: the constraint `x not_free_in A` from the statement of the theorem; a derived constraint `x2 not_free_in A`, introduced when the `assump` rule applied to node 6 passes context term 1 (`A`) through the filter `nfi(x2)`; and a requirement that `x` and `x2` be distinct object variables. Only the first of these constraints becomes a side condition of the resulting theorem, because `x2` does not appear in the theorem, so the constraints involving it can always be satisfied.

5.4. *Performance*

The following performance experiments were done on a lightly loaded 466Mhz Celeron processor, with 128Mb RAM running Linux 2.2.14. Ergo was run using Qu-Prolog 6.0 and Isabelle99-2 was run under Poly ML (version 4.0).

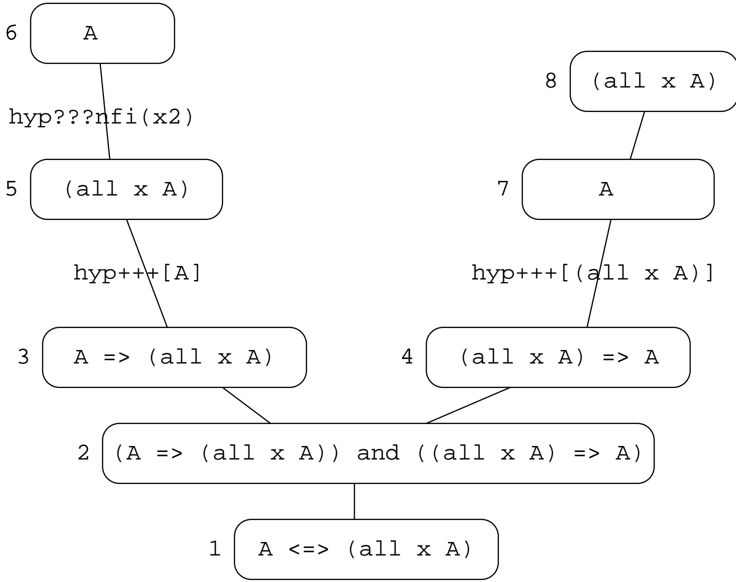

Figure 4: A proof of `A <=> (all x A) provided x not_free_in A`.

```
node(1)=proofnode(root,A<=>(all x A), jprop,unfold,        [2])
node(2)=proofnode(1, A => (all x A) and
                     (all x A) => A,  jprop,and_intro,     [3,4])

node(3)=proofnode(2, A => (all x A),  jprop,implies_intro,[5])
node(5)=proofnode(3, (all x A),       jpred,all_intro,     [6])
node(6)=proofnode(5, A,               jprop,assump,        [])

node(4)=proofnode(2, (all x A) => A,  jprop,implies_intro,[7])
node(7)=proofnode(4, A,               jpred,all_elim,      [8])
node(8)=proofnode(7, (all x A),       jprop,assump,        [])

hyp(1)=1     hyp(5)=2     context(1)=initialise
hyp(2)=1     hyp(6)=3     context(2)=add(1,A,1)
hyp(3)=1     hyp(7)=4     context(3)=filter(nfi(x2),2)
hyp(4)=1     hyp(8)=4     context(4)=add(2,(all x A),1)
```

Figure 5: Ergo 6 data structures for the proof in Figure 4

Table 1: Test results for Ergo and Isabelle

		Test 1	Test 2	Test 3
A.	Ergo	1.8	1.8	1.6
B.	Isabelle	3.5	20.9	3.4
C.	Isabelle (save)	7.9	22.1	4.4
	Speedup = C/A	6.1	12.3	2.8

In order to obtain a reasonable comparison between Ergo and Isabelle, it was necessary to carry out large proofs (in order to obtain accurate timing figures) that used equivalent sets of inference rules. This was accomplished by taking two small proofs for which there were already equivalent proofs in both systems, and repeatedly using rules that replicated goals in order to produce large proof trees.

The two theorems used were

$$(1) : (P \wedge Q \Rightarrow R) \Rightarrow ((Q \Rightarrow R) \Rightarrow R) \Rightarrow ((P \Rightarrow R) \Rightarrow R) \Rightarrow R,$$
$$(2) : (\exists x \forall y A) \Rightarrow \forall y \exists x A.$$

The proof of the first theorem takes eleven steps, and the second takes six steps. Tactics were written in each system that proved these theorems automatically.

In order to construct large proofs, a rule was used that replaced a goal with the goal conjoined with itself. Tactics were written that used this rule and conjunction introduction to generate four copies of the initial goal, prove three of these goals directly, and then repeat the process on the remaining goal. The replication step was carried out 400 times. This produced proof trees of about 15,600 proof nodes for the first theorem and about 9,600 proof nodes for the second theorem.

This technique was used to construct three test proofs in both Ergo and Isabelle:

- *Test* 1: a proof of (1) with goal replication applied to the last node at each level;
- *Test* 2: a proof of (1) with goal replication applied to the first node at each level;
- *Test* 3: a proof of (2) with goal replication applied to the last node at each level.

In both Ergo and Isabelle the tactics discharged the first open node first. As a consequence, the number of open nodes at any point in the proof in Test 1 did not exceed five, while in Test 2, the number of open nodes at one point in the proof reached 400.

In Ergo, the complete proof tree is stored during the proof while, by default, Isabelle does not store the proof tree. This can be changed by setting a flag. Table 1 lists the results for the three test proofs for Ergo and Isabelle (with and without the proof tree saved). All the times are in seconds.

This relatively good performance of Ergo is achieved largely because each inference rule is compiled directly into Qu-Prolog code (as described in Section 4.2), which allows the compiler to optimize each inference rule according to its structure. In contrast, Isabelle inference rules are represented purely as data, so these opportunities for compilation and optimization are not possible.

As described in Section 5.2, the context lists of Ergo 6 are not compiled, because they change frequently during a proof. So Ergo 6 can be slower than Isabelle in proofs that search context lists very frequently (because both systems are searching data structures, and ML

is generally faster than Prolog). However, in most proofs only a small percentage of proof steps search the context lists (2/11 and 1/6 steps in the above example proofs), so the speed of applying (compiled) inference rules far outweighs the slower accessing of context lists, as the above examples illustrated.

Other experiments showed that for simple inference rules, Ergo 6 is about five times faster than the earlier window-inference-based version of Ergo. Furthermore, the speed of most inferences in Ergo 6 is largely independent of the size of the context lists and the number of variables in the whole proof, whereas the earlier version slowed down dramatically as these increased.

6. *Conclusions*

This paper has described the design and implementation of the Ergo 6 proof engine. It is a generic interactive proof engine, with expressiveness similar to that of Isabelle (though it does not support type-based overloading as Isabelle does). Its performance is quite good in comparison with that of Isabelle, though more experiments on large proofs with identical theories and tactics would be necessary to make accurate comparisons.

This paper illustrates one way of effectively using Prolog implementation technology to implement a theorem prover that is highly generic and flexible. The key principle is to represent each inference rule by a Prolog clause that can be queried as if it is data, but can still be efficiently compiled and executed. A similar effect could perhaps be obtained in other systems (for example, Isabelle) by having two representations of each inference rule: one as a data structure and the other as compiled code that executes the inference rule in the most common (backwards) proof style.

Our representation of the proof tree relies on having efficient backtrackable hash tables (indexed implicit parameters), but these could be added to other Prolog systems. The use of Qu-Prolog provides other advantages, such as more flexible unification and support for quantifiers, substitutions and constraint management.

The Ergo 4 proof engine was based on window inference [**23**]. Ergo 6 is more general, but can easily support window inference by proving appropriate inference rules. In fact, the generic nature of Ergo 6 makes it easy to support aspects of window inference that were not easy to implement in Ergo 4 (for example, conditional opening rules, proof of derived opening rules, higher-order relations), simply by proving additional inference rules. By careful design, utilising the strengths of Prolog, with Ergo 6 we have been able to improve both speed and flexibility dramatically (‘*have our cake and eat it too*’).

Appendix A. *The Ergo 6 and Qu-Prolog releases*

A release of Ergo 6 is available at:

`http://www.svrc.uq.edu.au/Software/ErgoHome.html`.

This includes a hierarchy of about 40 predefined theories, ranging from Johansson’s propositional logic through to arithmetic and ZFC set theory. Qu-Prolog is needed to build and run Ergo 6. A release of Qu-Prolog is also available from the Software Verification Research Centre, University of Queensland, Australia, where Ergo and Qu-Prolog were developed. Readers will find the material at:

`http://www.svrc.uq.edu.au/Software/QuPrologHome.html`.

Acknowledgements. Thanks are due to Anthony Bloesch, Ed Kazmierczak, Peter Kearney, Peter Lindsay, John Staples and Owen Traynor for helpful input and discussions about future architectures for the Ergo proof tool. Special thanks to John Staples and Steve Reeves for their comments on early drafts of this paper. This work was funded by the Software Verification Research Centre, a Special Research Centre of the Australian Research Council.

References

1. R. Bornat and B. Sufrin, 'Jape: A calculator for animating proof-on-paper', *Automated Deduction – CADE-14, 14th International Conference on Automated Deduction, Townsville, North Queensland, Australia*, Lecture Notes in Comput. Sci. 1249 (ed. William McCune, Springer, 1997) 412–415. 194

2. R. S. Boyer and J. S. Moore, *A computational logic* (Academic Press, 1979). 194

3. P. Deransart, A. Ed-Dbali and L. Cervoni, *Prolog: the standard reference manual* (Springer, 1996). 196

4. Edsger W. Dijkstra, *A discipline of programming* (Prentice-Hall, Englewood Cliffs, NJ, 1976). 196

5. Gerhard Gentzen, 'Investigations into logical deduction', *The collected papers of Gerhard Gentzen*, Stud. Logic Found. Math. F (ed. M. E. Szabo, North-Holland, 1969; first published 1934) 68–131. 201, 213

6. Mike Gordon and T. F. Melham (eds), *Introduction to HOL: a theorem-proving environment for higher-order logic* (Cambridge University Press, 1993). 194

7. Nicholas Hamilton, Ray Nickson, Owen Traynor and Mark Utting, 'Interpretation and instantiation of theories for reasoning about formal specifications', *Proc. Twentieth Australasian Computer Science Conference* (*ACSC'97*), Austral. Comput. Sci. Comm. 19 (ed. M. Patel, Macquarie University, Sydney, 1997) 37–45; see also Technical Report SVRC-96-21, Software Verification Research Centre, The University of Queensland, 1996. 196

8. P. van Hentenryck, *Constraint satisfaction in logic programming*, Logic Programming Series (The MIT Press, Cambridge, MA, 1989). 200

9. C. B. Jones, K. D. Jones, P. A. Lindsay and R. Moore, **mural***: a formal development support system* (Springer, 1991). 194

10. Sara Kalvala, 'Annotations in formal specifications and proofs', *Form. Meth. Syst. Des.* 5 (1994) 119–144. 202

11. S. Kanger, 'A simplified proof method for elementary logic', *Computer programming and formal systems* (ed. P. Braffort and D. Hirschberg, North-Holland, 1963) 89–95. 200

12. Matt Kaufmann and J. Strother Moore, 'Design goals for ACL2', CLI Technical Report 101, Computational Logic, Inc., 1717 West Sixth Street, Suite 290, Austin, Texas 78703-4776, August 1994. 194

13. A. P. Martin, P. H. B. Gardiner and J. C. P. Woodcock, 'A tactic calculus' (abridged version), *Formal Aspects of Computing* 8 (1996) 479–489; the full version is available at the *Formal Aspects of Computing* FTP site: `ftp://ftp.cs.man.ac.uk/pub/fac`. 196

14. ANDREW MARTIN, RAY NICKSON and MARK UTTING, 'A tactic language for Ergo', *Formal Methods Pacific '97,* Discrete Math. Theor. Comput. Sci. (ed. Lindsay Groves and Steve Reeves, Springer, 1997) 186–207. 196

15. PETER NICKOLAS and PETER J. ROBINSON, 'The Qu-Prolog unification algorithm: Formalisation and correctness', *Theoret. Comput. Sci.* 169 (1996) 81–112; also available as Software Verification Research Centre Technical Report 94-23. 197, 203, 203, 205

16. SAM OWRE, JOHN RUSHBY, NATARAJAN SHANKAR and FRIEDRICH VON HENKE, 'Formal verification for fault-tolerant architectures: prolegomena to the design of PVS', *IEEE Trans. Software Engrg* 21 (1995) 107–125. 194

17. L. C. PAULSON, 'Natural deduction as higher-order resolution', *J. Logic Programming* 3 (1986) 237–258. 198, 208

18. L. C. PAULSON, *Logic and computation: interactive proof with Cambridge LCF*. Cambridge Tracts Theoret. Comput. Sci. (Cambridge University Press, 1987). 198

19. LAWRENCE C. PAULSON, 'The foundation of a generic theorem prover', *J. Automat. Reason.* 5 (1989) 363–397. 197

20. LAWRENCE C. PAULSON, with contributions by TOBIAS NIPKOW, *Isabelle: a generic theorem prover*, Lecture Notes in Comput. Sci. 828 (Springer, 1994). 194, 198

21. FRANK PFENNING, 'Logical frameworks', *Handbook of automated reasoning*, vol. 2 (Elsevier Science Publishers, 2001). 194, 197, 197, 201

22. PETER ROBINSON, 'Qu-Prolog 6.0 user guide', Tech. Rep. 00-20, Software Verification Research Centre, School of Information Technology, The University of Queensland, Brisbane 4072, Australia, (2000);
`http://svrc.it.uq.edu.au/Bibliography/svrc-tr.html?00-20`.
211

23. PETER J. ROBINSON and JOHN STAPLES, 'Formalizing the hierarchical structure of practical mathematical reasoning', *J. Logic Comput.* 3 (1993) 47–61. 198, 216

24. PETER VAN ROY, '1983–1993: The wonder years of sequential PROLOG implementation', *J. Logic Programming* 19, 20 (1994) 385–441. 196

25. JOHN RUSHBY, 'Mechanized formal methods: where next?' *FM99: The World Congress in Formal Methods, Toulouse, France, 1999*, Lecture Notes in Comput. Sci. 1708 (ed. Jeannette Wing and Jim Woodcock, Springer, 1999) 48–51. 194

26. J. STAPLES, P. J. ROBINSON, R. A. PATERSON, R. A. HAGEN, A. J. CRADDOCK and P. C. WALLIS, 'Qu-Prolog: an extended Prolog for meta level programming', *Metaprogramming in logic programming* (ed. H. Abramson and M. H. Rogers, MIT Press, Cambridge, MA, 1989) 435–452. 196, 210

27. MARK STAPLES, 'Window inference in Isabelle', presented at the Isabelle Users' Workshop, 18-19 April 1995, University of Cambridge; available from
`http://www.cl.cam.ac.uk/users/lcp/Workshop/index.html`. 198

28. MARK E. STICKEL, 'A Prolog technology theorem prover: Implementation by an extended Prolog compiler', *J. Automat. Reason.* 4 (1988) 353–380. 208, 209

29. MARK UTTING and STEVE REEVES, 'Implementing Z_c substitutions in Ergo', 'WESTAPP 2000: The Third International Workshop on Explicit Substitutions: Theory and Applications to Programs and Proofs, Norwich, UK', preprint, 2000, 35–49; available from `http://www.cs.waikato.ac.nz/~marku`. 200, 200, 201

30. Mark Utting, Ray Nickson and Owen Traynor, ‘Theory structuring in Ergo 4.1’, *Austral. Comput. Sci. Comm.* 18 (1996), Proceedings of CATS’96, Computing: The Australian Theory Symposium, 137–146. 196

Mark Utting marku@cs.waikato.ac.nz

Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton
New Zealand

Peter Robinson pjr@itee.uq.edu.au

Information Technology and Electrical Engineering,
The University of Queensland
Brisbane
Queensland 4072
Australia

Ray Nickson Ray.Nickson@mcs.vuw.ac.nz

School of Mathematical and Computing Sciences,
Victoria University of Wellington
P.O. Box 600
Wellington
New Zealand.