

**SOFTWARE VERIFICATION RESEARCH CENTRE
DEPARTMENT OF COMPUTER SCIENCE
THE UNIVERSITY OF QUEENSLAND**

**Queensland 4072
Australia**

TECHNICAL REPORT

No. 97-16

A Tactic Language for Ergo

**Andrew Martin, Ray Nickson,
and Mark Utting**

14th February 1997

Phone: +61 7 3365 1003

Fax: +61 7 3365 1533

Note: Most SVRC technical reports are available via anonymous ftp, from `ftp.cs.uq.edu.au` in the directory `/pub/SVRC/techreports`.

A Tactic Language for Ergo

Andrew Martin, Ray Nickson,
and Mark Utting

Abstract

A new version of the Ergo theorem prover is under development. It uses a single tactic language, based on *Angel*, for tactic programming, user interface, and proof representation. This paper describes the language as it is used in each of these cases, and explains the details of its implementation in Qu-Prolog. An example from classical propositional calculus is included.

1 Introduction

Ergo is an interactive proof tool that has been designed and implemented at the SVRC over the last ten years. It is implemented in Qu-Prolog (Robinson and Hagen, 1997), and is designed to be extensible, so that users can add new theories, tactics and user interfaces. Ergo 5 is currently under development. Having no inbuilt object logic, it is a generic prover that can be instantiated by providing a collection of axiomatic and/or definitional theories. The core of Ergo 5 provides support for (uninterpreted) sequents with named tuples of arbitrary terms as antecedents and single terms as consequents. The interpretation of sequents, and the axioms and fundamental inference rules that relate them to object-level terms and formulas, are under the control of the theory developer. In this paper, our examples will use the sequents with a single set of antecedents called hypotheses, which are to be understood as the antecedents in Gentzen's sequent calculus (Gentzen, 1969).

Ergo 5 proofs are tree-structured, constructed backwards in a style reminiscent of natural deduction. An inference rule is a meta-level structure of the form **from** $P_1 \& \dots \& P_n$ **infer** C , in which each term P_1, \dots, P_n, C is a sequent. The connectives $\&$ and **from-infer** are respectively meta-level conjunction and implication, which can be related to object-level connectives by the inference rules of a theory of intuitionistic logic, for example. The meaning of the meta-level implication **from** P **infer** C is that conclusion C can be deduced from premiss P . The meaning of the meta-level conjunction $P_1 \& P_2$ is that both P_1 and P_2 are (to be) proved.

A successful application of an inference rule **from** $P_1 \& \dots \& P_n$ **infer** C (where $n \geq 0$) transforms the proof tree. Some identified open node whose term matches C is extended with n child nodes, the terms of which are the corresponding matches for the premisses. We can leave the proof tree implicit, instead viewing a rule as a mapping from some specified open node to a sequence of n open nodes.

Earlier versions of Ergo used Qu-Prolog (the implementation language) as a tactic language (Whitwell, 1992). This allowed efficient and powerful tactics to be written, but created some problems.

- Some users were daunted by the power of unrestricted Prolog, and would prefer a simpler dedicated language.
- Full Prolog is a complex language and is not amenable to reasoning.
- The interface presented to the tactic programmer was not precisely defined (Qu-Prolog lacks a module system, though one is planned), and was different from the command-line interface available to users.
- Failure to distinguish the implementation language from the tactic language would frustrate any future attempts to re-implement in another language.

This paper describes the tactic language which is implemented in Ergo 5. We have a number of design aims:

- The tactic language should be simple but powerful.
- The semantics should be clean, so that we can reason about tactics.
- There should be a single language that is usable as the command-line interface, for writing tactics, and for presenting proofs.
- The tactic language should be clearly distinguished from the implementation language.

We have implemented an instantiation of *Angel* (see Section 2) for Ergo 5. This implementation raised some interesting issues about the handling of sub-proofs that are not directly addressed in the *Angel* core. These issues, and their resolution in the Ergo 5 implementation of *Angel*, are discussed in Sections 3 and 4. Section 5 demonstrates how the Guntree tactic constructs can be used to define some simple general-purpose proof procedures. In Section 6 we describe the implementation of the tactic language, and Section 7 shows how the tactics are used in an example proof. A final section discusses the context of the work, emphasising the advances over earlier work, and points to future research directions.

2 Angel

Since the Edinburgh LCF project first described *tactics* as programs for directing proof tools (Gordon, Milner, and Wadsworth, 1979), the notion has become very widespread. Many systems implement some form of tactic language for directing proofs. Sometimes, following the LCF style, tactics construct possible proofs which are later validated. In other systems the tactics form an extension of the set of primitive inference rules. The soundness of each tactic application is assured by ensuring that its only interaction with the proof under construction is by application of the primitive inference rules.

Angel (Martin, Gardiner, and Woodcock, 1996) is a generic tactic language. Initially it was intended to support proofs in the second of these styles, that

is, providing a framework for the composition of primitive inference rules in the construction of backwards (goal-directed) proofs, but it turns out to be more general than this. Term rewriting, for example, which in Cambridge LCF (Paulson, 1987) is directed by a separate set of operators from those which describe tactics, could also be described using Angel. Program refinement is an instance of term rewriting (Nickson and Hayes, 1996), and we expect Ergo 5’s implementation of Angel to support refinement tactics as well as proof tactics.

The concept of a *goal* is central to Angel’s semantics, and to its genericity. A goal is an object of the underlying system that can be manipulated by an inference rule. An execution of a basic Angel tactic maps a single goal to another. In a term-rewriting system, a goal is a term of the logic, and a tactic application yields a new goal (the transformed term). In Ergo 5, a goal is a sequence of open proof nodes; an application of a tactic yields a new sequence of open nodes. Section 3 describes the new *parallel composition* operator of the Guntree language, which can analyse goals (node sequences) into subgoals (subsequences or individual nodes).

The paper of Martin et al. (1996) describes Angel in an abstract way, independent of any particular implementation. It gives denotational semantics and a (more accessible) axiomatic semantics, in the form of a large collection of laws which allow one tactic to be transformed into another. Also presented are some commonly used derived tactics—together with derived transformation rules. Further derived tactics are discussed in Martin’s thesis (1994).

Because Angel is a small language, its semantics is quite clean and easy to reason about. Nevertheless it is able to describe a large class of useful algorithms. The language is named Angel because the account in (Martin et al., 1996) makes tactics angelically nondeterministic. That is, a tactic is a relation between goals, rather than a single function. Thus we may arrange that a compound tactic will fail only if there is no possible path from input to output: an implementation must avoid dead-end paths, which is typically achieved by a backtracking search. The angelic style removes the need for the tactic programmer to program backtracking explicitly.

The principal constructs of Angel are:

Rules The atomic tactic ‘**rule** R ’ applies an inference rule. If applied in the domain of the rule, it will map a goal to a single new goal. If a rule is applied outside its domain, it fails.

Special atomic tactics The atomic tactic **skip** always succeeds exactly once, leaving its goal unchanged. The atomic tactic **fail** always fails.

Sequential composition A sequential composition $t_1 ; t_2$ first applies t_1 and then applies t_2 to the resulting goal. If t_1 fails, the whole composition fails; if t_2 fails, other alternatives in t_1 may be explored.

Alternation An alternation $t_1 \mid t_2$ applies either t_1 or t_2 , succeeding if either one succeeds.

Cut the cut operator $!t$ prunes away all but the first alternative from t . That is, if some subsequent tactic fails, no alternatives within t will be explored.

We adopt the following order of precedence for operator binding: function application (such as application of **rule** to its argument) binds most tightly, followed

by cut, then sequential composition. The parallel combinator \parallel (to be defined in Section 3) is next, and alternation binds most loosely.

These semantics may be expressed by viewing a tactic as a function mapping a single input goal to a sequence of alternative output goals. Recursive tactics are given meaning using fixpoints. A forthcoming paper generalises the semantic description of Angel tactics, discussing which of the following laws hold in different circumstances (for instance, if a notion of a single-threaded state is added to the denotation of the tactics).

A collection of nineteen laws illustrates the interactions of the basic Angel tactic constructs. Some are straightforward: **skip** ; $t = t$, and **fail** ; $t = \mathbf{fail}$, etc. The right-distributive law is perhaps the most indicative of the angelic semantics: we have $(t_1 \mid t_2) ; t_3 = (t_1 ; t_3) \mid (t_2 ; t_3)$. This is not the case in (say) LCF or HOL.

In addition to these basic constructs, Angel also provides for *structural combinators* which permit tactics to be applied to sub-expressions of the goal. The set of structural combinators provided by an Angel implementation will generally depend on the nature of the inference system that underlies Angel. For example, the logic that underlies a refinement tool will have inference rules that allow refinement of the components of a compound program fragment; an instantiation of Angel for a refinement tool might include structural combinators corresponding to program constructors, as in the tactic language of Red (Vickers, 1990).

In a theorem prover supporting backwards inference in the style of natural deduction, such as Ergo 5, (Utting, 1996), the input of a typical fundamental inference step is a goal to be proved (the *conclusion* of the inference rule), and the result is a set or sequence of *premisses*. The account of Angel in (Martin et al., 1996) describes a *parallel* tactic combinator for dealing with proof trees which branch in this way. A different approach has been taken in Ergo 5; this is described in Section 3, below.

3 Gumtree for Tactic Programming

The tactic-based interface for Ergo 5 is called *Gumtree*. We have decided to use a single language for the interactive prover interface, for tactic programming and for the representation of proofs. This is a language based on Angel, with various adaptations. A major development is in the treatment of parallel subgoals, described in the following sections. Other changes make the language easier to use in an interactive context. We believe that the majority of the laws of Angel will also apply to Gumtree—see (Martin, Nickson, and Utting, 1997). We hope that these laws will permit tactic optimisation, as well as enhancing users' understanding of the semantics of the tactics they write.

Gumtree implements all of the tactic combinators described above. Ergo 5 is a generic prover with no inbuilt core logic. The genericity means that it is not possible to provide a fixed set of Angel structural combinators corresponding to object-level connectives. Instead, we introduce a single structural combinator that corresponds to the tree-structure arising from the meta-level conjunction in the premisses of inference rules. We write this operator \parallel (pronounced 'parallel').

Open Subgoals Because in a goal-directed proof the application of a rule or tactic to a single goal may give rise to several subgoals, and it may be appropriate to deal with several at once, Gumtree’s goals are *sequence* of subgoals (which are leaves of the proof tree) to record the user’s current position.

For example, with an initial goal of $\vdash A \wedge B \wedge C$, two applications of the rule *and-intro* will give rise to a new sequence of subgoals: $\vdash A \quad \& \quad \vdash B \quad \& \quad \vdash C$.

Gumtree’s Parallel In the paper of Martin et al., the parallel operator and meta-conjunction used in this way were strictly non-associative binary operators. This has proven to be sometimes hard to use. For example, following the application of a large tactic which produces three subgoals, it is necessary to know how they are grouped, before further tactics or rules can be applied. In the example above, the structure of the proof tree (and any tactics which are to be subsequently applied) would depend on the bracketing used in the term $A \wedge B \wedge C$, which is unfortunate. A completed proof branch is denoted by a special distinguished subgoal $()$. Special processing is needed to arrange that for any subgoal g , we have that $g\&()$ is transformed to g .

In Gumtree we allow \parallel and $\&$ to be *associative*, and to remove empty subgoals seamlessly. Thus, in effect, we treat the set of current subgoals as a sequence, and any collection of tactics which is to operate on them as a sequence, too. Added subtlety comes from the fact that in a tactic such as $(t_1 \parallel t_2)$, either (or both) of t_1 and t_2 may in turn contain a parallel composition—and by the associative property, therefore, this tactic may apply to a sequence of (say) five subgoals. Further discussion of this design decision, and the resulting tactic laws, can be found in the technical report (Martin et al., 1997).

Tactic Arity The manner in which these subgoals are associated with the tactics is clearly critical. When the tactics exhibit nondeterministic behaviour, the complication is multiplied. To explain how parallel tactics are composed, we define a notion of tactic *arity*. The arity of a deterministic tactic is a pair $m \rightarrow n$; a tactic with this arity will transform a goal with m subgoals into one with n subgoals. A system of arity-checking assists in pre-execution correctness checks for such tactics, and allows certain optimisations (see Section 6).

Behaviour of Parallel Informally, parallel behaves as follows: when the tactic $(t_1 \parallel \dots \parallel t_n)$ is applied to the goal $(g_1\&\dots\&g_m)$, tactic t_1 (of arity $j \rightarrow k$) operates on some initial subsequence of the g_i —say $g_1\&\dots\&g_j$ —returning some sequence $h_1\&\dots\&h_k$, and leaving t_2 to operate on some initial sequence of $g_{j+1}\&\dots\&g_m$, etc. Eventually all the h_i are concatenated to form a new goal, provided the last tactic is of a suitable arity to operate on all the remaining subgoals.

When one or more tactics in the parallel composition is nondeterministic, each of the possible outcomes must be tried. The outcomes may have different arities, so that arities of nondeterministic tactics may not be fixed.

Fixed-arity tactics The tactic **skip** is fully general; it will apply to any sequence of subgoals, and succeed, leaving them unchanged. Sometimes it is useful to have less general versions of **skip** which match specific numbers of subgoals. Thus we define **zero** which succeeds when the current list of subgoals is empty,

and fails otherwise, and **one** which succeeds (leaving the goal unchanged) when there is exactly one current subgoal. Then we may define

$$two = \mathbf{one} \parallel \mathbf{one}$$

and more generally

$$\begin{aligned} fix\ 0 &= \mathbf{zero} \\ fix\ (n + 1) &= fix\ n \parallel \mathbf{one} \end{aligned}$$

These tactics obey a number of interesting laws. Details may be found in the technical report of Martin et al. (1997).

Examining terms Angel includes a powerful construct (π) that can be used to write tactics whose behaviour depends on the detailed structure of the goals to which they are applied. For Guntree, we have adopted a simpler construct. The tactic **term**(T) succeeds if the current goal has a single subgoal, and the term associated with that subgoal is unified with T . Success or failure of **term** can be used to write tactics that conditionally branch based on the terms to which they are applied; the variable bindings resulting from unification can provide parameters for rule applications.

4 Guntree for User Interface

As well as being used for programming tactics, Guntree is the language used as Ergo's command line interface. This imposes some additional requirements on the language, and leads to some further departures from Angel.

The command line interface displays a prompt showing the current set of open nodes. Initially this is just the node which will be the root of the proof tree, whose sequent is the sequent to be proved. The user enters a Guntree command; this may be a Guntree tactic (such as a rule application, or a sequential or parallel composition, or a user-defined tactic), or a special interface command.

Executing tactics If the user command is a tactic, the tactic is executed, with the goal sequence shown in the prompt as input. If the tactic fails, a message is printed and the new prompt is the same as the old. If the tactic succeeds, the new prompt shows the goals that resulted from the tactic application. If there may be alternative solutions to the tactic, just the first solution is shown. The alternatives may be explored on backtracking.

Special commands The system maintains a history of all commands that have contributed to the proof. The **history** command displays this history as a numbered list. The **retry** command triggers backtracking to explore alternative solutions for previous commands; if there are no alternatives for the most recent command, that command fails and disappears from the history. Subsequent **retry** commands will explore alternatives for the next most recent command. With a numeric argument corresponding to one of the numbered history entries, **retry**(N) starts the process of finding alternatives from the specified command; those after it fail immediately. The **commit** command discards any alternative

solutions for the last command in the history, as if the command were surrounded by a cut operator; with a numeric argument it forms the sequential composition of all commands beyond that one in the history, and commits the resulting composition. The command `undo` is equivalent to `commit` followed by `retry`; with an argument N , it is equivalent to `commit(N)` and `retry`. These special interface commands cannot be used in tactics.

Global commands Other global Ergo commands can be used from the Gumtree prompt. The user can define named tactics with parameters, examine the state of the proof in various formats, alter the information displayed in the prompt, and save and load portions of proof trees in files. These global commands are not directly available in tactics (though they can be executed by escaping to Prolog; see Section 6).

Navigating the proof tree Because interactive users find it is sometimes convenient to focus attention on only part of the proof tree, two further commands are added to the Gumtree tactic language to support the user interface. The `defer` command temporarily removes the open proof nodes to which it is applied from the goal sequence, as if they had been proved. The `select(Nodes)` command makes the specified sequence of nodes current, deferring the sequence that was previously current and awakening any previously deferred nodes that are selected. Node selection and tactic execution may be combined, as `Nodes :: tactic`. To facilitate proof tree navigation using these commands, nodes may be named. The command `name(Name)` associates the name with the proof node to which it is applied; `names([Name1, . . . , Namen])` names many nodes simultaneously. The name will thenceforth be used in place of the number to identify the node.

These navigation commands are intended for use in interactive proof scripts. Their use in general-purpose tactics is discouraged as it makes analysis (and hence checking and optimisation) of tactics difficult.

5 Derived Tactics

Using the tactics we have defined, some derived tactics may be constructed. The following tactics arise in most tactic-based systems, though the names are not well standardised. The equations below may be viewed as definitions of the derived tactics, though the tactics may also be implemented directly (for example, for efficiency's sake).

try t attempts to apply t , but succeeds whether t is successful or not. *exhaust* t applies t repeatedly; as many times as possible.

$$\begin{aligned} \textit{try } t &= t \mid \mathbf{skip} \\ \textit{exhaust } t &= t ; \textit{exhaust } t \mid \mathbf{skip} \end{aligned}$$

Variations on these, such as exhaustive application subject to a minimum or maximum number of applications, are also possible.

We also define tactics which apply their arguments to the current parallel goals: *every* t will apply t to every available subgoal; *tryevery* is a robust version of *every* which applies t wherever possible, and leaves unchanged any goal to which t is not applicable:

$$\begin{aligned} \text{every } t &= (t \parallel \text{every } t) \mid \mathbf{zero} \\ \text{tryevery } t &= \text{every}(t \mid \mathbf{one}) . \end{aligned}$$

If we wish to try to apply t not just to each of the current goals, but also to any resulting subgoals, we can employ a form of breadth-first search. First, we define *any* t , which applies t exactly once to some of the current goals, and *some* t , which behaves like *tryevery* t , but fails if *all* of the parallel applications of t fail.

$$\begin{aligned} \text{any } t &= \mathbf{skip} \parallel t \parallel \mathbf{skip} \\ \text{some } t &= \text{every}(\text{any } t) \end{aligned}$$

Now, the breadth-first application of t is simply the exhaustive application of *some* t : try to apply t to each of the current subgoals. If it does not apply to any of them, then stop. Otherwise, we have a new list of subgoals to which the procedure can be applied recursively.

$$\text{bfs } t = \text{exhaust}(\text{some } t)$$

We may also define a depth-first application of t , and expect that for a deterministic t the outcome should be the same as that for the breadth-first application. The relative efficiencies will depend on the nature of t .

$$\text{dfs } t = \text{every}(t ; \text{dfs } t \mid \mathbf{one})$$

6 Implementation

Ergo is implemented in Qu-Prolog (Robinson and Hagen, 1997).¹ Qu-Prolog provides *implicit parameters* (Cheng, Robinson, and Staples, 1991), which behave like global variables, except that updates are undone on failure (as with normal Prolog variables), so that the Prolog backtracking search can be used. Implicit parameters are efficiently implemented in Qu-Prolog using destructive update (with trailing of values), but their semantics can be cleanly described by supposing that every Prolog procedure that refers to an implicit parameter has an extra argument providing the value of that parameter, and every procedure that updates the parameter has an extra argument for returning the new value.

A recent Qu-Prolog development implements *indexed implicit parameters*, which are associative arrays with the same properties as implicit parameters. Individual elements of indexed implicit parameters can be accessed and updated in near-constant time.

The Ergo proof tree is represented by an indexed implicit parameter (Utting, 1996). Each node has a unique identity, which is its index into the indexed implicit parameter. The value stored there is a term of the form:

$$\text{node}(\text{Term}, \text{Context}, \text{Rule}, \text{SubNodes})$$

The sequent represented by this node is $\text{Context} \vdash \text{Term}$. If the proof node has been justified, *Rule* is the name of the inference rule used, and *SubNodes* is a

¹This section assumes some familiarity with Prolog, but not with Qu-Prolog.

list (possibly empty) of the nodes representing the premisses of the rule. If the node is open, *Rule* and *SubNodes* are undefined.

The interface to the proof tree abstract data type is via the procedures

```
proof_start(Term, Context, Id)
```

(which creates the open root node *Id* of a new proof tree) and

```
proof_step(Rule, Id, SubNodes)
```

(which applies *Rule* to justify the node *Id* and returns the list of new nodes *SubNodes*). Gumtree tactics use these procedures to modify the proof structure. Additionally, the tactic interpreter must maintain the sequence of current goals, partitioning it appropriately among the branches of parallel compositions.

Essentially, tactics are viewed as nondeterministic functions between current goal sequences, that modify proof trees by ‘side effect’ (work is in progress to adapt the abstract model of Angel to incorporate side-effects, using monads (Moggi, 1989)). Nondeterministic functions are easily represented in Prolog as procedures with two arguments: one corresponding to the input and one to the output. The proof tree is modified (apparently by ‘side effect’) using the procedures for updating implicit parameter arrays.

The tactic **skip** is the identity operation; it leaves both the proof tree and current goal sequence unchanged. Its implementation in Prolog can be described by the clause:

```
tactic(skip, Goals, Goals).
```

The tactic **one** maps a singleton goal sequence to itself, without modifying the proof tree:

```
tactic(one, [Goal], [Goal]).
```

This will fail unless the input goal sequence (passed as the second argument) unifies with *[Goal]*, i.e., it must be a sequence of length one. The output goal sequence (returned as the third argument) is the same as the input. This generalises to *fix n*:

```
tactic(fix(N), Goals, Goals) :-
    length(Goals, N).
```

The tactic that applies a rule accepts a single input, to which the rule is applied, and returns the premisses of the rule:

```
tactic(rule(R), [InGoal], OutGoals) :-
    proof_step(R, InGoal, OutGoals).
```

Sequential composition simply ‘threads’ the goal sequence through the composed tactics, while alternation is represented directly by Prolog nondeterminacy, and **!** by Prolog cut.

```
tactic((T1;T2), InGoals, OutGoals) :-
    tactic(T1, InGoals, MidGoals),
    tactic(T2, MidGoals, OutGoals).
tactic((T1|T2), InGoals, OutGoals) :-
    tactic(T1, InGoals, OutGoals).
tactic((T1|T2), InGoals, OutGoals) :-
```

```

    tactic(T2, InGoals, OutGoals).
tactic(!T, InGoals, OutGoals) :-
    tactic(T, InGoals, OutGoals),
    !.

```

Note that the two tactics in an alternative composition may have different arities, so by analysing arities (perhaps during compilation) it may be possible to decide that one clause is inapplicable, reducing the need for search.

6.1 Parallel Composition

A parallel composition of two tactics must partition the input goal sequence between the composed tactics, and concatenate the results:

```

tactic((T1||T2), InGoals, OutGoals) :-
    append(InGoals1, InGoals2, InGoals),
    tactic(T1, InGoals1, OutGoals1),
    tactic(T2, InGoals2, OutGoals2),
    append(OutGoals1, OutGoals2, OutGoals).

```

This implementation would be hopelessly inefficient: for each possible partitioning of `InGoals`, it tries `T1` and (perhaps) `T2`. Instead, we augment `tactic/3` with a further argument, and allow tactics to operate on any prefix of the input goal sequence, returning the unused input as well as the output. Now, sequential composition passes the result of `T1` as input to `T2`, while parallel composition passes the unused goals.

```

tactic(fix(N), InGoals, UnusedGoals, OutGoals) :-
    %% OutGoals = the initial segment of InGoals, of length N;
    %% UnusedGoals = the remainder.
    length(OutGoals, N),
    append(OutGoals, UnusedGoals, InGoals).
tactic(rule(R), [InGoal|UnusedGoals], UnusedGoals, OutGoals) :-
    %% The rule is applied to the first InGoal, leaving the
    %% remainder unused. The OutGoals are the premisses of the
    %% rule application.
    proof_step(Rule, InGoal, OutGoals).
tactic(term(T), [InGoal|UnusedGoals], UnusedGoals, [InGoal]) :-
    %% T is unified with the term associated with the first InGoal,
    %% leaving the unused. The single output goal is the first InGoal.
    proof_node_term(InGoal, T).
tactic((T1;T2), InGoals, UnusedGoals, OutGoals) :-
    %% The unused goals are those left unused by T1;
    %% the output goals of T1 are passed to T2, and it must use
    %% all of them.
    tactic(T1, InGoals, UnusedGoals, MidGoals),
    tactic(T2, MidGoals, [], OutGoals).
tactic((T1|T2), InGoals, UnusedGoals, OutGoals) :-
    %% Simple nondeterministic choice.
    tactic(T1, InGoals, UnusedGoals, OutGoals).
tactic((T1|T2), InGoals, UnusedGoals, OutGoals) :-
    tactic(T2, InGoals, UnusedGoals, OutGoals).
tactic(!T, InGoals, UnusedGoals, OutGoals) :-
    tactic(T, InGoals, UnusedGoals, OutGoals),
    !.

```

```
tactic((T1||T2), InGoals, UnusedGoals, OutGoals) :-
    %% The unused goals of T1 are passed to T2; any that T2
    %% does not use are unused by the composition.
    %% are merged.
    tactic(T1, InGoals, Unused1, Out1),
    tactic(T2, Unused1, UnusedGoals, Out2),
    append(Out1, Out2, OutGoals).
```

The remaining occurrences of `append` (which leads to non-linear time behaviour) can be eliminated using standard Prolog transformations (accumulator pairs and difference lists).

To apply a tactic, the user interface simply calls `tactic/4`, passing the current goal sequence as *InGoals*, ensuring that no goals are left unused, and obtaining the new goal sequence from *OutGoals*:

```
do_command(T, InGoals, OutGoals) :-
    tactic(T, InGoals, [], OutGoals).
```

6.2 Compilation of Tactics

The above is a simple tactic interpreter. Using the meta-programming facilities (`term_expansion` and definite clause grammars) available in most Prolog systems, including Qu-Prolog, we can modify the definitions to produce a tactic compiler. Instead of passing a tactic to `tactic/4` for execution, the compiler uses the clauses of `tactic/4` (suitably annotated) as a database of translation rules. For example, the tactic *fix N* is translated directly to

```
length(OutGoals, N), append(OutGoals, UnusedGoals, InGoals).
```

Recursive calls on `tactic/4` can almost always be handled at compile time (i.e., as recursive calls on the translator); for example, the tactic

```
rule R1; (one || rule R2 || (rule R3; fix 2)); fix 4
```

is translated to:

```
%% rule(R1)
InGoals = [InR1|UnusedR1],
proof_step(R1, InR1, OutR1),
%% ;(one||
length(OutOne, 1),
append(OutOne, UnusedOne, OutR1),
%% rule(R2)||
UnusedOne = [InR2|UnusedR2],
proof_step(R2, InR2, OutR2),
%% rule(R3)
UnusedR2 = [InR3|UnusedR3],
proof_step(R3, InR3, OutR3),
%% ;fix(2)
length(OutFix2, 2),
append(OutFix2, UnusedFix2, OutR3),
UnusedFix2 = [],
%% )
UnusedR3 = [],
append(OutOne, OutR2, OutMidA),
```

```

append(OutMidA, OutFix2, OutMid),
%% ;fix(4)
length(OutFix4, 4),
append(OutFix4, UnusedFix4, OutMid),
UnusedFix4 = [],
UnusedGoals=UnusedR1,
OutGoals = OutFix4.

```

The technique is similar to a Hidden Accumulator Grammar (Tarau, Dahl, and Fall, 1995).

If this tactic is to be applied anywhere but as a component of a parallel composition (e.g. as a ‘top-level’ tactic), the compiler can determine that *UnusedGoals* must be empty. By carefully analysing arities (easy since this example is fully deterministic) the above can be simplified to:

```

InGoals = [InR1],
proof_step(R1, InR1, [OutGoal1,InR2,InR3]),
proof_step(R2, InR2, [OutGoal2]),
proof_step(R3, InR3, [OutGoal3,OutGoal4]),
OutGoals = [OutGoal1,OutGoal2,OutGoal3,OutGoal4].

```

The arity of the whole tactic has been determined ($1 \rightarrow 4$), as have the arities of the rules ($R1 : 1 \rightarrow 3$; $R2 : 1 \rightarrow 1$; $R3 : 1 \rightarrow 2$). If any of these arities disagrees with information already available to the compiler, it can issue a warning that the tactic must fail. Use of *fix*, as in this example, often allows dramatic optimisations.

6.3 Proof navigation

To handle the extra Gumptree tactics for proof navigation (Section 4) requires further minor changes to the representation of tactics.

Deferring and selecting nodes To deal with deferred nodes, the tactic system needs an additional pair of parameters, representing the initial and final lists of deferred nodes. The list is initially empty, and all constructs apart from *defer* and *select* thread the list unchanged. To defer a node, it is moved from the input list to the deferred list; to select nodes, they are moved into the current goal list:

```

tactic(defer, In, Unused, Out, InDef, OutDef) :-
%% Defer the first input node; no output.
In = [Node|Unused],
Out = [],
append(InDef, [Node], OutDef).
tactic(select(Nodes), In, Unused, Out, InDef, OutDef) :-
%% separate available nodes (input or deferred) into
%% outputs (selected nodes) and new list of deferred
%% nodes (the remainder).
append(In, InDef, Available),
difference(Available, Nodes, OutDef),
Out = Nodes,
Unused = [].

```

Naming nodes The names for nodes are Prolog (meta-)variables. The command `name(Name)` simply unifies *Name* with the number of the input node. Unlike other Prolog systems, Qu-Prolog retains the names of metavariables throughout a session, so the user can refer to the node by the name of its metavariable. Ergo simply arranges to substitute the name of the metavariable for the node number whenever it appears in output.

6.4 Escape to Prolog

Within the simple, clean Gumtree language, it is possible to execute certain Prolog code safely. The code can directly affect neither the proof (the forthcoming module system for Qu-Prolog will enforce this) nor the goal sequences (since these parameters are hidden from Gumtree users). It *can* influence the flow of control in the tactic (by failing or by succeeding on backtracking), and can bind proof variables that are used in later proof steps. The syntax is

`{G}`

where *G* is a Prolog goal; the idea and syntax are borrowed from Definite Clause Grammars (Pereira and Warren, 1980), and it is implemented in the same way.

7 Example

This section illustrates the use of the Gumtree interface for Ergo 5. We will discuss proofs of the following theorem of classical propositional logic:

$$(A \vee (B \Leftrightarrow C)) \Leftrightarrow ((A \vee B) \Leftrightarrow (A \vee C)) .$$

We will use some classical sequent calculus rules as our primitive inference rules (Gentzen, 1969), with minor modifications because Ergo 5 uses sequents with a single consequent. Thus, the implication-left and not-left rules will be

$$\frac{\vdash p \vee r \quad q \vdash r}{p \Rightarrow q \vdash r} \textit{implies-left} \qquad \frac{\vdash q \vee p}{\neg p \vdash q} \textit{not-left} .$$

Clearly, this is not a particularly taxing activity for a proof tool, and decision procedures for propositional calculus are known which are far more efficient than those presented here. We use this logic merely as a readily-accessible example of the application of Gumtree.

The following sections will be expressed using Ergo 5's concrete syntax. The logical connectives are clear enough. The tactic combinators are mostly as they have been presented previously. The language's sequents appear as **hyps** `--->` **consequent**. Hypothesis lists are presented separately from the sequents themselves; see below. When discussing interaction with the tool, output is in **typewriter** font and user inputs are presented in **sans serif**.

7.1 Fully Interactive Proof

Having initiated an Ergo 5 proof with the proposition above as our goal, the system prints a list of open goals and prompts for a command to apply at the single open node (number 1):

```

1::: A or (B <=> C) <=> (A or B <=> A or C)
[1]:::

```

Clearly, our first step must be to apply the rule of equivalence introduction:

```
[1]::: rule(iff_intro).
```

and the system responds with

```

2::: (A or (B <=> C) => (A or B <=> A or C)) and
((A or B <=> A or C) => A or (B <=> C))
[2]:::

```

The next major connective is the conjunction, so we apply and-introduction, but we notice that this will give two subgoals, each of which is an implication, so we may save time by following the and-introduction with an implication-introduction in each of the subgoals:

```

[2]::: rule(and_intro);every(rule(implies_intro)).
hyp 1::: A or (B <=> C)
hyp 2::: A or B <=> A or C
7::: hyp=[1] ---> A or B <=> A or C
8::: hyp=[2] ---> A or (B <=> C)
[7, 8]:::

```

Now the response is considerably more complicated. Firstly, we have a listing of hypotheses which appear in some of the following sequents, then a list of open subgoals. Finally, the prompt now reflects the fact that there are two ‘current’ subgoals.

The two goals are quite different in form, so we will deal with them separately. Goal 7 is similar to the goal we started with, and, recognising that the same pattern of reasoning may be useful repeatedly, we define a tactic:

```
[7, 8]::: tactic local === rule(iff_intro);rule(and_intro);every(rule(implies_intro)).
```

The goal state and the prompt remain unchanged. Now we may apply the tactic to the goals we have chosen to work on (goal 7):

```

[7, 8]::: [7]::: local.
hyp 1::: A or (B <=> C)
hyp 2::: A or B <=> A or C
hyp 3::: A or B
hyp 4::: A or C
12::: hyp=[1, 3] ---> A or C
13::: hyp=[1, 4] ---> A or B
8::: hyp=[2] ---> A or (B <=> C)
[12, 13]:::

```

Notice that goal 8 remains open, but not part of the current context. The hypotheses may be simplified by application of the or-left rule,

```

[12, 13]::: every(rule(or_left)).
hyp 1::: A or (B <=> C)
hyp 2::: A or B <=> A or C
hyp 5::: A
hyp 6::: B
hyp 7::: A

```

```

hyp 8::: C
14::: hyp=[1, 5] ----> A or C
15::: hyp=[1, 6] ----> A or C
16::: hyp=[1, 7] ----> A or B
17::: hyp=[1, 8] ----> A or B
8::: hyp=[2] ----> A or (B <=> C)
[14, 15, 16, 17]:::

```

and some of the resulting subgoals discharged by the rule `or-intro-L` followed by the rule of assumption.

```

[14, 15, 16, 17]::: some(rule(or_intro_L);rule(assump)).
hyp 1::: A or (B <=> C)
hyp 2::: A or B <=> A or C
hyp 6::: B
hyp 8::: C
15::: hyp=[1, 6] ----> A or C
17::: hyp=[1, 8] ----> A or B
8::: hyp=[2] ----> A or (B <=> C)
[15, 17]:::

```

7.2 Increasing Automation

Clearly, this proof will take some time with this level of user intervention. We have used some of the general purpose tactics, but a little more programming can considerably reduce the effort required.

Restarting the proof, we may observe from the above that the tactic `local` may be applied several times. In fact, it would be a useful general-purpose replacement for the rule `iff_intro`. The proof may begin by applying `local` exhaustively.

```
[1]::: !(bfs(local))
```

The result is the same as the result of the sequence of steps taken above, up to the formation of the subgoals which were numbered 8, 12, and 13 above.

Concentrating on the subgoals 12 and 13, if we define a tactic which applies any of the left-hand-side (antecedent) inference rules, as appropriate,

```
tactic lefts == rule(or_left) | rule(iff_left) |
                rule(implies_left) | rule(and_left) |
                rule(not_left).
```

we may simplify all the antecedents at once, using `bfs(lefts)`. This produces twenty subgoals, all similar to:

```

hyp 5::: A
hyp 9::: A
hyp 29::: C
hyp 25::: B
hyp 30::: C
hyp 11::: A
16::: hyp=[5, 9] ----> A or C
36::: hyp=[5] ----> B or (C or (A or C))
37::: hyp=[5, 29] ----> C or (A or C)
38::: hyp=[5, 25] ----> B or (A or C)
39::: hyp=[5, 25, 30] ----> A or C

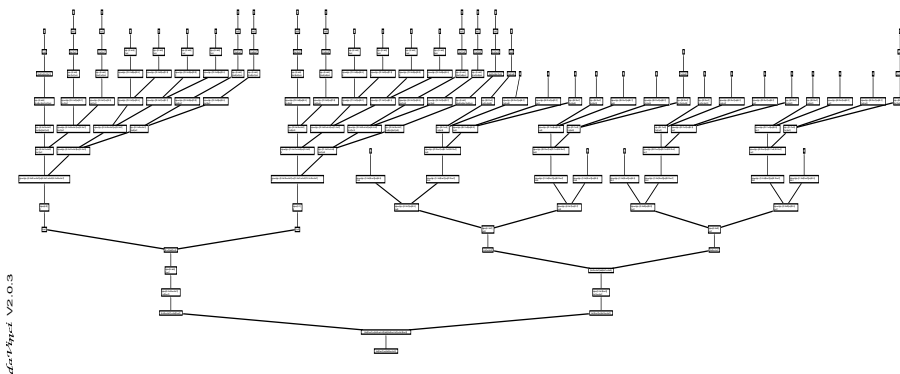
```

And each of these can be solved by some number of instances of `or_intro` followed by `assump`.

[16, 36, 37, 38, 39, 18, 40, 41, 42, 43, 20, 44, 45, 46, 47, 22, 48, 49, 50, 51] :: every(exhaust(rule(or_intro_L) | rule(or_intro_R));rule(assump))

The other half of the proof tree may be approached similarly.

The result is quite a complex formal proof; not one that we would wish to produce without a significant measure of automation.² Here it is in diagrammatic form, each arc denoting the application of one primitive rule. This graph is produced via Ergo's daVinci interface (Fröhlich, 1996).



7.3 Fully Automatic Propositional Proofs

The tactics above point the way towards a fully-general tactic for proving tautologies. In the most general Gentzen-style system, this is entirely straightforward: one applies all the introduction rules, and all the left-hand-side (antecedent) rules, and then applies assumption. With the single-conclusion sequents of Ergo 5, slightly more ingenuity is needed. Nothing is claimed about the efficiency of the following solution, but it demonstrates the style of tactics which we are able to write using Gumtree.

Difficulty arises because the rule for or-introduction must swap one of the disjuncts into the antecedent.³

Conversely, the rule for *not* in the antecedent must re-introduce the negated term as a disjunct in the consequent. These two are, then, virtually mutual inverses, and must not be applied indiscriminately if cycles are to be avoided. Therefore, we create a tactic which will apply any one of the left- or right-introduction rules, *but* will apply *not-left* only if the negated term is not atomic—that is, if one of the introduction rules may subsequently be applied to

²An alternative (or complement) to automation is a well-developed set of lemmas. For decidable theories like propositional calculus, full automation is a better solution.

³The rules which simply discard one of the disjuncts (as used previously) are not sufficiently general, as they cannot be used, for example in a proof of $\vdash p \vee \neg p$. Therefore, we use the rule *constructive-or* $\hat{=}$ **from**($\neg p \vdash q$) **infer** ($\vdash p \vee q$).

it.

```
tactic props === !(rule(and_intro)
| rule(not_intro)
| rule(constructive_or)
| rule(iff_intro)
| rule(implies_intro)
| rule(true)
| (rule(not_left) ; rule(constructive_or);
  !(rule(not_intro) | rule(implies_intro)|
  rule(iff_intro) | rule(and_intro)|
  rule(constructive_or)))
| rule(and_left)
| rule(or_left)
| rule(implies_left)
| rule(iff_left)
| rule(false_left)).
```

Following application of *bfs(props)* no propositional connectives will be left in any of the subgoals, except that some antecedents will be negated. It suffices, then, to apply the rule of assumption and a rule of contradiction to determine which subgoals may be discharged. The final tactic, then, is

```
!(bfs(props)) ; some(rule(contrad) | rule(assump)) .
```

This tactic also has the benefit that if it is applied to a goal which is not a tautology it will simplify it into subgoals containing only atomic (non-propositional) terms.

7.4 Recording Proof Information

When the partial proof at Section 7.1 is saved, Ergo 5 uses the tactic language to record the proof. The proof record is a tactic whose structure exactly reflects the structure of the proof tree: the proof tree nodes are applications of **rule**; sequential composition connects the outputs of one rule to the input of the next rules; and parallel compositions represent the branches that occur when a rule has multiple premisses. Interspersed with rule applications are **annotate** commands, which are automatically generated by the user interface to annotate discharged nodes with the user-level command that discharged the node.

The annotations in the proof representation can be used to automatically generate a proof script, which is a cleaned-up version of the commands typed at the user interface:

```
prove(A or (B <=> C) <=> (A or B <=> A or C)).
rule(iff_intro).
rule(and_intro);every(rule(implies_intro)).
names([N_2,N_3]).
N_2:: local.
names([N_4,N_5]).
[N_4, N_5]::
every(rule(or_left)).
names([N_6, N_7, N_8, N_9]).
```

```
[N_6, N_7, N_8, N_9]:::
  some(rule(or_intro_L);rule(assump)).
```

Since we did not explicitly name nodes in this proof, the proof script generator has invented names (N_i) wherever the proof branched, to make it easier to trace the flow of proof terms through the script.

Where the user invoked high-level tactics, and where generality (and alternation) were present in the user commands, these are seen in the proof script too. By contrast, another presentation of the proof eliminates the annotations and displays just the lowest-level rule applications:

```
rule(iff_intro);
rule(and_intro);
(
  rule(implies_intro);
  rule(iff_intro);
  rule(and_intro);
  (
    rule(implies_intro);
    rule(or_left);
    (
      rule(or_intro_L); rule(assump)
    ||
      one
    )
  ||
    rule(implies_intro);
    rule(or_left);
    (
      rule(or_intro_L); rule(assump)
    ||
      one
    )
  )
||
  rule(implies_intro);
  one
)
```

This presentation yields a tactic that is closely fitted to the goal at hand, and will not normally be more widely applicable.

Both presentations support reuse of proofs. The proof script can be used to reapply the proof steps, one command at a time; it may freely be edited, for example to adapt it to a related but different proof. The lower level presentation is more suited to fully automatic reapplication, for example when rebuilding a theory (perhaps after trivial changes in its axiomatic basis). It may also be suitable as input to an external proof checking tool, in case additional certification of proofs is needed. Tactics are free to add their own annotations to proof nodes (distinct from the command annotations inserted by the user interface), and it is foreseen that these may be used to provide other presentations of proofs, perhaps to support more powerful or flexible facilities for reuse.

8 Discussion and Conclusion

This paper has described the Gumtree interface for Ergo 5. We have been careful to make a single language be the basis of the tactic programming mechanism, the interface, and the tool’s proof storage and presentation. Certain primitives are more useful in one of these situations, and less so in the others. By presenting the user with a single language, we simplify the process of learning to use the tool and maximise the opportunities for reuse. Storing both the proof tree and the commands used to create it in a single structure gives us the greatest possible opportunity to reconstruct the proof—either in the presence of changes to the definitions of the tactics used to construct the proof, or of changes to the primitive (or derived) rules used in the proof.

We have been careful to preserve *Angel*’s simple axiomatic semantics for the tactics in Gumtree. We find this to be an aid to understanding which will in turn be an aid in constructing correct⁴ tactics. Our early experiences confirm that this is a useful paradigm. Gumtree adapts and extends the notion of *parallel* composition of tactics, and we expect to be able to give an axiomatic semantics for this. We believe that this is the first full implementation of a theorem-proving system with a tactic language which has an independent semantics; that is, one whose semantics is not given operationally.

Related Work Many systems implement tactics with broadly the same semantics as *Angel*. In particular, much of the language described here may be regarded as a subset of Isabelle’s tactic language (Paulson, 1989). Isabelle does not take the general approach to subgoals which we have presented, but instead, designating subgoals by positive integers, it provides tacticals for combining values of type `int->tactic`. These tacticals allow effects similar to our *every*, *some*, etc. Support is provided for treating subgoals as a stack; our approach again appears to generalise this.

Closer to the system described here is the work of Felty (1993) on tactics in logic programming languages. She describes the common tactic combinators in much the same way as we have, but again adopts a different approach to parallel goals (branching trees). She has the tactic `maptac`, which applies a tactic over a structured (collection of) goal(s), permitting a greater variety of structure than the simple list goals we permit here. `maptac` applies only a single tactic, however, and does not offer the structuring permitted by our parallel operator.

Future Work A separate report will consider how the laws of *Angel* are preserved and extended in the Gumtree implementation. Further generalisation of the denotational semantics of *Angel* using monads (Moggi, 1989) will assist in this. As we develop application theories for Ergo 5, we plan to write Gumtree versions of relevant proof procedures, expecting the semantics of *Angel* to assist in validating these procedures. We are also developing Gumtree tactics for window inference (Robinson and Staples, 1993), so that Ergo 5 can be used in a style similar to that of previous Ergo versions, which formed the basis of the program refinement tool (Carrington, Hayes, Nickson, Watson, and Welsh, 1996).

⁴Observe that correct tactics are those which perform as expected; tactics cannot, by their construction, be *unsound*.

Acknowledgements

We are grateful to several reviewers for their comments on an earlier draft of this paper.

Andrew Martin notes that the tactic *props* presented above was developed jointly with Stephen Brien. It is of interest because we wrote it for an earlier partial implementation of Angel. It is, therefore, the first truly portable Angel tactic.

References

- Carrington, D., Hayes, I., Nickson, R., Watson, G., and Welsh, J. (1996). A tool for developing correct programs by refinement, in He Jifeng (ed.), *Proc. BCS 7th Refinement Workshop, Bath, UK*, Electronic Workshops in Computing, Springer, pp. 1–17. Also available as Technical Report UQ-SVRC-95-49, Software Verification Research Centre, University of Queensland.
URL: <http://www.springer.co.uk/eWiC/Workshops/7RW.html>
- Cheng, A. S. K., Robinson, P. J., and Staples, J. (1991). Higher level meta programming in Qu-Prolog 3.0, *Proceedings of the 1991 International Conference for Logic Programming*, Paris.
- Felty, A. (1993). Implementing tactics and tacticals in a higher-order logic programming language, *Journal of Automated Reasoning* **11**: 43–81.
- Fröhlich, M. (1996). Real world applications of daVinci: Theorem prover Ergo.
URL: <http://www.informatik.uni-bremen.de/~davinci/applications/ergo.html>
- Gentzen, G. (1969). Investigations into logical deduction, in M. E. Szabo (ed.), *The Collected Papers of Gerhard Gentzen*, North-Holland, pp. 68–131.
- Gordon, M. J. C., Milner, R., and Wadsworth, C. P. (1979). *Edinburgh LCF: A Mechanised Logic of Computation*, Vol. 78 of *LNCS*, Springer-Verlag.
- Martin, A. (1994). *Machine-Assisted Theorem-Proving for Software Engineering*, D.Phil. thesis, University of Oxford. Also available as Technical Monograph PRG-121, ISBN 0-902928-95-3, Oxford University Computing Laboratory Wolfson Building, Parks Road, Oxford, OX1 3QD, UK.
- Martin, A., Nickson, R., and Utting, M. (1997). Improving Angel's parallel operator: Gumtree's approach, *Technical Report 97-15*, Software Verification Research Centre, The University of Queensland, QLD 4072, Australia. To appear.
- Martin, A. P., Gardiner, P. H. B., and Woodcock, J. C. P. (1996). A tactic calculus, *Formal Aspects of Computing* **8**(4): 479–489. An abridged version appears in the journal; the full version is available at the Formal Aspects of Computing FTP site, <ftp://ftp.cs.man.ac.uk/pub/fac>.
- Moggi, E. (1989). Computational lambda-calculus and monads, *Proceedings Fourth Annual Symposium on Logic in Computer Science*, IEEE Computer Society Press, Washington, D.C.

- Nickson, R. and Hayes, I. (1996). Supporting contexts in program refinement, *Technical Report 96-29*, Software Verification Research Centre, Department of Computer Science, The University of Queensland. Accepted for publication in *Science of Computer Programming*.
- Paulson, L. C. (1987). *Logic and Computation—Interactive Proof with Cambridge LCF*, Cambridge University Press.
- Paulson, L. C. (1989). The foundation of a generic theorem prover, *Journal of Automated Reasoning* **5**: 363–397. Also University of Cambridge Computer Laboratory Technical Report No. 130.
- Pereira, F. C. N. and Warren, D. H. D. (1980). Definite clause grammars for language analysis – a survey of the formalism and a comparison with augmented transition networks, *Artificial Intelligence* **13**: 231–278.
- Robinson, P. and Hagen, R. (1997). Qu-Prolog 4.2 reference manual, *Technical Report 97-11*, Software Verification Research Centre, The University of Queensland.
- Robinson, P. and Staples, J. (1993). Formalizing a hierarchical structure of practical mathematical reasoning, *Journal of Logic and Computation* **3**(1): 47–61.
- Tarau, P., Dahl, V., and Fall, A. (1995). Backtrackable state with linear assumptions, continuations and hidden accumulator grammars, *Technical Report 95-2*, Département d’Informatique, Université de Moncton.
URL: <http://clement.info.umoncton.ca/html/state.html>
- Utting, M. (1996). An architecture for a unified refinement/proof tool, *Fifth Australasian Refinement Workshop*, Software Verification Research Centre, The University of Queensland.
- Vickers, T. (1990). An overview of a refinement editor, *Proceedings of the Fifth Australian Software Engineering Conference*, pp. 39–44.
- Whitwell, K. (1992). A tactical environment for an interactive theorem prover, *Technical Report 92-7*, Software Verification Research Centre, The University of Queensland.