A Generic Tableau Prover and its Integration with Isabelle

Lawrence C. Paulson
lcp@cl.cam.ac.uk

9 January 1998

Abstract

A generic tableau prover has been implemented and integrated with Isabelle [14]. It is based on leantap [3] but is much more complicated, with numerous modifications to allow it to reason with any supplied set of tableau rules. It has a higher-order syntax in order to support the binding operators of set theory; unification is first-order (extended for bound variables in obvious ways) instead of higher-order, for simplicity.

When a proof is found, it is returned to Isabelle as a list of tactics. Because Isabelle verifies the proof, the prover can cut corners for efficiency's sake without compromising soundness. For example, it knows almost nothing about types.

Contents

1 Introduction

Interactive proof tools obviously benefit from the addition of automatic proof procedures, provided they are well integrated. Interactive tools are popular because of their flexibility: they support expressive formalisms and large developments. The user must guide the proof, but would like to have straightforward subgoals proved automatically.

Isabelle [14] is an interactive theorem prover. Unusually, Isabelle is *generic*: it supports numerous formalisms, including higher-order logic (Isabelle/HOL), first-order logic, set theory (Isabelle/ZF), some modal logics and linear logic. This paper describes a new tableau prover and its integration with Isabelle.

I have previously [17] described `Fast_tac`, a tableaux-like proof tactic for Isabelle. `Fast_tac` automatically finds proofs that consist only of so-called obvious inferences [5, 21]. Crucially, the tactic is itself generic. It works in most of Isabelle's classical logics and reasons directly with user-defined primitives. New concepts from the application domain can be supported without the search-space explosion that would result from simply adding their definitions to the tableau.

`Fast_tac` is not really an integration between automatic and interactive tools because it runs on the same generalized Prolog engine that Isabelle uses for single-step inferences. Isabelle itself provides the automation. This engine is rather slow, in part because it performs higher-order unification [8]. To improve decisively over `Fast_tac`, I decided to code a separate tableau prover and arrange that only a successful proof (rather than the full search) went through Isabelle's engine.

This paper makes two contributions. First, it describes a generic tableau prover, listing the many differences between such a tool and a first-order prover. Second, it describes the prover's integration with Isabelle and how compatibility constraints were overcome.

Both aspects of the work are pragmatic. The goal was to improve upon Isabelle's performance on hard problems arising in domains such as the verification of cryptographic protocols [18]. Some well-known theoretical refinements turned out to have little measurable benefit. Even the most basic theoretical properties, soundness and completeness, are not paramount. Soundness is not essential because the final proof is given to Isabelle for checking, though we want few proofs to fail. Completeness in the semantic sense is obviously impossible for the strong theories considered here, which contain set theory and therefore arithmetic. Anyway, the user will interrupt the prover after a minute or so: completeness is hardly relevant to interactive tools.

Paper outline. The paper begins with a review of tableau methods and introduces the notion of generic tableau rules (§2). Then it outlines the

methods used to design the Isabelle's tableau prover (§3). Some points require detailed discussion: instantiation of variables (§4), Skolemization (§5), and types (§6). Several minor points are briefly outlined (§7), and integration of the prover with Isabelle is discussed (§8). A few statistics are presented (§9) and conclusions drawn (§10).

2 Generic Tableau Methods

As is well known, the tableau method operates on *branches*: lists of formulas, interpreted conjunctively. Tableau rules are of four types: α-rules, which divide a conjunctive formula into two parts on one branch, β-rules, which split a branch according to the two parts of a disjunctive formula, γ-rules, which instantiate a universal quantifier, and δ-rules, which Skolemize an existential quantifier. Here are examples of each type of rule, for first-order logic:

$$\begin{array}{cccc} \text{type } \alpha & \text{type } \beta & \text{type } \gamma & \text{type } \delta \\ \dfrac{\phi \wedge \psi}{\begin{array}{c}\phi\\ \psi\end{array}} & \dfrac{\phi \vee \psi}{\phi \mid \psi} & \dfrac{\forall x\, \phi(x)}{\phi(?t)} & \dfrac{\exists x\, \phi(x)}{\phi(\mathbf{s})} \end{array}$$

Generally, this paper identifies the formula $\neg\phi$ with the goal ϕ. The formula $\neg(\phi \wedge \psi)$ counts as being disjunctive. It has the α-rule

$$\frac{\neg(\phi \wedge \psi)}{\neg\phi \mid \neg\psi}\,,$$

which reduces the goal $\phi \wedge \psi$ to the two subgoals ϕ and ψ.

In the γ-rule (for $\forall$), the term that replaces x is written $?t$ to indicate that it is a meta-variable and can be updated during unification. [1] In the δ-rule (for $\exists$), the symbol $\mathbf{s}$ can be generated in alternative ways discussed below.

For all rule types except γ, the formula above the line can be deleted after applying the rule. Controlling application of γ-rules is a key problem in tableau theorem-proving. Another problem is how to organize the search: if closing a branch instantiates a variable, then that step might have to be undone later.

For first-order logic, tableau methods are much less powerful than resolution. Their advantage for interactive theorem proving is that they can be extended to reason directly in application domains. Take set theory as an

[1] The question mark is included for emphasis, but it could mislead. In the Isabelle representation of such rules, all free variables are treated in the same way. Even the formula ϕ, strictly speaking, should be written $?\phi$.

example:

$$\begin{array}{cccc} \text{type } \alpha & \text{type } \beta & \text{type } \gamma/\beta & \text{type } \delta/\alpha \\ \dfrac{t \in A \cap B}{\begin{array}{c} t \in A \\ t \in B \end{array}} & \dfrac{t \in A \cup B}{t \in A \mid t \in B} & \dfrac{A \subseteq B}{\neg(?x \in A) \mid ?x \in B} & \dfrac{\neg(A \subseteq B)}{\begin{array}{c} \mathbf{s} \in A \\ \neg(\mathbf{s} \in B) \end{array}} \end{array}$$

As a trivial indication of how generic tableau proving complicates matters, note that the tidy classification of rules into types α, β, γ and δ no longer holds. These rules can best be understood through the equivalences that reduce them to first-order logic:

$$\begin{aligned} t \in A \cap B &\iff t \in A \wedge t \in B \\ t \in A \cup B &\iff t \in A \vee t \in B \\ A \subseteq B &\iff \forall x\,(x \in A \to x \in B) \end{aligned}$$

Using special tableau rules is much more efficient than adding the definitions of $\cap$, $\cup$ and $\subseteq$ to the initial branch. A preliminary rewriting phase could eliminate these symbols, but many application domains cannot easily be reduced to first-order logic. Examples are those involving inductive definitions, such as the λ-calculus [12] and my models of security protocols [18].

Binding constructs such as $\bigcup_{x \in A} B(x)$ and $\{x \mid \phi(x)\}$ are common in set theory, which is as fundamental to Isabelle/HOL as it is to Isabelle/ZF. A generic tableau prover therefore needs a higher-order syntax: a syntax that includes the λ-calculus.

Leantap [3] uses negation normal form, but a generic tableau prover cannot easily use a normal form. The notion of normal form would depend upon the precise set of primitives being supported in a particular call to the prover.

Tableau rules can be expressed in Isabelle's meta-logic and given to tools such as `Fast_tac` for execution on Isabelle's proof engine [17]. To improve upon `Fast_tac`, I have written a generic tableau prover from scratch. It is independent of Isabelle's proof mechanisms, but its design is constrained by the requirements of integration and compatibility. The new tactic (which is called `Blast_tac`) must let Isabelle check the proof it finds and must behave reasonably to users accustomed to `Fast_tac`.

3 Designing the Prover

As the starting point, I adopted leantap [3], a tableau prover consisting of five Prolog clauses. Although it is far from being the top-performing prover, it is much better than `Fast_tac` on standard benchmarks such as Pelletier's problems [20]. Leantap proves the first 46 problems in under half a second each, while `Fast_tac` takes several seconds for some of them and

cannot prove others at all. Leantap's simplicity and clarity make it a good framework on which to build other provers.

Several features of leantap still remain in `Blast_tac`.

- Depth-first iterative deepening [9] is now the search strategy of choice for such systems. `Fast_tac` can use depth-first search because of its incomplete treatment of γ-rules, which ensures termination but proves only *obvious* [5] theorems.

- The resource bounded by iterative deepening is mainly the number of γ-rule applications, but it includes other 'costs' of the proof. (For Stickel's Prolog Technology Theorem Prover [22], the resource is the number of subgoals allowed in a proof.)

- Formulas in a branch are considered in a stack discipline. If a rule adds the formula A to a branch, then A and the formulas derived from A will be expanded before any other formulas on that branch. The usual effect is to reduce A quickly to literals.

  (`Fast_tac` has a queue discipline due to the workings of Isabelle's proof engine. A new formula is expanded last rather than first. The stack/queue distinction refers not to the search strategy but to the method used to select the next formula for expansion.)

Extensive experimentation with problems in first-order logic (largely Pelletier's) and ZF set theory [13, 15] suggested extensions to leantap's strategy. Though many of these extensions were driven by the need for `Blast_tac` to handle generic rules, most of them can be explained in terms of first-order logic, which may be clearer to some readers.

Deferral of γ-Formulas. The stack discipline works well—with some exceptions. When a γ-formula such as $\forall x\, A$ is the next formula to expand, it is deferred until formulas of all other types have first been expanded. Great care is taken to preserve the stack discipline even with this exception. The deferred γ-formula does not go the end of a global queue but merely after all other formulas in the present group arising from some expansion: there is a stack of queues.

A formula such as $\forall x\, A$ can make a branch grow without limit, so delaying its expansion can realize dramatic savings. Another exception to the stack discipline concerns transitivity rules (see §7).

Retention of γ-Formulas. When a γ-formula such as $\forall xy\, A$ or $\forall x\, (B \to \forall y\, A)$ is expanded, the outer formula must be retained. However, the inner γ-formula (which is $\forall y\, A$ in both of the examples above) can be discarded with no loss of completeness: equivalent copies of it can be generated from

the retained outer formula. This optimization, which can be extended to generic tableau rules, suppresses an explosion of redundant γ-formulas.

Note that $\forall x\,\exists z\,\forall y\,A$ requires retention of the subformula $\forall y\,A$ because each instance of it will contain a different term for z, generated by a δ-rule. So, the optimization works even better for provers that employ Skolemization or the δ^{++}-rule [2].

Rules that Close Branches. In a first-order tableau, the only way to close a branch is by unifying complementary literals. But in generic tableau theorem-proving, many rules can close branches. Here are three examples:

$$\underline{\neg(x = x)} \qquad \underline{0 = \mathsf{Suc}(n)} \qquad \underline{n < 0}$$

The first rule, reflexivity, accepts the goal formula $x = x$, while the other two recognize the given formulas as contradictory. The possibility of a rule's closing a branch complicates the treatment of backtracking. But it is more directional (and therefore more effective) than the approach of regarding $x = x$, $\neg(0 = \mathsf{Suc}(n))$ and $\neg(n < 0)$ as axioms that can be added to a branch at any time.

Search-Space Pruning. A limited form of intelligent backtracking takes place whenever a branch is closed. The prover is coded in ML [16] and manages backtracking with an explicit list of choice points. Closing a branch typically proves a number of parent subgoals. When a branch is closed, a tree-pruning function looks backward to identify those subgoals and to determine if their proofs involve instantiating variables present in branches that are still open. All choice points for parent goals free of such clashes are removed.

4 Inferences that Instantiate Variables

A first-order tableau prover only instantiates variables when a branch is closed. A generic tableau prover may be supplied rules that instantiate variables. Dealing with this possibility is a major source of complications.

For example, suppose we are working in set theory and have the 'big union' operation $\bigcup C$, defined to satisfy

$$t \in \bigcup C \iff \exists X\,(t \in X \wedge X \in C).$$

Suppose we have tableau rules for $\bigcup C$ as well as for binary union $(A \cup B)$ and intersection $(A \cap B)$:

$$\frac{\neg(t \in \bigcup C)}{\begin{array}{c}\neg(t \in ?X)\\ \neg(?X \in C)\end{array}} \qquad \frac{\neg(t \in A \cup B)}{\begin{array}{c}\neg(t \in A)\\ \neg(t \in B)\end{array}} \qquad \frac{\neg(t \in A \cap B)}{\neg(t \in A) \mid \neg(t \in B)}$$

The rule for $\bigcup C$ says that to show $t \in \bigcup C$ it suffices to show $t \in ?X$ and $?X \in C$, where $?X$ may get instantiated to any term. The other two rules, the duals of those shown in §2, handle $t \in A \cup B$ and $t \in A \cap B$ as goal formulas.

Given the goal $t \in \bigcup C$, the first rule adds $\neg(t \in ?X)$ and $\neg(?X \in C)$ to the branch. We now have the ingredients of disaster, because the new goal $t \in ?X$ matches all three rules given above, and in a realistic environment, dozens of rules. If the $A \cap B$ rule is chosen next, then $?X$ will be instantiated to $?A_1 \cap ?B_1$, and the new goals will be the equally disastrous $t \in ?A_1$ and $t \in ?B_1$.

A partial solution is to replace the $\bigcup C$ rule shown above by one that creates goals in the opposite order: first $?X \in C$, then $t \in ?X$. The search for solutions to the first goal will be acceptably constrained, and proving that goal will probably instantiate $?X$, constraining the second goal. In general, however, we must be prepared to handle unconstrained subgoals like $t \in ?X$.

If a rule instantiates variables while closing the branch, no special treatment is necessary. But if the rule does not close the branch, then something must be done to prevent runaway instantiations. The search already imposes a bound on the number of expansions of γ-formulas; the same bound can control variable instantiations.

The precise handling of this bound is problematical. Decreasing the bound by one prevents looping, but allows goals such as $t \in ?X$ to swamp the search space. We need a greater penalty, depending upon the number n of rules that are applicable to the formula being expanded. (Indexing of rules, needed anyway for efficiency, can provide this information cheaply.) If the penalty is too great, many theorems will not be proved. The penalty used at present is $\log_4 n$, determined after extensive experimentation; it is a compromise between banning instantiation altogether and allowing it freely.

5 Skolemization and δ-Rules

Unlike leantap, my tableau prover cannot easily use Skolemization (which complicates proof reconstruction), so δ-rules are necessary. A typical δ-rule replaces the formula $\exists x\, \phi(x)$ on a branch by $\phi(\mathbf{s})$. The question is, what precisely is $\mathbf{s}$?

The standard answer is that $\mathbf{s}$ is a Skolem term constructed in the usual manner by applying a fresh Skolem function to all the branch's free variables. Although it may resemble Skolemizing the formula before calling the prover, this form of δ-rule poses no problems for proof reconstruction: it corresponds (in the example above) to the standard $\exists$-elimination rule.

Two liberalizations of the δ-rule have been published. The first [6] is to apply the Skolem function only to the variables free in the formula $\exists x\, \phi(x)$

rather than to all those of the branch; having fewer variables lets the Skolem term take part in more successful unifications. The second liberalization [2] goes further by allowing $\exists x\,\phi$ and $\exists x\,\psi$ to share one Skolem function provided the two formulas are identical up to variable renaming. (This includes the important case where both have arisen through expansion of a formula such as $\forall y\,\exists x\,\phi$.) Both liberalizations can be understood intuitively as the replacement of existential variables by Hilbert ϵ-terms, changing $\exists x\,\phi(x)$ to $\phi(\epsilon x\,\phi(x))$.

Experiments, mainly on Pelletier's examples [20], found that the first example made little difference in practice. One proof (problem 43) got shorter, but the runtime actually rose due to the increased branching factor! I did not investigate the second liberalization. I did find that Skolemization made a big improvement in the proof of a problem discussed by Lifschitz [10].

With Skolemization and the liberalized δ-rules, the obvious method of proof reconstruction involves manipulating ϵ-terms in Isabelle. This method would be prohibitively inefficient: the terms are large.

Finally, there is the question of prenexing. Baaz and Leitsch [1] have proved the folklore result that prenexing a formula makes its proof longer. Quantifiers should be pushed in, not pulled out. Isabelle's tableau prover can expect that task to have been done for it beforehand. Users normally apply simplification first, and the simplifier is equipped with rewrite rules such as

$$\forall x\,(\phi(x) \vee \psi) \leftrightarrow (\forall x\,\phi(x)) \vee \psi$$

and even

$$\Big(\bigcup_{x\in A} A(x) \cap B\Big) = \Big(\bigcup_{x\in A} A(x)\Big) \cap B.$$

Omitted is the distributive law

$$\forall x\,(\phi(x) \wedge \psi(x)) \leftrightarrow (\forall x\,\phi(x)) \wedge (\forall x\,\psi(x))$$

and its $\exists$-$\vee$ dual, which by increasing the number of Skolem functions can sometimes be harmful [1].

6 Types and Overloading

Isabelle's framework for specifying formalisms is typed. Some of its logics, such as ZF set theory [19], are essentially typeless: types serve only to prevent absurd expressions like $\bigcup(\bigcup)$. Other logics, such as higher-order logic (HOL), are not only typed but provide polymorphism and overloading.

Omitting types helps make the tableau prover more efficient than Isabelle's proof engine. But HOL demands some knowledge of types. For

example, the equality $x = y$ can be treated in three different ways, depending on the type of x. Every equality satisfies the usual axioms such as transitivity. If x has type bool then the equality means 'x if and only if y.' If x has type $(\tau)\mathsf{set}$ then it is set equality, enjoying additional axioms such as extensionality; moreover, τ is the type of the set's elements, determining which rules apply to them. Each of these three might apply to the goal $t = u$, depending upon types:

$$\begin{array}{ccc} \text{transitivity} & \text{iff introduction} & \text{extensionality} \\ \dfrac{\neg(a = c)}{\neg(a = ?b) \mid \neg(?b = c)} & \dfrac{\neg(\phi = \psi)}{\begin{array}{c|c} \phi & \psi \\ \neg\psi & \neg\phi \end{array}} & \dfrac{\neg(A = B)}{\neg(A \subseteq B) \mid \neg(B \subseteq A)} \end{array}$$

The collection of rules supplied to the prover can be different at each invocation, so it needs a general strategy for using type information.

Storing the types of all constants is prohibitively expensive, and storing only the types of overloaded constants is insufficient. Consider the constant insert, defined to satisfy

$$(x \in \mathsf{insert}\ y\, A) = (x = y \vee x \in A).$$

One tableau rule resembles the corresponding α-rule for disjunction:

$$\frac{\neg(x \in \mathsf{insert}\ y\, A)}{\begin{array}{c} \neg(x = y) \\ \neg(x \in A) \end{array}}$$

Because of Isabelle's polymorphism, static analysis of this rule cannot reveal the type of x. After applying the rule, if x turns out to have a set type, then the prover will not know to try the set equality rule on the goal formula $x = y$. Overloading resolution by static type analysis is workable—I used it in an early version of `Blast_tac`—but it cannot find proofs involving such inferences.

The solution I have adopted is to store some types dynamically: during the proof search. The prover can be instructed to record the types of certain constants. These should include not just the overloaded constants but other basic, polymorphic constants such as $\in$. To keep the prover simple, it represents Isabelle types using its data structure for terms; unification propagates type constraints.

7 Minor Points

Now, let us briefly consider some additional features of the prover.

Unification. Isabelle uses higher-order unification [8]. For efficiency, the tableau prover uses first-order unification. Bound variables, represented by de Bruijn indices, are handled in the obvious way: a bound variable unifies only with itself. Unification between two λ-abstractions fails. Detecting that $\lambda x.x$ is not unifiable with $\lambda x.?y$ would complicate the algorithm and is not necessary. However, β-reductions (from $(\lambda x.M)N$ to $M[N/x]$) are performed so that quantifier rules will work. Redundant λ-abstractions are erased using η-reduction, which takes $(\lambda x.Mx)$ to M provided x is not free in M.

Closing Branches. For simplicity, leantap attempts to close a branch only when it can do nothing else, and it only compares the current formula (which must be a literal) with existing literals. `Blast_tac`'s prover always attempts to close the branch using the current formula before doing anything else with it. It searches for a complementary formula even among the unexpanded formulas. Branches are closed more quickly than otherwise would be possible, particularly if the closing formula is compound.

If the current formula can neither close the branch nor be expanded, then it is moved to a list of literals, as in leantap. There is an important difference, however. In generic tableau proof, the notion of literal depends upon what rules are supplied. For example, $x \in A \cup B$ is not a literal if the corresponding rule from §2 is available.

Equality. The treatment of equality is simple and incomplete. Suitable assumptions of the form $\mathbf{s} = t$ are deleted, replacing $\mathbf{s}$ by t throughout the branch. Which assumptions are suitable?

Typically, $\mathbf{s}$ is a Skolem term, introduced by a δ-rule. We require that $\mathbf{s}$ does not occur in t—otherwise the substitution will not eliminate $\mathbf{s}$—but this condition is not strong enough. If the branch contains the formulas $\mathbf{s} = ?y$ and $\forall z\, z \neq \mathsf{Suc}\, z$, then we should hope eventually to close the branch with $?y \mapsto \mathsf{Suc}\, \mathbf{s}$. The equality assumption must not be deleted.

If $\mathbf{s}$ is a Skolem constant, then $\mathbf{s} = ?y$ is not suitable for substitution because $?y$ might later be instantiated with a term containing $\mathbf{s}$. If $\mathbf{s}$ is the Skolem term $\mathbf{f}(x_1, \ldots, x_m)$, then t may contain only the variables x_1, $\ldots$, x_m, because those variables cannot be instantiated with $\mathbf{s}$. (In Isabelle's meta-logic, which does not use Skolemization, the corresponding condition is literally that $\mathbf{s}$ does not occur in t.)

Any literals affected by the substitution are moved back to the list of unexpanded literals for reconsideration. For example, after eliminating the equality $\mathbf{s} = A \cup B$, the literal $x \in \mathbf{s}$ becomes the compound formula $x \in A \cup B$. The ensuing rearrangement of the formulas interferes with proof reconstruction, occasionally making it fail.

Undoable rules. Classical tableau rules are purely analytic: each rule captures the full logical content of the expanded connective. Backtracking from a rule application is never necessary. Choice points arise only when closing a branch with unification.

We already have to allow backtracking for rule applications that instantiate a variable (which cannot happen in a first-order tableau), so it is easy to allow backtracking for other reasons. Isabelle has a concept of *unsafe* rule: those where backtracking is suitable because the conclusion is weaker than the premises.[2] A generic tableau prover can expect to be supplied rules that need backtracking. For example, if our problem domain involves transitive closure, we might supply three rules:

$$\frac{\neg(x\,R^*\,x)}{} \qquad \frac{\neg(x\,R^*\,y)}{\neg(x\,R\,y)} \qquad \frac{\neg(x\,R^*\,z)}{\neg(x\,R^*\,?y) \mid \neg(?y\,R^*\,z)}$$

These rules should be tried in the order shown, trying to prove the goal $a\,R^*\,b$ first by reflexivity, then by reduction to $a\,R\,b$ and only as a last resort by transitivity.

A rule application may be undone if other unifiable rules exist (as in our example), if it instantiates variables, or if the inference does not introduce new variables (and thus is not a true γ-rule). The overall effect is to allow backward chaining over a variety of types of rule. Equally important, it makes `Blast_tac` treat such rules similarly to `Fast_tac`.

Transitivity. Transitivity and similar rules are notoriously hard to deal with, but some proofs require them. They are unfortunately incompatible with the stack discipline of §3. If the current subgoal is $a\,R\,c$, then expanding by the transitivity rule shown above replaces it by the subgoals $a\,R\,?b$ and $?b\,R\,c$. Expanding these subgoals before the rest of the branch, as the stack dictates, could permanently exclude the latter from consideration as the R^* subgoals multiply. The prover checks whether the conclusion of the current rule matches any premises of the same rule, and if so arranges that those premises are expanded after the rest of the branch. Mutually recursive rules could defeat this simple heuristic, but they seem not to occur in practice.

8 Integration with Isabelle

The purpose of this tableau prover is to improve the degree of automation available to Isabelle users. Because the prover is generic, integration has two aspects:

- translation of Isabelle rules to tableau rules

[2] This concept was originally used to allow backtracking over γ-rules in `Fast_tac`, which does not retain the universal formula.

- translation of tableau proofs to Isabelle proofs

The translation of Isabelle rules has to contend with different treatments of eigenvariables in quantifiers—Isabelle does not employ Skolemization—and it discards virtually all type information. Both of these complications could render the resulting tableau proof incorrect; Isabelle must check the tableau proof to ensure soundness.

8.1 The Translation of Isabelle Rules

The translation from Isabelle rules to tableau rules is largely straightforward. I have elsewhere described the connection between Isabelle and tableau rules [17], and more details are available in the documentation [14, Chap. 14].

Some further points can be seen in the treatment of the natural deduction rule $(\forall I)$. This inference rule takes the premise ϕ to the conclusion $\forall x\, \phi$, subject to the usual proviso that x is not free in the assumptions. The rule is represented in Isabelle as a generalized Horn clause:

$$\left(\bigwedge x.\ \mathsf{Tr}(\phi x)\right) \Longrightarrow \mathsf{Tr}(\mathsf{All}\,\phi)$$

Here, $\bigwedge$ is Isabelle's meta-universal quantifier, Tr is a meta-level predicate to recognize true formulas, and All is the constant for the first-order universal quantifier. Isabelle's treatment of Skolemization (which is its dual [11]) must be converted to conventional Skolemization. The bound variable (x) in the premise is replaced by a Skolem term (call it $\mathbf{s}$) containing the free variables of the current branch. (Thus, it is a δ-rule of the tableau calculus.)

The conclusion of the rule, namely $\mathsf{Tr}(\mathsf{All}\,\phi)$, involves no variable-binding. Unification will probably instantiate ϕ (which is a function variable) to a λ-term representing some quantified formula. Although the generic prover does not implement higher-order unification, it can perform the β-reduction needed to replace the quantified variable by the Skolem term $\mathbf{s}$.

One difference between Isabelle rules and tableau rules is that the former can have only one goal formula, while the latter can have many. Isabelle represents multiple goal formulas as negative formulas, but (for a number of reasons) retains its natural deduction concept of goal formula too. Therefore, the rule $(\forall I)$ becomes two tableau rules:

$$\frac{\mathsf{Goal}(\mathsf{All}\,\phi)}{\mathsf{Goal}(\phi(\mathbf{s}))} \qquad \frac{\neg(\mathsf{All}\,\phi)}{\neg(\phi(\mathbf{s}))}$$

If a rule is to generate multiple goal formulas, then all but one of them must be negative.

`Blast_tac` also has to translate an Isabelle proof state to an initial tableau. This process resembles that of translating rules and on rare occasions can fail.

8.2 Giving Tableau Proofs to Isabelle

Translating a proof found by one tool for checking by another is an old idea. For example, John Harrison has shown how to translate OBDD derivations into proofs acceptable to the HOL system [7]. Translating proofs can be difficult and slow, so I have taken pains to make the tableau proofs closely resemble Isabelle proofs. The main advantage of the tableau prover over Isabelle is its greater search speed.

Of the many minor differences between the tableau proof style and Isabelle's style, only one could not be settled straightforwardly. A key heuristic of the tableau approach is to expand formulas using a stack, while Isabelle's normal mode would yield a queue; I had add an Isabelle primitive for reordering a subgoal's assumptions.

Proof reconstruction is simple in concept, if complicated in its details. During its search, each time the tableau prover proposes some inference, it records the corresponding Isabelle tactic. If a proof is found, then the full list of tactics is applied to the original Isabelle subgoal. Isabelle's tactic mechanism supports backtracking, but for efficiency reasons, most of the tactics returned by `Blast_tac` do not offer alternative outcomes. Backtracking is supposed to occur during the search, not during the proof reconstruction.

Isabelle's proof engine repeats the unifications originally found by the tableau prover. An attempt to deliver those instantiations to Isabelle yielded no speed-up; the tableau prover only finds first-order unifiers, which Isabelle can reconstruct easily.

Proof reconstruction occasionally fails. After printing a message, the tableau prover backtracks, but it cannot always recover. It prunes its search space under the presupposition that its inferences are legal. Pruning is essential for efficiency, but it reduces the chances of finding alternative proofs.

The usual cause of failed proof reconstruction is that the tableau and Isabelle proofs have somehow diverged. Typically, they disagree on the order in which formulas appear on a branch. Proof reconstruction occasionally fails because the tableau proof is unsound. Overloading, when a rule expects a constant to have a certain type, is the only cause of unsoundness that I know of. The prover knows little about types, but its mechanisms cope with all but the subtlest uses of overloading.

9 Results

This project must be judged on the pragmatic grounds for which it was undertaken: to augment Isabelle's existing tools with something similar but more powerful. The classical reasoner already provided `Fast_tac` (based on depth-first search) and `Best_tac` (based on best-first search). `Blast_tac` indeed outperforms them in most cases: it is faster and proves many theorems that they cannot.

	Search			*Verify*		`Fast_tac`
Problem	depth	branches	time	tactics	time	time
24	4	16	60	52	40	220
26	3	17	30	43	40	430
28	3	7	20	29	30	140
34	7	10	200	431	1120	FAILED
38	4	30	50	10	130	900
43	5	24	50	48	60	FAILED
46	7	15	610	48	50	2,090
52	7	63	110	68	290	1,490
62	1	17	10	46	40	130
Halting II	7	2,015	1,960	1,086	4,540	226,000
Union-image	3	12	40	40	70	370
Inter-image	3	12	50	36	50	$830^{\dagger}$
Singleton I	4	110	320	19	20	∞
Singleton II	4	108	310	19	20	∞

runtimes given in milliseconds
∞ = still running after 5 minutes
$\dagger$ = using `Best_tac`; would be ∞ for `Fast_tac`

Table 1: `Blast_tac` Compared With `Fast_tac`

Table 1 compares the performance of `Blast_tac` and `Fast_tac` on several examples.[3] *Search* refers to the tableau prover, while *Verify* refers to the Isabelle reconstruction of the proof. The numbered problems are those of Pelletier [20]; problem 34 is also known as Andrews' Challenge. Halting II refers to the halting problem presented by Dafa [4].[4] The last four problems are formulated in the set theory of Isabelle/HOL.

$$\bigcup_{x\in C}\Big(A(x)\cup B(x)\Big) = \bigcup(A\text{“}C)\cup\bigcup(B\text{“}C) \qquad \text{Union-image}$$

$$\bigcap_{x\in C}\Big(A(x)\cap B(x)\Big) = \bigcap(A\text{“}C)\cap\bigcap(B\text{“}C) \qquad \text{Inter-image}$$

$$\forall_{x\in S}\ \forall_{y\in S}\ x\subseteq y \to \exists z\ S\subseteq\{z\} \qquad \text{Singleton I}$$

$$\forall_{x\in S}\ \bigcup(S)\subseteq x \to \exists z\ S\subseteq\{z\} \qquad \text{Singleton II}$$

Here " is the image operator, which satisfies $y\in f\text{“}A \iff \exists_{x\in A}\ y=f(x)$.

Proof reconstruction time often exceeds search time, especially when the tactic proof is long. Examples include problem 34 and Halting II, and there are instances throughout the Isabelle proof scripts.

[3]Benchmarks run on a 300Mhz Pentium Pro running Linux. Isabelle was compiled using Standard ML of New Jersey, version 109.32.

[4]This theorem is large rather than deep: even `Fast_tac` can prove it. Dafa's short proof doubtless takes advantage of its repetition of large subformulas.

The set theory problems are largely insensitive to the set of rules used, provided they cover all of set theory. However, the two Singleton problems are proved without the trivial rule for the universal set:

$$\underline{\neg(?x \in \mathsf{UNIV})}$$

Adding this rule makes the problems much harder. It probably closes too many branches, increasing the search space.

Benchmarks give a distorted picture. For most first-order problems, `Blast_tac` is overwhelmingly superior to `Fast_tac`, but the latter is sometimes faster because of its incomplete search strategy. First-order problems allow comparison with other systems, but are of little relevance to Isabelle. Set theory problems are more relevant. `Blast_tac` is most important in application domains such as security protocols [18], whose use of inductive definitions cannot easily be reduced to first-order logic. `Blast_tac` can handle recursive rules such as transitivity, which otherwise could force the user to write a detailed, single-step proof.

10 Conclusions

This work has two aspects, (1) as a contribution to tableau theorem proving and (2) as an extension to Isabelle. Regarding (1), a generic tableau prover is possible, but is much more complicated than a first-order prover. Leantap consists of five Prolog clauses; `Blast_tac` is around 1,300 lines of ML (or 45K bytes). Higher-order syntax is essential in a generic prover, and it is easily implemented. The integration with Isabelle causes many complications and restricts the use of refinements. A stand-alone generic prover could use liberalized δ-rules, β-rules that incorporate lemmas, etc. Another obvious area for improvement is equality handling.

Regarding (2), `Blast_tac` is certainly useful, though it is not a killer tool. Its complete treatment of quantifiers makes little difference in practice, which comes as a surprise. It makes some proofs significantly faster, and some proof scripts shorter. Most security protocol proofs [18] consist of calls to `Blast_tac` on the simplified subgoals arising from induction. Each subgoal corresponds to one protocol action, and typically is proved by one `Blast_tac` call, using relevant lemmas. The next level of automation could involve supplying most lemmas by default, so that users do not have to consider them. If `Blast_tac` can cope with the resulting search space, then proof scripts will become much simpler. Preliminary experiments suggest that this may be possible.

Acknowledgement. The research was funded by the EPSRC, grants GR/K77051 'Authentication Logics' and GR/K57381 'Mechanizing Temporal Reasoning.' Fabio Massacci commented on a draft.

References

[1] Baaz, M., Leitsch, A., On Skolemization and proof complexity, *Fundamenta Informaticae* **20** (1994), 353–379

[2] Beckert, B., Hähnle, R., Schmitt, P. H., The even more liberalized δ-rule in free variable semantic tableaux, In *Computational Logic and Proof Theory: Third Kurt Gödel Colloquium, KGC'93 Colloquium* (1993), G. Gottlob, A. Leitsch, D. Mundici, Eds., LNCS 713, Springer, pp. 108–119

[3] Beckert, B., Posegga, J., leanTAP: Lean tableau-based deduction, *Journal of Automated Reasoning* **15**, 3 (1995), 339–358

[4] Dafa, L., Unification algorithms for eliminating and introducing quantifiers in natural deduction automated theorem proving, *Journal of Automated Reasoning* **18**, 1 (1997), 105–134

[5] Davis, M., Obvious logical inferences, In *7th International Joint Conference on Artificial Intelligence (IJCAI '81)* (Aug. 1981), pp. 530–531

[6] Hähnle, R., Schmitt, P. H., The liberalized δ-rule in free variable semantic tableaux, *Journal of Automated Reasoning* **13**, 2 (Oct. 1994), 211–221

[7] Harrison, J., Binary decision diagrams as a HOL derived rule, *Computer Journal* **38**, 2 (1995), 162–170

[8] Huet, G. P., A unification algorithm for typed λ-calculus, *Theoretical Comput. Sci.* **1** (1975), 27–57

[9] Korf, R. E., Depth-first iterative-deepening: an optimal admissible tree search, *Artificial Intelligence* **27** (1985), 97–109

[10] Lifschitz, V., What is the inverse method?, *Journal of Automated Reasoning* **5**, 1 (1989), 1–23

[11] Miller, D., Unification under a mixed prefix, *Journal of Symbolic Computation* **14**, 4 (1992), 321–358

[12] Nipkow, T., More Church-Rosser proofs (in Isabelle/HOL), In *Automated Deduction — CADE-13 International Conference* (1996), M. McRobbie J. K. Slaney, Eds., LNAI 1104, Springer, pp. 733–747

[13] Paulson, L. C., Set theory for verification: I. From foundations to functions, *Journal of Automated Reasoning* **11**, 3 (1993), 353–389

[14] Paulson, L. C., *Isabelle: A Generic Theorem Prover*, Springer, 1994, LNCS 828

[15] Paulson, L. C., Set theory for verification: II. Induction and recursion, *Journal of Automated Reasoning* **15**, 2 (1995), 167–215

[16] Paulson, L. C., *ML for the Working Programmer*, 2nd ed., Cambridge University Press, 1996

[17] Paulson, L. C., Generic automatic proof tools, In *Automated Reasoning and its Applications: Essays in Honor of Larry Wos*, R. Veroff, Ed. MIT Press, 1997, ch. 3

[18] Paulson, L. C., Proving properties of security protocols by induction, In *10th Computer Security Foundations Workshop* (1997), IEEE Computer Society Press, pp. 70–83

[19] Paulson, L. C., Grąbczewski, K., Mechanizing set theory: Cardinal arithmetic and the axiom of choice, *Journal of Automated Reasoning* **17**, 3 (Dec. 1996), 291–323

[20] Pelletier, F. J., Seventy-five problems for testing automatic theorem provers, *Journal of Automated Reasoning* **2** (1986), 191–216, Errata, JAR 4 (1988), 235–236 and JAR 18 (1997), 135

[21] Rudnicki, P., Obvious inferences, *Journal of Automated Reasoning* **3**, 4 (1987), 383–393

[22] Stickel, M. E., A Prolog technology theorem prover: Implementation by an extended Prolog compiler, *Journal of Automated Reasoning* **4**, 4 (1988), 353–380