Journal of Universal Computer Science, vol. 5, no. 3 (1999), 73-87
submitted: 1/11/98, accepted: 22/2/99, appeared: 28/3/99 © Springer Pub. Co.

A Generic Tableau Prover and its Integration with Isabelle

Lawrence C. Paulson
lcp@cl.cam.ac.uk
Computer Laboratory, University of Cambridge, England

Abstract: A generic tableau prover has been implemented and integrated with Isabelle [Paulson, 1994]. Compared with classical first-order logic provers, it has numerous extensions that allow it to reason with any supplied set of tableau rules. It has a higher-order syntax in order to support user-defined binding operators, such as those of set theory. The unification algorithm is first-order instead of higher-order, but it includes modifications to handle bound variables.
The proof, when found, 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, the prover can use type information to guide the search without storing type information in full.
Categories: F.4, I.1

1 Introduction

Interactive proof tools are popular because of their flexibility: they support expressive formalisms and large developments. Users must guide the proof, but would like to have straightforward subgoals proved for them. Automatic proof procedures can help provided they are well integrated with the interactive facilities.

Isabelle [Paulson, 1994] 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. The prover handles higher-order syntax and accepts an arbitrary set of rules.

I have previously [Paulson, 1997a] described `Fast_tac`, a tableaux-like proof tactic for Isabelle. `Fast_tac` automatically finds proofs that consist only of so-called obvious inferences [Davis, 1981, Rudnicki, 1987]. 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 because it performs higher-order unification [Huet, 1975], handles backtracking using lazy lists, etc. 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 objective is to improve upon Isabelle's performance on hard problems arising in domains such as the verification of cryptographic protocols [Paulson, 1997b]. Some well-known refinements, such as generalized δ-rules, turn out to be unsuited to this application. 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.

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 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 formulæ, 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:

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

This paper identifies the formula $\neg\phi$ with the goal ϕ. For example, the goal of establishing the conjunction $\phi \wedge \psi$ is identified with the formula $\neg(\phi \wedge \psi)$. This formula is regarded as 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. Note that identifying goals with negative formulæ is only possible in classical logic, where we have $\neg\neg\phi \leftrightarrow \phi$; allowing a branch to hold more than one goal formula is essential for classical reasoning.

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. Limiting the expansion of γ-formulæ 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.

Tableau provers are less efficient than resolution provers. Their advantage for interactive proof is that they can be extended to reason directly in application domains. Take set theory, for example:

$$\begin{array}{cccc}
\text{type } \alpha & \text{type } \beta & \text{type } \gamma/\beta & \text{type } \delta/\alpha \\[1ex]
\dfrac{t \in A \cap B}{\begin{array}{c} t \in A \\ t \in B \end{array}} & \dfrac{t \in A \cup B}{t \in A \mid t \in B} & \dfrac{A \subseteq B}{\neg(?x \in A) \mid ?x \in B} & \dfrac{\neg(A \subseteq B)}{\begin{array}{c} \mathbf{s} \in \mathbf{A} \\ \neg(\mathbf{s} \in \mathbf{B}) \end{array}}
\end{array}$$

Generic tableau proving complicates matters; for a start, the tidy classification of rules into types α, β, γ and δ fails. 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 be reduced to first-order logic. 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.[2] A generic tableau prover therefore needs a higher-order syntax: a syntax that includes the typed λ-calculus.

Normal forms bring further complications. With generic theorem proving, the notion of normal form depends upon the precise set of primitives being used. This set can vary from one invocation to the next. Although many conventional tableau provers use a normal form, a generic one probably will not.

Finally, a format must be chosen for representing tableau rules. A more liberal format yields a broader notion of 'generic,' but makes implementation harder. In this case, the representation should be general enough to capture the sort of rules that can be expressed in Isabelle's meta-logic [Paulson, 1997a]. Such a rule has a fixed structure—for example, it has a fixed number of premises—but it is schematic, perhaps with higher-order variables. There is no Skolemization, but a rule can introduce eigenvariables. Such tableau rules can be given to tactics such as `Fast_tac` for execution on Isabelle's proof engine. Before use, the rules must themselves be proved in Isabelle, using tactics with existing rules.

---

[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 $?\phi$.

[2] With higher-order syntax, Zermelo-Fraenkel set theory consists of finitely many axioms. We no longer have to use the Bernays-Gödel axioms and can reason directly about expressions such as $\{x \in A \mid \phi(x)\}$.

To improve upon `Fast_tac`, I have written a new generic tableau prover. It is independent of Isabelle's proof mechanisms, but its design is constrained by the requirements of integration and compatibility:

– Isabelle must be able to verify (efficiently!) any claimed proof.
– To the user, the prover should simply behave like a more powerful version of `Fast_tac`, despite using different technology.

The new tactic is called `Blast_tac`.

3 Designing the Prover

As the starting point, I adopted lean$T^{A}P$ [Beckert and Posegga, 1995], 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 some standard benchmarks [Pelletier, 1986]. Lean$T^{A}P$ 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. My strategy was to code lean$T^{A}P$ in ML [Paulson, 1996] and to modify it as necessary.

Several features of lean$T^{A}P$ still remain in `Blast_tac`.

– Depth-first iterative deepening [Korf, 1985] 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* [Davis, 1981] theorems.
– The resource bounded by iterative deepening is mainly the number of γ-rule applications, but it includes other 'costs' of the proof. (For the Prolog Technology Theorem Prover [Stickel, 1988], the resource is the number of subgoals allowed in a proof.)
– Formulæ in a branch are considered in a stack discipline. If a rule adds the formula A to a branch, then A and the formulæ derived from A will be expanded before any other formulæ on that branch. The usual effect is to reduce A quickly to literals. `Fast_tac` considers formulæ in a queue discipline because Isabelle's proof engine normally works that way.

Extensive experimentation suggested extensions to lean$T^{A}P$'s strategy. Problems were drawn from first-order logic (largely Pelletier's) and from ZF set theory [Paulson, 1993, Paulson, 1995, Paulson and Grąbczewski, 1996]. 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-Formulæ

The stack discipline works well, with the exception of γ-formulæ. (Another exception concerns transitivity rules, see §7.) When a γ-formula such as $\forall x\, A$ is the next formula to expand, it is deferred until formulæ 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 to the end of a global queue but merely after all other formulæ in the present group arising from some expansion.

Retention of γ-Formulæ

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-formulæ.

Note that $\forall x\, \exists z\, \forall y\, A$ requires retaining 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 perform Skolemization during a preprocessing phase, eliminating existential quantifiers at the outset. The δ^{++}-rule [Beckert *et al.*, 1993] also avoids having to retain the subformula $\forall y\, A$.

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:

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

The first rule, reflexivity, accepts the goal formula $x = x$, while the other two recognize the given formulæ 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

The prover maintains an explicit list of choice points for backtracking. Closing a branch typically proves a number of parent subgoals. When a branch is closed, the prover deletes the remaining choice points for the solved parent goals provided their proofs do not instantiate variables present in other open branches.

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 additional 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

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

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

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

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 formulæ.

Given the goal $t \in \bigcup C$, the first rule splits the branch adding $\neg(t \in ?X)$ and $\neg(?X \in C)$. 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 (adopted in Isabelle) is for the user 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-formulæ; 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 lean$T^4\!P$, my tableau prover cannot begin by Skolemizing the formula to be proved. The proof reconstruction phase, where the tableau proof is given to Isabelle for checking, would become difficult or impossible, because Isabelle does not use Skolemization. Instead, the tableau prover uses δ-rules. 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.

Several liberalizations of the δ-rule have been published. The first proposal [Hähnle and Schmitt, 1994] 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 [Beckert *et al.*, 1993] goes further by allowing $\exists x\, \phi$ and $\exists x\, \psi$ to share one Skolem function provided the two formulæ 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 $\phi(\epsilon x\, \phi(x))$. [Baaz and Fermüller, 1995] have proved that these liberalizations can yield gigantic (non-elementary) improvements over conventional tableaux.

Limited experiments found that the first liberalization made little difference. One proof got shorter: problem 43 of [Pelletier, 1986]. However, the runtime actually increased due to the greater 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, 1989].

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. Perhaps another method of proof reconstruction could be found, but my experiments gave no guarantee that the effort would be sufficiently rewarded. Therefore, I decided to adopt standard δ-rules with no liberalizations.

Finally, there is the question of prenexing. [Baaz and Leitsch, 1994] 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. Users normally call the simplifier first, and it is equipped with rewrite rules such as $\forall x\, (\phi(x) \vee \psi) \leftrightarrow (\forall x\, \phi(x)) \vee \psi$ and even

$$(\bigcup_{i \in I} A_i \cap B) = (\bigcup_{i \in I} A_i) \cap B.$$

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 [Baaz and Leitsch, 1994].

6 Types and Overloading

Isabelle's framework for specifying formalisms is typed. Some of its logics, such as ZF set theory, 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 makes the tableau prover run faster than it could otherwise. Its data structures become simpler, and unification has less work to do. 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:

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

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

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

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. The prover can be directed 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.

Omitting types could allow non-normalizing terms such as $(\lambda x.\, xx)(\lambda x.\, xx)$ to form during a proof, resulting in non-termination. There is no evidence that this possibility occurs in practice, and it seems unlikely, but it cannot simply be dismissed.

7 Minor Points

Now, let us briefly consider some additional features of the prover. Most of them are required in order to handle generic rules.

Unification

Isabelle uses higher-order unification [Huet, 1975]. For efficiency, the tableau prover uses first-order unification. Bound variables are represented by de Bruijn indices: by their numeric depth in the formula. (My ML book explains de Bruijn indices in more detail [Paulson, 1996, page 376].) The two λ-abstractions $\lambda x.M$ and $\lambda x.N$ unify if M and N do. A variable $?x$ and term M are unifiable provided $?x$ does not occur in M and the instantiation captures no bound variables. This extension to the occurs check, easily performed with the de Bruijn representation, prevents errors such as unifying $\lambda x.x$ with $\lambda x.?y$. A bound variable unifies only with itself. Redundant λ-abstractions are erased using η-reduction, which takes $(\lambda x.Mx)$ to M provided x is not free in M. Also, β-reductions are performed, simplifying $(\lambda x.M)N$ to $M[N/x]$.

A possible improvement is to use pattern unification [Miller, 1992], allowing function variables to be instantiated in simple cases. However, pattern unification's main use is to support quantifier reasoning by a technique called *lifting* [Paulson, 1989]. (Miller calls it *raising*.) The tableau prover does not use this technique and thus can get by with first-order unification, which is much easier to implement.

Literals

If the current formula can neither close the branch nor be expanded, then it is moved to a list of literals. However, in generic tableau proof, the notion of literal depends upon the available rules. 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 $\mathbf{s} = \mathbf{t}$ 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 formulæ $\mathbf{s} = ?\mathbf{y}$ 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 $\mathbf{s} = ?\mathbf{y}$ is not suitable for substitution because $?y$ might later be instantiated with a term containing $\mathbf{s}$. If $\mathbf{s}$ is the Skolem term $\mathbf{f}(\mathbf{x_1}, \ldots, \mathbf{x_m})$, 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 $\mathbf{s} = \mathbf{A} \cup \mathbf{B}$, the literal $x \in \mathbf{s}$ becomes the compound formula $x \in A \cup B$. The ensuing rearrangement of the formulæ 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 might be necessary. (This concept was first used to allow backtracking over γ-rules in `Fast_tac`.) A rule is unsafe if its conclusion is weaker than its premises. 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 rule types. Equally important, it makes `Blast_tac` treat such rules similarly to `Fast_tac`.

Transitivity

Transitivity and similar 'recursive' 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. If so, then those premises are potentially recursive; the formulæ they generate will be put at the end of the branch to be expanded last. 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
- translation of tableau proofs to Isabelle proofs

The translation of Isabelle rules has to contend with different treatments of eigenvariables in quantifiers, since Isabelle does not employ Skolemization. Moreover, it discards virtually all type information. Both of these points 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 [Paulson, 1997a], and more details are available in [Paulson, 1994, Chap. 14].

Some further points can be seen in the treatment of a standard natural deduction rule, ∀ introduction:

$$\frac{\phi(x)}{\forall x\,\phi(x)}$$

This inference rule takes the premise ϕ to the conclusion $\forall x\,\phi$ and is subject to the usual proviso that x is not free in the assumptions. The rule is represented in Isabelle as an axiom:

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

Here, $\bigwedge$ is Isabelle's meta-level universal quantifier, Tr is a meta-level predicate to recognize true formulæ, and All is a constant used to represent the first-order universal quantifier. The Isabelle axiom states that if $\phi(x)$ is true for all x, then the formula $\forall x\,\phi(x)$ is also true. Like all Isabelle axioms, it can be seen as a generalized Horn clause.

The axiom must be translated to a tableau rule, which in this case introduces Skolemization. Isabelle formalizes quantifier reasoning using higher-order variables [Miller, 1992, Paulson, 1989]. During translation, the bound variable in the premise (namely x) is replaced by a Skolem term (call it $\mathbf{s}$) containing the free variables of the current branch. Thus, translating the axiom yields 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 formulæ as negative formulæ, but 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 formulæ, 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. Rarely, it fails, for example if the proof state contains function variables.

8.2 Giving Tableau Proofs to Isabelle

Translating a proof found by one tool for checking by another is an old idea. Proof planning [Bundy *et al.*, 1991] involves conducting searches in a specialized formalism and applying the resulting plans in a general, complex formalism. 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 formulæ using a stack, while Isabelle's normal mode would yield a queue; I had to add an Isabelle primitive for re-ordering a subgoal's assumptions.

Proof reconstruction is a simple idea, but the details can be complicated. 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, 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 done during the search. 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.

Rarely, proof reconstruction fails. After printing a warning, the tableau prover backtracks, but it seldom recovers. 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, the tableau prover has allowed a branch's formulæ to get out of order. If two similar formulæ are exchanged, then the wrong one might get expanded. Proof reconstruction occasionally fails because the tableau proof is unsound; almost certainly, because it has used a rule that involves overloading. Such a rule expects a constant to have a certain type. The techniques described in §6 cope even with overloading, except for uses of axiomatic type classes [Nipkow and Snelting, 1991].

[Stenz *et al.*, 1999] describe integrating the tableau prover $3T^AP$ with KIV (Karlsruhe Interactive Verifier). Their proof transformations can cope with *universal variables*: free variables that can be instantiated independently to close several branches. The great complexity of this transformation is justified if universal variables can speed up the search. LeanT^AP's statistics offer little support for universal variables [Beckert and Posegga, 1995, page 350], but Stenz says that they are sometimes useful.

9 Results

There are no recognized benchmarks for generic theorem provers: in particular, none involving higher-order syntax. The TPTP library is typical in consisting largely of first-order problems in clause form [Sutcliffe and Suttner, 1998]. For evaluation purposes, I have chosen some problems from [Pelletier, 1986]. They are relatively easy and are available in a non-clausal form. I have added the halting problem and four examples of my own.

Isabelle's existing classical reasoner provided `Fast_tac` (based on depth-first search) and `Best_tac` (based on best-first search). `Blast_tac` outperforms them in most cases: it is faster and proves many theorems that they cannot.

Table 1 compares the performance of `Blast_tac`, `Fast_tac` and leanT^AP on several examples. *Search* refers to the tableau prover, while *Verify* refers to the Isabelle reconstruction of the proof. The numbered problems are Pelletier's; problem 34 is also known as Andrews' Challenge. Halting II refers to the halting problem presented by [Dafa, 1997].[3] The benchmarks were run on a 300Mhz Pentium Pro running Linux. Isabelle was compiled using Standard ML of New Jersey version 110.0.3. LeanT^AP, with iterative deepening and universal variables, was compiled using SICStus Prolog version 3.

Statistics for the first-order problems are for Isabelle/FOL, though they also run in Isabelle/HOL. The last four problems are formulated in the set theory of Isabelle/HOL; `Blast_tac` can also prove their ZF equivalents. `Blast_tac` is indeed generic.

[3] This theorem is large rather than deep: even `Fast_tac` can prove it. It contains many large subformulæ that are repeated—presumably they are shared in Dafa's short proof.

$$\bigcup_{x\in C}\Big(f(x)\cup g(x)\Big)=\bigcup(f``C)\cup\bigcup(g``C) \qquad \text{Union-image}$$
$$\bigcap_{x\in C}\Big(f(x)\cap g(x)\Big)=\bigcap(f``C)\cap\bigcap(g``C) \qquad \text{Inter-image}$$
$$(\forall_{x\in S}\ \forall_{y\in S}\ x\subseteq y)\rightarrow\exists z\ S\subseteq\{z\} \qquad \text{Singleton I}$$
$$(\forall_{x\in S}\ \bigcup(S)\subseteq x)\rightarrow\exists z\ S\subseteq\{z\} \qquad \text{Singleton II}$$

Here " is the image operator, satisfying $y\in f``A \iff \exists_{x\in A}\ y=f(x)$.

	Search			*Verify*		`Fast_tac`	leanT^AP
Problem	depth	branches	time	tactics	time	time	time
24	4	16	40	52	30	210	540
26	3	17	30	43	40	430	0
28	3	7	20	29	30	140	0
34	7	100	200	431	2090	FAILED	24,170
38	4	30	50	100	130	840	70
43	5	24	50	48	60	FAILED	10
46	7	15	610	48	50	2,090	590
52	7	86	140	101	540	1,370	n/a
62	1	17	10	46	40	130	0
Halting II	7	2,015	10,990	1,086	8,310	220,000	∞
Union-image	3	12	90	40	50	560	n/a
Inter-image	3	12	90	36	50	$1430^\dagger$	n/a
Singleton I	4	117	370	19	20	∞	n/a
Singleton II	4	115	370	19	10	∞	n/a

runtimes given in milliseconds
∞ = still running after 5 minutes
$\dagger$ = using `Best_tac`; would be ∞ for `Fast_tac`
n/a = not applicable

Table 1: `Blast_tac` Compared With `Fast_tac`

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, $\neg(?x\in\mathsf{UNIV})$. Adding this rule greatly increases the search space, probably by closing too many branches.

Proof reconstruction time often exceeds search time, especially when the proof is long and is found with little search. Examples include problems 34 and 38, and there are instances throughout the Isabelle proof scripts. Coding the tableau prover directly in ML makes it decisively faster than Isabelle's proof engine.

Benchmarks can give a distorted picture. For most first-order problems, `Blast_tac` outperforms `Fast_tac`, but the latter is occasionally much faster because of its incomplete search strategy. First-order problems allow comparison with other systems, but are of little relevance to Isabelle. `Blast_tac` is most important in application domains such as security protocols [Paulson, 1997b], whose use of inductive definitions cannot be reduced to first-order logic. A major advantage of `Blast_tac` is its ability to cope with transitivity and similar rules.

`Blast_tac` compares favourably with lean$T^{A}P$, especially if we ignore the time taken for proof reconstruction. `Blast_tac`'s greater generality slows it down, but its additional heuristics compensate for that loss.

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. Lean$T^{A}P$ consists of five Prolog clauses; `Blast_tac` is around 1,300 lines of ML (or 46K 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, 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 [Paulson, 1997b] 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 involves supplying most lemmas by default so that users do not have to think about them. `Blast_tac` can often cope with the resulting search space.

Perhaps the work makes a third contribution: that the experience reported here may benefit the implementors of other proof tools.

Acknowledgement

The research was funded by the EPSRC, grants GR/K77051 'Authentication Logics' and GR/K57381 'Mechanizing Temporal Reasoning.' Nikolaj Bjørner, Fabio Massacci and various referees commented on previous versions.

References

[Baaz and Fermüller, 1995] Baaz, M. and Fermüller, C. G. (1995). Non-elementary speedups between different versions of tableaux. In P. Baumgartner, R. Hähnle, and J. Posegga, editors, *Theorem Proving with Analytic Tableaux and Related Methods: 4th International Workshop, TABLEAUX '95*, LNAI 918, pages 217–230. Springer.

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

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

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

[Bundy *et al.*, 1991] Bundy, A., van Harmelen, F., Hesketh, J., and Smaill, A. (1991). Experiments with proof plans for induction. *Journal of Automated Reasoning*, **7**(3), 303–324.

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

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

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

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

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

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

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

[Nipkow and Snelting, 1991] Nipkow, T. and Snelting, G. (1991). Type classes and overloading resolution via order-sorted unification. In *5th ACM Conf. Functional Programming Languages and Computer Architecture*, pages 1–14. Springer. LNCS 523.

[Paulson, 1989] Paulson, L. C. (1989). The foundation of a generic theorem prover. *Journal of Automated Reasoning*, **5**(3), 363–397.

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

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

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

[Paulson, 1996] Paulson, L. C. (1996). *ML for the Working Programmer*. Cambridge University Press, 2nd edition.

[Paulson, 1997a] Paulson, L. C. (1997a). Generic automatic proof tools. In R. Veroff, editor, *Automated Reasoning and its Applications: Essays in Honor of Larry Wos*, chapter 3. MIT Press.

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

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

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

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

[Stenz *et al.*, 1999] Stenz, G., Ahrendt, W., and Beckert, B. (1999). Proof transformations from search-oriented into interaction-oriented tableau calculi. *Journal of Universal Computer Science*, **5**(this number).

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

[Sutcliffe and Suttner, 1998] Sutcliffe, G. and Suttner, C. (1998). The TPTP problem library: CNF Release v1.2.1. *Journal of Automated Reasoning*, **21**(2), 177–203.