Theory and Practice of a Generic Tableau Engine: The Tableau WorkBench

Pietro Abate[1] and Rajeev Goré[2]*

The Australian National University and NICTA
Canberra ACT 0200, Australia
[Pietro.Abate|Rajeev.Gore]@anu.edu.au

Abstract. The tableau method is widely used by logicians to give decision procedure for classical and non classical logics. But the move from proof-theoretic tool to an efficient automated theorem prover is often difficult for researchers with no technical background. The main "problem" is the non-determinism inherent in the tableau method. In this paper we give a functional algorithm to mechanize the tableau method which is concise and easy to re-use. We also briefly describe an implementation of this algorithm called the Tableau Workbench.

1 Introduction and Motivation

Tableau calculi and all other variants like labelled tableaux, sequent calculi, hypersequents, etc. can be characterised by their common underlying theme: proof search consists of building a *proof tree* using a finite set of *rules*. Modularity and simplicity allow tableau methods to cleanly specify logical systems from a proof theoretical perspective. For example, tableaux for basic modal logics simply add rules for modal connectives to tableaux for classical propositional logic. Tableau calculi can also be viewed as non-deterministic algorithms that search for models of a given formula.

From an automated reasoning prospective, the main obstacle to adopt the tableau method as a practical algorithm is the inherent non-determinism which makes a naive implementation unsuited to solve large problems efficiently. The three main forms of non-determinism that can be identified are associated with three aspects of the proof search, namely, with which leaf node to continue first, which rule to apply and to which principal formula. From a programming perspective, each of these three aspects corresponds to a specific part of the proof search algorithm, namely, a function that explores the search space, a function that selects which rule to apply, and an heuristic which selects which formula to consider first. Once these choices have been made, there still remains the question of backtracking over some of these choices.

Although the literature abounds with many implementations of the tableau method for specific logics, to the best of our knowledge, none reveal details that can be re-used in a generic manner. Very efficient domain specific theorem provers such as Fact++

* National ICT Australia is funded by the Australian Government's Dept of Communications, Information Technology and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Centre of Excellence program.

or MSPASS [TH06,HS00] includes many technicalities like optimizations and translation methods in their algorithms, making them extremely focused to solve a specific problem but difficult to extend. Generic theorem provers such as Lotrec and the LWB [GHLS05,Heu96] target a wide range of logics, but are implemented in programming languages like Java and C++, leading to large programs which are hard read, to generalize and to extend. Extensions of LeanTap, despite their simplicity, require some in-depth knowledge of Prolog programming and there is no generic implementation that can be extended without a technical background. For example, it is not trivial to add histories (loop checking) to implementations based upon LeanTap.

These efforts to merge efficiency with generality highlight the mismatch between the high level concepts of the tableau method and the low level details of a generic implementation of it. If we compare the solutions in generic interactive provers we find that almost all are implemented in high level functional languages. This is mainly because functional programming is particularly well suited to the nature of the problem, leading to small programs which are easier to understand and maintain.

Here we present an algorithm for a generic and deterministic procedure based on the tableau method. Although it may seem like a simple recursive formulation of the tableau method, we believe the tableau community will benefit from it. We also present the Tableau Workbench (TWB), a generic tableau engine that implements this algorithm and that can be used to define traditional tableau calculi with little programming knowledge. In Section 2 we discuss different aspects of mechanizing the tableau method. In Section 3 we present our generic tableau algorithm using a high-level functional programming language. In Section 4 we describe the TWB, present a prover for modal logic **K** and compare its performance with the Logics Work Bench on standard benchmarks.

1.1 Functional Programming

Hughes, the father of the Haskell programming language, argues that since modularity is the key to successful programming, the functional programming style is vitally important to solve real world problems [Woo01]. Functional programs are easier to understand and to maintain, are often shorter than programs written in imperative style [HJ94]. The functional programming style achieves modularity via the ability to compose functions in an abstract way, and to have control of the data flow that is made explicit (sometimes, in pure functional languages, painfully explicit).

Functional programming languages are divided in two categories: *pure* languages, such as Haskell and *impure* languages such as OCaml. In impure languages, programmers have access to a number of constructs with *side effects* such as exception or assignments. Pure languages on the other hand, are easier to reason about because of the absence of computation with side effects. An important programming technique, developed to make pure functional languages more accessible without introducing unwanted features, is the monadic programming style. The concept of monads arises from category theory and was first applied by Moggi to structure the denotational semantics of programming languages. It was brought closer to the programming world by the seminal work of Wadler [Wad92]. In general, a monad is a construct to abstract a computation and to regain some flexibility peculiar to impure functional languages.

$$(\wedge)\ \frac{A \wedge B; Z}{A; B; Z} \qquad (\wedge_{\text{inv}})\ \frac{A \wedge B; X}{A; B; X} \qquad \textbf{(K)}\ \frac{\Diamond P; \Box X; Z}{P; X}$$

Fig. 1. Definition of the $(\wedge)$ and $(\wedge_{\text{inv}})$ and (K) rule.

Monads can be used to implement non-deterministic computations, where the evaluation of an expression returns a list of possible answers. The monad that encapsulates non-deterministic computation is the *list monad* that we use in Section 3.2.

Another interesting functional programming technique that we use in Section 3.2 is the continuation-passing style technique. Continuations represent the future of a computation as a function from an intermediate result to the final result. In a nutshell, if we consider the evaluation of an expression $f(e)$, a continuation k of $f(e)$ represents the part of the computation of $f(e)$ that occurs after e has been evaluated.

2 Mechanising the Tableau Method

We now discuss several aspects of tableau calculi which are important when trying to mechanize the tableau method. We assume the reader is familiar with tableau methods.

Basic Notions Tableau rules are of the form $\dfrac{n}{d_1 \ldots d_m}$ where n is the numerator, while $d_1 \ldots d_m$ is a list of denominators. The numerator contains one or more distinguished formula schemas called *principal formulae* and one or more formulae schema called *side formulae*. We use meta-variables $A, B, P, Q, \cdots$ to represent principal formulae and X, Y, Z for, possibly empty, "containers" of side formulae: traditionally sets, multisets or sequences, but possibly more complex data structures for generality.

Operationally, tableau rules can be seen as functions to transform a (container) node into a list of (container) nodes during the construction of a graph. We say that a rule is *applicable* if we can *partition* the contents of the current node to instantiate the meta-variables in the numerator . The *denominators* of a rule represent actions to expand the graph by creating new nodes according to these instantiations.

The repeated application of rules gives rise to a tree of nodes, with children obtained from their parent by instantiating a rule. To capture sequent calculi as well as tableau calculi, we view this process as a search for a proof where all leaves are "closed". But in general, for tableau calculi with cyclic proofs, there are really three notions: *success*, *failure* and *undetermined*: we stay with the dichotomy "closed" as success (of proof search) and "open" as failure (of proof search) for now to simplify the exposition.

A numerator pattern like Z is *unrestricted* since its shape does not restrict its "contents" in any way. Conversely, the pattern $\Box X$ is restricted to contain only $\Box$-formulae, $X \wedge Y$ is restricted to contain only $\wedge$-formulae while $A \wedge B$ is restricted to contain exactly one single $\wedge$-formula. Patterns like $X \wedge Y$ are allowed to be empty, but $A \wedge B$ is not. Thus the $(\wedge)$-rule from Figure 1 is seen as instructions to partition the contents of a node N to instantiate $A \wedge B$ to a single $\wedge$-formula from N and to instantiate Z to all the *other*

formulae so that $Z = N - (A \wedge B)$ where "$-$" is the "subtraction" operator appropriate to our containers. In particular, if N is a set then there are no "hidden contractions".

Similarly, the (K)-rule from Figure 1 instructs us to partition N into three disjoint parts: a single $\Diamond$-formula $\Diamond P$; a container $\Box X$ of $\Box$-formulae; and a container Z of all *other* formulae. The intended interpretation of the (K)-rule is invariably that Z should not contain any $\Box$-formulae since this ensures completeness of the rule. We therefore further assume that the partitions specified by the numerator are *maximal*.

Currently, we forbid "purely structural" numerators like $X;Y$ which non-deterministically partition the node N into two arbitrary disjoint containers X and Y (even though such numerators may be useful for linear logic), and numerators like $\Box X;\Box Y;Z$ since both can lead to an exponential number of possible different partitions.

Consequently, the *only* form of non-determinism in rules is the *formula choice* non-determinism in choosing the principal formula. More generally called *instance choice*, this non-determinism leads to *implicit branching* since the choice-point is hidden in the pattern-matching (rule instance). We return to this issue later after identifying the various forms of non-determinism inherent in tableau proof search.

Removing Non-Determinism A *don't care* non-deterministic choice is an arbitrary choice of one among multiple possible continuations for a computation, all of which return the same result. A *don't know* non-deterministic choice is one choice among multiple possible continuations, which do not necessarily return the same result. For completeness in a tableau calculus, it is sufficient to assume that all choices are *don't know* non-deterministic. However, a deterministic algorithm based on this assumption will be extremely inefficient since it has to consider all possible continuations for all choices. Thus it is imperative to remove as many forms of non-determinism as possible.

Traditionally, logicians focus on the existence of a derivation tree rather than on how it is found. Conversely, automated reasoners focus on designing algorithms to search for a derivation tree as efficiently as possible. From a theoretical perspective, it is therefore not important to specify the order in which rules are applied or how the proof tree is explored. Practically, however, specifying a particular rule order can dramatically speed up the search and construction of a proof tree. Moreover, since different rule orders can produce different derivation trees, an efficient decision procedure should always try to build the smallest derivation tree. In general, we can identify three forms of non-determinism associated with three fundamental aspects of a generic decision procedure:

Node-choice: the algorithm determines which leaf becomes the current node;
Rule-choice: the algorithm determines which rule to apply to the current node;
Instance-choice: a heuristic procedure determines the order in which to explore the different instantiations (partitions) of the current node created by the chosen rule.

Node-choice. The first source of non-determinism is in which (leaf) node to select to explore the search space and is usually handled by a visit algorithm that selects this node out of possibly many in the current tree. For exposition, we use a depth first visit function that deterministically selects the left most node as candidate but our algorithm allows any visit strategy to be used to explore the search space (ie. breadth-first).

$$(\vee)\ \frac{A \vee B; Z}{A; Z | B; Z} \qquad \text{(det-K)}\frac{\Diamond P_1; \cdots; \Diamond P_n; \Box X; W}{P_1; X || \ldots || P_n; X} \qquad \text{(rec-K)}\frac{\Diamond P; \Box X; \Diamond Y; W}{P; X \,||\, \Box X; \Diamond Y}$$

Fig. 2. Universally and Existentially Branching rules.

Rule-choice. The second source of non-determinism in tableau methods arises from the lack of guidance when applying logical rules. For example, in classical propositional logic, where all rules are invertible, the order in which rules are applied is not important. However specifying a strategy that applies all linear rules and the axiom first, and then all branching rules, can potentially shorten the proof tree. The problem is however different in basic modal logic, where not all sequences of rule applications ensure a solution since the (K)-rule is not invertible.

The problem is even more complicated in tableau-sequent calculi with more than one non-invertible rule: for example intuitionistic logic as in [Dyc92]. In this situation, at any given choice point, after all invertible rules are applied, we are forced to guess which non-invertible rule to apply, and eventually to undo this decision if it does not lead to the construction of an acceptable proof tree. Consequently, if the proof tree obtained from the application of the first rule of a sequence of non-invertible rules does not respect a logic-specific condition, the entire proof tree generated from that rule application must be discarded. To recover from this wrong choice, the proof must be re-started using the next non-invertible rule available in the sequence.

Thus, for efficiency, tableau rules must be coupled with a *strategy* to control the order in which rules are applied. Such a tactic language is described in Section 3.1.

Instance-choice. The third form of non-determinism is the aforementioned choice of one rule instance (partition) of the current node from potentially many rule instances (partitions) of the current node. As we saw, the different instances are obtained by choosing different formulae as the principal formulae. Then, this nondeterminism can be resolved by using optimisation techniques based on logic-specific considerations used to reduce the size of the search space. For example, if the chosen rule is an $(\vee)$-rule, heuristics such as MOMS [Fre95] (Maximum number of Occurrences in disjunctions of Minimum Size) or iMOMS [Fre95] (inverted MOMS) order the disjunctions (formulae) in the node to always choose the less/more constrained disjunct, which, in principle, should lead to an earlier clash.

Controlling Backtracking. The search procedure is an attempt to find a "closed" tableau, so it must recover from unsuccessful branches that do not close. For rules with "implicit branching", the best recovery action depends upon the type of non-determinism embodied in the different rule instances.

For example, the traditional $(\wedge)$-rule from Figure 1 is invertible in many tableau calculi: in that case, it embodies a don't care non-determinism for formula choice since we are free to choose *any* conjunction from the current node as the principal formula of this rule instance. If the chosen instance does not lead to a closed denominator, then there is no need to backtrack over the other different conjunctions in the current node since invertibility guarantees that none of them will lead to a closed denominator.

Operationally, it is currently not feasible to automatically detect whether a rule is invertible in a given tableau calculus, so one way to declare a rule as invertible is by writing it as $(\wedge_{\text{inv}})$ using a double-line separator as shown in Figure 1.

On the other hand, in the traditional (K)-rule, the implicit branching is essential since different choices may give different results, thus embodying a form of don't know non-determinism. For example, consider the different choices of principal $\Diamond$-formula for a node containing both $\Diamond p$ and $\Diamond\bot$: the $\Diamond p$-choice may or may not close, but the $\Diamond\bot$-choice is guaranteed to close. That is, if one rule instance does not close then we must backtrack over the other possible rule instances *until we find an instance that gives a closed denominator* or all instances are found to be open, before backtracking further.

Traditional tableau calculi do not envisage any form of communication between the branches of a tableau, but operationally we wish to allow such communication. For traditional explicitly branching rules like $(\vee)$, it is possible to add decorations that allow inter-branch communication. But this is not possible with the branches implicit in the choice of principal formula of the (K) rule. We therefore allow the denominators of a rule to be separated by a new type of separator $||$ which captures "existentially branching": the numerator is closed if **some** denominator is closed. This is dual to the "universally branching" separator $|$ used for explicitly branching rules like $(\vee)$: the numerator is closed if **all** denominators are closed.

Figure 2 shows two ways of "determinising" the implicit backtracking in the (K)-rule using existential branching. By our maximality constraint on numerator patterns, the rule (det-K) instructs us to partition the numerator into n principal $\Diamond$-formulae, a container $\Box X$ for all the $\Box$-formulae, and a container W for all the other formulae. The double-lines tells us to commit to this partition and the $||$ separator tells us that the numerator is closed if some denominator is closed. This rule is actually difficult to implement since the parameter n is effectively a free variable which must be instantiated. But we can determinise even this aspect by using a recursive version called the (rec-K)-rule from Figure 2 instead which can be read as: choose a principal formula $\Diamond P$, put all other $\Diamond$-formulae in $\Diamond Y$, put all $\Box$-formulae in $\Box X$, put all non-boxed and non-diamond formulae in W, commit to this partition, and close the numerator if some denominator closes. By repeatedly applying (rec-K) to the right denominator, we can step through the same partitions as the (det-K) rule, without having to worry about the value of n since the (rec-K) rule will fail to apply when the right denominator contains no $\Diamond$-formulae because $\Diamond P$ cannot be empty. We need a strategy to ensure that no other rules are applied between these consecutive applications of (rec-K) but it is easy to capture such a strategy using the tactic language defined in Section 3.1.

The traditional $(\vee)$-rule also contains a form of implicit branching since different choices of the principal formulae may be possible if the current node contains more than one disjunction. So we also need to explicitly declare whether such rules are to be treated as invertible by using a double-line separator as for the $(\wedge_{\text{inv}})$-rule.

From a general perspective, the two types of branching can be characterised as conditionally branching: "explore different denominators until some user-defined condition becomes true", with both universal and existential branching as instances.

Our resulting rule classification is shown in Table 1. Linear rules have one denominator, while branching rules have two or more which are universal or existential related.

Instance choice	Branching			Linear
	Universal	Existential	Conditional	
Backtrack	$(\vee)$	-	-	(K) $(\wedge)$
Commit	$(\vee_{\text{inv}})$	(rec-K)	-	$(\wedge_{\text{inv}})$

Table 1. Rule Classification

Each rule can either commit or not commit to the partition (instantiation) selected first by the formula-choice heuristic (by being invertible or non-invertible). If the rule commits then only this first partition is considered while expanding the denominator(s). If the rule does not commit then the denominator(s) will be re-instantiated for each partition generated by the application of the rule numerator to the current node until an instantiation is found that closes the numerator. With this rule classification it is easier to mechanize tableau rules as specified in the literature whilst removing non-determinism. Rules for classical modal logic can be characterized as shown.

3 A Generic Algorithm for the Tableau Method

The algorithm that we present is a procedure that visits a tree that is generated by the repeated application of a finite set of rules to an initial node containing a finite number of formulae. This visit procedure is composed of two functions. The first function (`strategy`) selects a new rule to be applied to the current node while the second function (`visit`) traverses the tree generated by the application of the rule, by recursively calling the visit procedure. Given a node and the state of the strategy, the procedure follows this algorithm:

1. Select a new rule using the `strategy` applied to the current node;
2. Build the new leaves of the proof tree according to the selected rule;
3. Call `visit` on all new branches (leaves) of the tree;
4. If there are no more rules, then return a leaf and backtrack;

In Figure 3, we show a snapshot of the visit procedure by focusing on a node in the proof tree (the *current node*). We can isolate four basic components of the visit procedure related to the rule that was chosen to be applied to the current node:

Rule condition: this function is used by the strategy procedure. It checks if a rule is applicable to the current node and if all side conditions are satisfied. The result is a partition of the current node according to the patterns specified in the numerator of the selected rule.

Rule action: this function accepts a rule that has been selected and applies it to the current node in order to expand the proof tree. The result of this function is a list of nodes, each identifying a new branch in the proof tree.

Branch condition: this function is executed during the visit of the sub-tree rooted at the current node. Assume we have a branch condition for each branch generated by the rule application, and assume further we have *n* branches. Then, after visiting

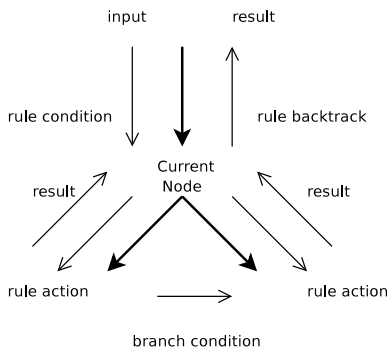

Fig. 3. Algorithm Overview

branch *i*, the branch condition related to branch *i* is executed to determine if the sub-tree created so far (including all the already visited branches) satisfies a logic-specific condition. If this is the case, the visit explores the next branch (if any), otherwise, the visit is interrupted and the procedure backtracks.

Rule backtrack: this function is executed after all branches (or a number of them according to the branch condition) are visited, and synthesises the result of the visit of the sub-tree rooted at the current node and passes it up to the parent.

3.1 The Tactic Language

Naive tableau methods do not specify the order in which rules are applied during proof-search. A tactic language is important to complement and extend a tableau calculus specification and therefore to design efficient decision procedures. By removing the inherent non-determinism of tableau methods, the search space can be greatly reduced, making problems with high complexity tractable on the average case.

Tactical languages originated in the work of the Edinburgh LFC and then were refined in `Isabelle`. A tactic specifies the order in which rules are applied during the proof search. The following tactic language which can be used to specify decision procedures for classical and non-classical logics is inspired by the work in [MGW95]:

$Skip$ is the tactic that always succeeds
$Fail$ is the tactic that always fails
$Rule(r)$ is the tactic that succeeds when rule r is applicable.
$t = t_1; t_2$ is the tactic that fails if either t_1 or t_2 fail, succeeds otherwise.
$t = t_1 | t_2$ is the tactic that fails if both t_1 and t_2 fail, succeeds otherwise.
$Repeat(t) = mu(X).((t; Var(X))|Skip)$ using a fix-point operator $mu(X)$. If tactic t succeeds, the tactic $Repeat(t)$ behaves like $t; Repeat(t)$. If t fails, then the tactic $Repeat(t)$ behaves like $Skip$. The strategy Repeat always succeeds. We write this tactic as $(t)*$.

For example, assuming the traditional (id) rule $p; \neg p; X$ for closing branches, the tactic `((id | ∧ | ∨ )* ; K )*` specifies the standard search strategy for modal

tableau where we first saturate a node with the classical propositional logic rules to obtain a frontier of leaves, and then apply the (K) rule to obtain the successor "world", and repeat the tactic until no rules are applicable (when repeat succeeds).

3.2 The Core Algorithm

We now give a high-level description of the algorithm outlined in the previous section, focussing on aspects related to node-choice and rule-choice types of non-determinism. Formula-choice is easily solved, for example, by an ordering function so we do not consider it in this paper. Implementation details about backtracking are also omitted.

To make the code more accessible to a wider audience, we give the two main functions in a pseudo-code evocative of the Haskell programming language. The given algorithm is functionally pure: that is, if the rule object does not have computational side effects, the recursive functions do not use heap space. Moreover, we assume that the language is lazy, so the execution of a function is delayed until the results are needed.

3.2.1 Preliminaries We use the following standard list functions with indicated types:

`lookup a s` : $k \to (k,e)$ `list` $\to t$ given a key of type k and a list containing pairs of the form $(key, entry)$, returns the entry of type e associated with the key if any, or it fails otherwise.

`map f l` : $(e \to e')$ $\to e$ `list` $\to e'$ `list` applies the function f to each element of the list l and returns the list of results.

`flatten l` : e `list list` $\to e$ `list` flattens a list of lists into a list.

@ : e `list` $\to e$ `list` $\to e$ `list` concatenates two lists into one list.

:: : $e \to e$ `list` $\to e$ `list` returns a new list consisting of its first argument appended to its second argument. We write $[a]$ for a :: [].

We write *lambda x . f* as a short-hand to declare an anonymous function with body f and an argument x. For example, *lambda* (x,y) *.* $x+y$ expects an argument that is a pair to be applied as an argument to the anonymous function with body $x+y$. We also write *lambda _ . f* for the function that discards its argument and returns the function body f. In the following we also use curried functions, a technique which is used to partially evaluate functions. A curried function is a function with n arguments that is considered as a function of one argument which returns another function of $n-1$ arguments. For instance, if $\lambda x.\lambda y.x+y$ is a function of two arguments, and is given an argument 1, the result is a partial evaluation $\lambda y.1+y$ of the original function that now has only one argument. We use curried functions to encode continuations.

A rule is an object, in terms of object oriented programming, with three methods, *check*, *down* and *up*, associated with rule conditions, rule application and rule backtrack from Section 3, respectively. We use the symbol ♯ to invoke a method so `skip♯check` means that we invoke the method `check` of the object `skip`. The three methods are:

check : given a node returns a boolean value;

down : given a node, returns a recursive type *tree* of the form $Leaf(n)|Tree(l)$ where l is a list of subtrees and n is a tableau node;

Listing 1.1. Strategy function

```
function strategy env tactic node =
  case tactic of
  'Fail'       : []
  'Skip'       : [(skip, [])]
  'Rule(rule)': guard (rule#check node) >> lambda _ . [(rule, [])]
  'Alt(t1,t2)': (strategy env t1 node) ++ (strategy env t2 node)
  'Seq(t1,t2)': (strategy env t1 node) >>
     lambda (rule,stack) . [(rule, stack @ [strategy env t2])]
  'Mu(x,t)'    : strategy ((x,t)::env) t node
  'Var(x)'     : if x undefined then "Variable not defined"
                 else strategy env (lookup x env) node
```

up : given a list of trees returns a tree.

We now define some monadic operators to characterize a *list monad* [Wad92]:

$[]$: $\_\,$`mlist` This is an empty monad list usually called the "zero" of the monad.

$>>$: e `mlist` $\rightarrow (e \rightarrow e$ `mlist`$) \rightarrow e$ `mlist` This operator, given an element of type e `mlist` and a function of type $e \rightarrow e$ `mlist` returns a list monad of type e `mlist`. This function is commonly called "bind" and its result is the application of the function (second argument) to all results of the computation of the first argument. We use an infix notation for this operator, thus $a >> f$ is equivalent to $>>\ a\ f$. This function can also be expressed in terms of list operators `flatten` and `map` where $a >> f \equiv$ `flatten` (`map` $f\ a$). For example:

$$
\begin{array}{ll}
[1;2;3] >> \lambda x.[x+1] & \equiv \\
\texttt{flatten}\,(\texttt{map}\,(\lambda x.[x+1])\,[1;2;3]) & \equiv \\
\texttt{flatten}\,([[2];[3];[4]]) & \equiv \\
[2;3;4] &
\end{array}
$$

$++$: e `mlist` $\rightarrow e$ `mlist` $\rightarrow e$ `mlist` This operation is the concatenation of two monadic computations. In this particular case of a list monad it is the same as a list concatenation. It is commonly called "plus".

`guard` : *bool* $\rightarrow$ *bool* `mlist` If boolean argument b evaluates to true then returns a non zero monad of type *bool* `mlist`, else returns an empty monad list.

3.2.2 The Strategy Function In Listing 1.1 we give the implementation of an interpreter for the tactic language described in Section 3.1. The function `strategy` takes the following arguments:

`env` a list composed of pairs of the form $(variable, tactic)$;
`tactic` a term of the language described in Section 3.1;
`node` the current tableau node.

The function `strategy` returns a list of pairs of the form $(rule, stack)$ containing a rule object and a stack of future computations, and is evaluated as follows:

`Fail` simply returns an empty list;
`Skip` returns a list composed of one pair where the first component is the rule `skip` and the second is an empty list representing the continuations;
`Rule(rule)` applies the function `guard` with argument `rule♯check node` to check if the rule is applicable to `node`. If so, it passes the boolean `true` to the anonymous function `lambda _ . [(rule, [])]` which discards this argument and returns a list composed of one pair where the first component is the rule `rule` and the second is an empty list representing its continuation. Otherwise if `rule♯check node` evaluates to false, the result of the function `guard` is an empty list so `>>` returns an empty list representing a failed rule application;
`Alt(t1 ; t2)` returns the monadic concatenation of the result of the computation of the function `strategy` to both arguments of the alternation;
`Seq(t1 ; t2)` applies the tactic $t1$ to the node and obtains a list of pairs of the form $(rule, stack)$ where *rule* is the first applicable rule in tactic $t1$ and *stack* is the continuation of the function `strategy env t1 node`. It then binds this list to an anonymous function that returns a list of pairs where each pair contains *rule* and a continuation that extends the *stack* from the first computation by attaching the curried function `strategy env t2` which is a continuation function waiting for its node argument;
`Mu(t)` modifies the environment with a new entry (x,t) and calls the strategy function;
`Var(x)` invokes the strategy function with the tactic associated to the defined variable x in the current environment or reports that x is undefined.

3.2.3 The Visit Function The visit function in Listing 1.2 is implemented as two mutually recursive functions, `visit` and `dfs`. The `visit` function has arguments:

`traversal` a function that specifies a traversal of an n-ary tree;
`str` a strategy function that given a node returns the next rule to be applied;
`stack` the control-stack containing the continuation of the strategy;
`node` the current tableau node.

The proof tree is explored by calling the visit function as follows:

```
visit dfs (strategy [] tactic) [] node
```

where:

`dfs` implements a depth first traversal of an n-ary tree (the proof tree);
`strategy [] tactic` calls the function strategy with an empty environment and the initial tactic for applying rules as specified by the user;
`[]` is the initial control-stack that is empty;
`node` is the initial node as given by the user.

Listing 1.2. Visit function

```
function visit traversal str stack node =
 (str node) >>
  lambda (rule,newstate) .
   case (newstate,stack) of
    '([],[])':
     let tree = rule#down node in
     let result =
      case tree of
       'Leaf(n)'       : [Leaf(n)]
       'Tree(l)': map (lambda n . Leaf(n)) l
     in [rule#up (result)]
    '([],t::tl)'    : traversal t rule tl (rule#down node)
    '(t::tl,stack)' : traversal t rule (tl@stack) (rule#down node)

function dfs str rule stack tree =
 case tree of
  'Leaf(n)' : [rule#up [Leaf(n)]]
  'Tree(l)' : [rule#up (l >> visit dfs str stack)]
```

In particular the `dfs` function is implemented as follows: If the argument `tree` is a $Leaf(n)$ containing a node n then we apply the method up of the object `rule` to the list composed of only one element $[Leaf(n)]$. The result of this operation is a list that contains the outcome of the exploration of the subtree with root the current node n. Otherwise, if the argument `tree` is of type $Tree(l)$ then we apply the method up of the object rule to the list returned by the computation of the function (`l >> visit dfs str stack`). In details, we bind the list l to the curried function `visit dfs str stack` which is waiting for its node argument and therefore we execute the function on each element of the list l preserving the order. The result of this operation is a list.

The `visit` function implements the control mechanism to select a rule and backtrack if no more rules are available. If there exists a successful tactic, the `visit` function returns the result of the visit of the proof tree generated by using that tactic from the initial node. In Listing 1.2 first we apply the function `str` to the current node. Note that the function `str` is a curried function that represents the continuation of the `strategy` function with the environment and a tactic saved in its closure as created by the function strategy from Listing 1.1. If `str node` succeeds, and therefore there exists a rule that is applicable to the current node, the value of *newstate* and the control-stack determines the behaviour of the visit function as follows:

$([],[])$ The newstate and the control-stack are both empty. So we have to apply the rule and backtrack by returning a tree to the parent. We apply the rule to the node via *rule♯downnode* to obtain the tree. If the rule was linear and therefore its resulting tree is immediately of type $Leaf$, then we return it via *rule♯up result*. Otherwise if the rule was a branching rule, we first need to transform each child n in the list l into a node of type $Leaf(n)$ and then return the resulting list via *rule♯up result*;

$([], t :: tl)$ The newstate is empty, but the control-stack has at least one element. Therefore the traversal proceeds by selecting the first tactic continuation t and continuing with the traversal function using the remainder of the list tl to traverse the result of applying the $rule\sharp down$ function to the current node;

$(t :: tl, \_)$ The newstate is not empty. Then we continue the traversal with the first component t in newstate and we prepend tl to the control stack, again to traverse the result of applying the $rule\sharp down$ function to the current node.

4 The Tableau WorkBench

The Tableau Workbench (TWB) is a generic framework for building automated theorem provers for arbitrary propositional logics without learning complex programming languages. The TWB has a small core that defines its general architecture, some extra machinery to specify tableau-based provers and an abstraction language for expressing tableau rules. A new logic module defined by a user is translated and compiled with the proof engine to produce a specialized theorem prover for that logic [Aba07].

The TWB allows users to specify tableau rules using a high level language designed to allow users to "cut and paste" tableau rules from textbooks and to generate a specialized theorem prover for the chosen logic. We put great effort to make the input language accessible to logicians with little programming background.

For example, in Figure 4 we give as example the tableau calculus for modal logic K (with no optimisations). The connectives are defined and given a binding strength of `Zero` (highest) to `Two` (lowest). The principal formulae of each rule are enclosed in braces (cannot be empty) or in parentheses (can be empty). The horizontal line in the rules either commits to the choice of the principal formula via a sequence of at least two "$==$" signs or allows backtracking over this choice via a sequence of at least two "$--$" signs. This syntax actually permits rules to be written much more concisely than shown: for example, the $(\wedge)$ can be written in one line as:

```
RULE And { A & B } ; Z == A ; B ; Z END
```

Even using the long-hand, theorem provers for many classic modal logics can be implemented in the TWB in less than 50 lines of code, plus support functions to convert the input formula to negated normal form. See `http://twb.rsise.anu.edu.au/demo`

The TWB, including the parser and interpreter for the end use language, is under 4000 lines of code. This is the result of two design choices. The first to use a functional programming language like OCaml, and the second to minimize the use of imperative constructs. The TWB implements the algorithm presented in this paper and also provides two well known extensions to the tableau method, histories and variables. Histories [HSZ96] store the results of previous applications of certain rules on the current branch and are passed from numerator to demoninators to block us from re-creating these results again. Variables are used to pass information regarding already explored branches from denominators to numerators [Sch98]. In conventional tableau calculi, we can imagine that the information regarding the "status" of a branch (being open or closed) is stored in a variable passed from the leaves to the root Implementation details can be found in [Aba07]. The TWB is available at `http://twb.rsise.anu.edu.au`.

```
CONNECTIVES
  And, "_&_",  Two;
  Or,  "_v_",  Two;
  Dia, "Dia_", One;
  Box, "Box_", One;
  Not, "~_", Zero;
  Falsum, Const;
END
TABLEAU
  RULE Id
       { A } ; { ~ A } ; Z
      ======================
          Close
  END
  RULE False
       Falsum ; Z
    =================
       Close
  END
  RULE And
       { A & B } ; Z
      ===============
        A ; B ; Z
  END
  RULE Or
        { A v B } ; Z
      ===============
       A ; X | B ; Z
  END
  RULE K
      { Dia A } ; Box X ; Z
      ----------------------
            A ; X
  END
END
PP := Pclib.nnf  (* a preprocessor for negated normal form *)
NEG := Pclib.neg (* a function to negate the input formula *)
STRATEGY ( (Id|False|And|Or)* ; K)*
```

Fig. 4. Simple and Compact Tableau Rules for Modal Logic K

class	twb	lwb	class	twb	lwb
k_branch_p	6	15	k_branch_n	4	11
k_d4_p	**11**	10	k_d4_n	**17**	6
k_dum_p	**18**	16	k_dum_n	**19**	21
k_grz_p	**21**	17	k_grz_n	**21**	21
k_lin_p	**21**	14	k_lin_n	**21**	5
k_path_p	**16**	14	k_path_n	**14**	12
k_ph_p	4	7	k_ph_n	**6**	7
k_poly_p	**19**	11	k_poly_n	**20**	21
k_t4p_p	**21**	10	k_t4p_n	**21**	8

Table 2. Benchmarks comparing the TWB and LWB for modal logic *K*

Benchmarks. The TWB was engineered to be a generic and flexible framework, rather than to produce highly optimised theorem provers, but it is important to compare it against other well established theorem provers. Since the TWB makes no logic-specific assumptions, the user has to make logic-specific optimisations using the hooks provided. Below, we compare the TWB with the LWB using the benchmark suite from [HS96].

Note 1. We ran our benchmarks on the following system: Hardware: Pentium 4 (2.4 Ghz), 1GB RAM, 1GB swap space; Software: Debian GNU/Linux OS, OCaml 3.09.2.

Table 2 show, for each class, how many formulae of each set could be solved. For each class, we generated all formulae with complexity up to 21 with a timeout of 100 seconds. We used the same set of formulae for each prover. For the LWB we recorded a failure either if it timed out after 100 seconds or if it failed with an error (`Error : lwb stack too small`). We did not investigate the reason for this error. For these benchmarks, we used an optimized version of our tableau calculus for *K* using simplification, semantic branching, back-jumping and a simple form of caching (tabling) [HS98].

The results are very encouraging: we highlight where the TWB does better or roughly the same as the LWB. The TWB can compete with the LWB for the logic *K*. The LWB used considerably more memory than the TWB for certain formula classes. Also, without caching, the TWB uses no more than 32MB of stack space, and virtually no heap space.

References

[Aba07] P Abate. *The Tableau Workbench: a framework for building automated tableau-based theorem provers*. PhD thesis, Australian National University, 2007.

[Dyc92] R Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *The Journal of Symbolic Logic*, 57(3):795–807, sep 1992.

[Fre95] J W Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, University of Philadelphia, 1995.

[GHLS05] O Gasquet, A Herzig, D Longin, and M Sahade. Lotrec: Logical tableaux research engineering companion. In *TABLEAUX'05*, pages 318–322, 2005.

[Heu96] A Heuerding. LWBtheory: information about some propositional logics via the WWW. *Logic Journal of the IGPL*, 4:169–174, 1996.

[HJ94] P. Hudak and M. P. Jones. Haskell vs. Ada vs. C++ vs. Awk vs. ... An Experiment in Software Prototyping Productivity. Technical report, Yale, July 1994.

[HS96] A. Heuerding and S. Schwendimann. A benchmark method for the propositional modal logics k, kt, and s4. TR-IAM-96-015, University of Bern, Switzerland, 1996.

[HS98] U Hustadt and R A. Schmidt. Simplification and backjumping in modal tableau. In *Proc. TABLEAUX'98* LNCS 1397, 1998.

[HS00] U. Hustadt and R. A. Schmidt. MSPASS: Modal reasoning by translation and first-order resolution. In *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference*, volume 1847 of *LNAI*, pages 67–71. Springer, 2000.

[HSZ96] A Heuerding, M Seyfried, and H Zimmermann. Efficient loop-check for backward proof search in some non-classical propositional logics. In *Analytic Tableaux and Related Methods*, pages 210–225, 1996.

[MGW95] A. Martin, P. Gardiner, and J. Woodcock. A tactic calculus - abridged version. *Formal Aspects of Computing*, 8(4):479–489, 1995.

[Sch98] S Schwendimann. A new one-pass tableau calculus for PLTL. In *TABLEAUX'98*, LNCS 1397:277–291. Springer, 1998.

[TH06] D Tsarkov and I Horrocks. Fact++ description logic reasoner: System description. In *IJCAR*, 2006.

[Wad92] P Wadler. The essence of functional programming. In *Proc. Principles of Programming Languages*, pages 1–14, Albequerque, New Mexico, 1992.

[Woo01] A Wood. Structure and interpretation of computer programs. *Journal Functional Programming*, 11(2):253–262, 2001.