Automated Synthesis of a Tableaux Theorem Prover for Classical Propositional Logic using Coq

Alexander Soen

Australian National University

Abstract. Proving or disproving that a formula is a theorem in a logic is an essential process in many areas of applied logic. A variety of logics utilize the tableau method to implement theorem provers [1]. We describe a framework which allows users to synthesise tableau-based theorem provers for various logics. Unlike the current systems in the literature, we implement this framework in Coq [2], an interactive theorem prover. Similarly to the literature's current tools for general logics, the framework allows the user to input a description of a logic, including non-classical logics, to produce a corresponding theorem prover. However, as the framework is implemented in an interactive theorem prover, it can be used to prove various properties about the synthesised theorem prover in Coq. The properties which can proven include: the correct correspondence between the rule set and the synthesis theorem prover, and the completeness and soundness of the rule set with respect to the logic being defined. We demonstrate this framework in Coq by implementing the standard tableau calculus for classical propositional logic.

1 Introduction

Many different implementations of the tableau method exists within the literature for specific logics. There exists some very effective domain specific theorem provers such as Fact++ [3] and MSPASS [4]. However, whenever a new tableau calculus is devised, trying to develop a corresponding theorem prover can be difficult without specific knowledge in programming. The current systems which try and accommodate for generic logic systems include LoTREC [5] and the Tableau Work Bench [6]. LoTREC accepts a semantic description of a logic to define a graph-based tableau [7]. The Tableau Work Bench allows the user to define tableau rules and specify a strategy to guide the proof search procedure which determines if a formula is a theorem. We choose to follow the Tableau Work Bench's input language as it directly corresponds to the tableau calculus the user describes.

Although both LoTREC and the Tableau Work Bench allows a user to describe the tableau rules to define a logic as a tableau system, there are no guarantees that the tableau rules correctly associates to the logic the user tries to describe. Currently, the user must prove this correspondence separately to using LoTREC or the Tableau Work Bench. However, even if the user does this,

there are still no guarantees that LoTREC or the Tableau Work Bench correctly translates the set of rules such that the tableau method defined correctly represents the logic. Furthermore, there is no way to prove properties regarding proof search when using LoTREC or the Tableau Work Bench.

We aim to implement a system similar to the Tableau Work Bench which additionally allows us to prove correctness of the rules which the user inputs with respect to the logic they are trying to define. Furthermore, we want the system to allow the user to prove properties about the tableau search procedure with respect to the defined logic. We choose to implement the framework in Coq for two main reasons. The first reason is the extraction mechanism of Coq. This allows us to transform Coq proofs and, importantly, functions into functional programs [8]. This allows us to prove various properties about the search procedure to determine if a formula is a theorem in a logic defined in Coq and its corresponding extracted program. The second reason is to couple the implementation and associated proofs of the theorem prover. For example, within the Coq environment, a user can prove that the rule set they give to the framework describes a specific logic. The framework then can guarantee that the generated theorem prover correctly corresponds to the logic the user is trying to describe with the rule set.

Although providing additional guarantees about the generated theorem prover requires the user to provide proofs to the system, many of these proofs are standard when defining a useful tableau calculi for a logic. For example, to show correctness of a proof search, the rules first must be shown to be sound and complete with respect to the logic it is describing. However, when defining a tableau calculi for a logic, a proof of soundness and completeness is needed to show that the tableau calculi correctly corresponds to that logic. Hence, before the implementation of a theorem prover for a tableau calculi, this property is already proven. For these logics, the proof of soundness and completeness just need to be translated to Coq in the framework.

In this paper, we provide a description of the implementation of a framework implemented in Coq which synthesises theorem provers when given a grammar of the formulas in a logic and a set of corresponding tableau rules. We demonstrate the framework with the standard grammar and tableau rules for classical propositional logic.

2 Preliminaries

In this section we outline the basic concepts on tableau, tableau for classical propositional logic and notation for the rest of the paper.

2.1 Tableau Calculi

A tableau calculus provides a decision procedure to determine if a formula is satisfiable through the decomposition of sets of formulae. Furthermore, the tableau method can be used to determine if a formula is valid in a specific logic.

More specifically, a tableau calculus consists of a finite set of rules which describe a logic, L. The tableau method can determine if formulae are L-valid through repeated application of the rule set to determine if the tableau is closed. Underlying the tableau method is a tree structure, where branches from nodes to nodes represent a rule application.

Definition 1. *Tableau Rule*

The rules of tableau calculus consist of a numerator $\mathcal{N}$ and either finitely many denominators $\mathcal{D}_i$ separated by vertical bars (1), or a $\times$ symbol signifying a terminating branch (2). The numerator an denominators are sets of formulae.

$$\frac{\mathcal{N}}{\mathcal{D}_1 | \cdots | \mathcal{D}_n} \quad (1) \qquad \frac{\mathcal{N}}{\times} \quad (2)$$

Each tableau rule has a set of main formulae which dictate the way the rule gets applied. These formulae are denoted as the principal formulae of the rule. The rest of the formulae are denoted as the side formulae of the rule.

The numerator and denominators of the rules are often represented as sets, multi-sets or lists depending on what logic is being represented by the tableau calculus. In this paper we will primarily be using the list representation as we are working with classical propositional logic.

Definition 2. *Tableau Calculus*

A tableau calculus is defined to be a finite set of tableau rules.

Definition 3. *Tableau*

A tableau for a formula set Γ is a tree of nodes where Γ is the root and all children of a node are applications of a rule on that node.

Definition 4. *Closed and Open Tableau*

We describe the relation between a closed tableau and satisfiability with respect to the tree in which a tableau expresses.

A tableau is closed if all the leaves of the tableau tree have the symbol $\times$. If a tableau is not closed, then it is open.

Definition 5. *A formula Γ is unsatisfiable if there exists a tableau of Γ which is closed.*

To apply a rule to a formula set Γ, the variables in the numerator $\mathcal{N}$ must be unified to match Γ. This can only occur if there is a set formulae in Γ which is an instance of the numerator of the rule. Then the denominator must be instantiated following the unification of the numerator. Each branch of the denominator act as sub-goals in showing the satisfiability of Γ.

We formalise the unification process when applying a rule to a formula as the following.

Definition 6. *Partition*

We define a partition to be a map from propositional variables, PV, to formulae, F, in a logic. Given a partition π, we define the evaluation of π on a

formula f, $\pi(f)$, to be the substitution of the propositional variables in f with respect to π. For example, in classical propositional logic suppose we have a partition π, which maps the propositional variable p to the formula $q \wedge r$, and a formula $f = p \vee p$. Then $\pi(f) = (q \wedge r) \vee (q \wedge r)$.

Definition 7. *Formulae Instance*

A formula p is an instance of another formulae q if there exists a partition π such that $\pi(q) = p$.

A set of formulae Γ is an instance of another set of formulae Δ if there exists a partition π such that for each formula $\delta \in \Delta$, there exists a formula $\gamma \in \Gamma$ such that $\pi(\delta) = \gamma$.

If a formulae Γ is an instance of the numerator of a rule R, we say that the rule R can be applied to the formula Γ. Further, given a partition π which makes Γ an instance of the numerator of a rule R, we call

$$\Gamma \setminus \pi(numerator(R)) \cup \pi(denominator(R))$$

an instance of the rule R on a set of formulae Γ.

Definition 8. *Applicable Partition*

An applicable partition of a set of formulae Γ with respect to another set of formulae Δ is a partition π where Δ is an instance of the set with respect to the partition π.

Definition 9. *Invertible Rule*

A rule ρ is invertible if and only if whenever there exists a closed tableau for an instance of its numerator, there exists a closed tableau for each branch in its denominator.

2.2 Classical Propositional Logic

We define classical propositional logic (CPL) as the following.

Definition 10. *Syntax of CPL*

Given a finite set of propositional symbols, PV, a formula in CPL is described by the following grammar (where $p \in PV$).

$$\varphi \ ::= \ p \mid \top \mid \bot \mid \neg(\varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \rightarrow \varphi)$$

We also define the other standard connectives for CPL as the following.

$$(\varphi \leftrightarrow \psi) \ \equiv \ ((\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi))$$

Definition 11. *CPL Models and Valuations of CPL Formulae*

We define a CPL Model to be a function $\vartheta : PV \rightarrow \{True, False\}$ [9]. The valuation of a formula under a model ϑ is defined recursively as the following.

$$\vartheta(\bot) = False$$

$$\vartheta(\top) = True$$

$$\vartheta(\neg(\varphi)) = \begin{cases} True & \text{If } \vartheta(\varphi) = False \\ False & otherwise \end{cases}$$

$$\vartheta(\varphi \wedge \psi) = \begin{cases} True & \text{If } \vartheta(\varphi) = True \text{ and } \vartheta(\psi) = True \\ False & otherwise \end{cases}$$

$$\vartheta(\varphi \vee \psi) = \begin{cases} True & \text{If } \vartheta(\varphi) = True \text{ or } \vartheta(\psi) = True \\ False & otherwise \end{cases}$$

$$\vartheta(\varphi \rightarrow \psi) = \begin{cases} True & \text{If } \vartheta(\varphi) = False \text{ or } \vartheta(\psi) = True \\ False & otherwise \end{cases}$$

Definition 12. *Satisfiability in CPL*

A CPL formula φ is satisfiable if and only there exists a CPL model ϑ such that $\vartheta(\varphi) = True$.

Definition 13. *Validity in CPL*

A CPL formula φ is valid if and only for all CPL models ϑ, $\vartheta(\varphi) = True$.

It follows that in CPL, a formula φ is valid if and only if its negation $\neg\varphi$ is not satisfiable.

We will omit parentheses for CPL formulae for convenience and clarity. The precedence will be from highest to lowest as defined in the definition.

3 Tableau for CPL

The tableau calculus we use for CPL requires formulae to be in negation normal form.

Definition 14. *Negation Normal Form for CPL Negation normal form for CPL, is a form for formulae such that they only consist of connectives in the following set, $\{\bot, \neg, \vee, \wedge\}$. We define the negative normal form of a formulae inductively.*

$$\begin{aligned} nnf(\top) &= \top \\ nnf(\neg\top) &= \bot \\ nnf(\bot) &= \bot \\ nnf(\neg\bot) &= \top \\ nnf(p) &= p \\ nnf(\neg p) &= \neg p \\ nnf(\neg\neg\varphi) &= nnf(\varphi) \\ nnf(\varphi \wedge \psi) &= nnf(\varphi) \wedge nnf(\psi) \\ nnf(\neg(\varphi \wedge \psi)) &= nnf(\neg\varphi) \vee nnf(\neg\psi) \\ nnf(\varphi \vee \psi) &= nnf(\varphi) \vee nnf(\psi) \\ nnf(\neg(\varphi \vee \psi)) &= nnf(\neg\varphi) \wedge nnf(\neg\psi) \\ nnf(\varphi \rightarrow \psi) &= nnf(\neg\varphi \vee \psi) \\ nnf(\neg(\varphi \rightarrow \psi)) &= nnf(\varphi) \wedge nnf(\neg\psi) \end{aligned}$$

It is known that for every CPL formulae, there is an equivalent negation normal form formula.

Definition 15. *Tableau Calculus for CPL*

Let the following rule set define the tableau calculus for CPL which we will be using in this paper.

$$(\bot)\frac{\bot\,;\,Z}{\times} \qquad (Id)\frac{p\,;\,\neg p\,;\,Z}{\times} \qquad (\wedge)\frac{\varphi\wedge\psi\,;\,Z}{\varphi\,;\,\psi\,;\,Z} \qquad (\vee)\frac{\varphi\vee\psi\,;\,Z}{\varphi\,;\,Z\,|\,\psi\,;\,Z}$$

It can be shown that these rules are invertible.

4 Implementation in Coq

In this section we detail the main restrictions we have in defining the framework in Coq instead of a traditional programming language. The framework is similar to the Tableau Work Bench which is implemented in O'Caml [6, 9].

4.1 Defining Functions

Coq defines a dependently typed functional programming language. However, unlike programming languages not in an interactive theorem prover (like O'Caml and Haskell), Coq requires the decidability of type-checking for all defined functions to allow for proofs [10]. As a result, all functions defined in Coq are required to be provably terminating, total and deterministic. Totality and deterministic functions are enforced through the Coq language when defining functions. However, to enforce termination, Coq checks the definition of the function to ensure that recursive calls of the function have a structurally (strictly) decreasing argument. For simple functions, sometimes the termination of a function can be automatically inferred by Coq. However, in general this requires the user to explicitly provide a proof of termination.

To prove termination of general recursive functions, the functions are defined using well-founded recursion where a proof of termination needs to be given. Defining a function this way requires two steps. The first is to prove that a relation is well-founded. The second is to prove that the function has a decreasing argument, with respect to the well-founded relation in the first part, in each of its recursive calls.

To prove a relation R is well-founded in Coq, we prove in Coq that `well_founded R` is a theorem. Underlying this in Coq is the notion of the accessibility of a relation, defined as `Acc` in Coq. The definition of a well-founded relation in Coq is equivalent to the definition that the relation does not have any infinite chains. This follows the intuition that there are no infinitely nested recursive calls in the function, that is the function terminates. Once the relation is proven to be well-founded, a general recursive function can be defined with the `Fix` keyword in Coq, where a proof that the recursive calls has a decreasing argument with respect to the well-founded relation is required.

In the implementation of the tableau-based theorem prover synthesiser, we use the well-founded relation for the depth of a tree to define various functions. We define the relation as the following.

Definition 16. *Depth Order Relation (DOR)*

We define the depth order relation with respect to the depth of trees where the depth of a tree is defined as the maximum number of edges between the root node and the tree's leaves. That is for trees t_1 *and* t_2*,*

$$t_1 \preceq_{DOR} t_2 \iff \textit{the depth of } t_1 \leq \textit{the depth of } t_2 \tag{3}$$

We prove that these relations are well-founded with respect to the specific data structures we define in Coq.

4.2 Code Extraction

Coq's ability to extract programs from proofs and functions in a theorem prover is one of the major motivations for implementing the synthesiser for tableau-based theorem provers in Coq. The advantages of extracting programs using Coq's extraction mechanism is that any property proven in Coq will still hold true after extraction. With this property, we can generate certified tableau-based theorem provers using the framework where the theorem provers generated are guaranteed to express the logic the input rules describe, once a proof is given [8]. An important consideration for the implementation of the framework is to define the data structures in `Type` and not `Prop` as proofs and functions defined in `Prop` will be removed after extraction.

5 Data Structures

In implementing the tableau-based theorem prover synthesiser, we define general notions of tableau and CPL in Coq. Additionally, we define a specific implementation of a tableau-based theorem prover for CPL with the rule set defined in section (3).

5.1 CPL in Coq

We define CPL in Coq similarly to definition (10) based on [11].

```
Inductive PropF : Type :=
  | Var  : string -> PropF
  | Bot  : PropF
  | Conj : PropF -> PropF -> PropF
  | Disj : PropF -> PropF -> PropF
  | Impl : PropF -> PropF -> PropF.
```

We further provide notation for the negation connectives for propositional logic with respect to the definition of `PropF`.

```
Definition Neg A := Imp A Bot.
```

We define a propositional variable as a map from a `string` to a `PropF`. We note that we need to define a propositional variable as a map from a type that is decidable with respect to equality to `PropF` formulae. We use the pre-defined lemma in Coq, `string_eq_dec` to prove that `PropF` is decidable with respect to equality. `PropF` needs to be decidable with respect to equality as we need to compute comparisons of `PropF` formulae in the framework defined.

5.2 Tableaux in Coq

Using the defined notion of CPL in section (5.1), we define general data structures to express a tableau calculus. We define the rules of a tableau calculus and nodes in tableau as the following.

```
Inductive Denom :=
  | Terminate.
Definition PropFSet := list PropF.
Definition Numerator := PropFSet.
Definition Denominator := sum (list PropFSet) Denom.

Definition Rule := prod Numerator Denominator.
Definition TableauNode := sum PropFSet Denom.
```

Where `prod` defines a product type and `sum` defines a sum type.

Notably, we define `Rule` in this context to be the respresentation of a numerator and denominator of a tableau rule which only contains the principal formulae. This becomes important when we try and generate the applicable of a rule with respect to a `PropFSet`.

We further explicitly define the derivation tree of a proof in the tableau calculus similarly to [12].

```
Inductive DerTree :=
  | Clf : DerTree
  | Unf : PropFSet -> DerTree
  | Der : PropFSet -> Rule -> list DerTree -> DerTree.
```

This data structure is similar to a general rose-tree. The main difference is that we distinguish between two types of leaves. `Clf` represents a closed branch in the tableau tree. As we are attempting to show the unsatisfiability of the formula in the root node, reaching this type of leaf signifies that no additional applications of rules need to be done. `Unf` represents a node in a tableau tree in which no rule has been applied to it. `Der` represents an inner node of a tableau tree. This type of node holds information on the set of formula it had, the rule that was applied to it, and the children it generates from that set of formula and rule.

It should be noted that `Unf` does not mean that the node in the tableau cannot be made open. A tableau with a `Unf` node is defined as opened if there

exists no rule in the tableau calculus which can expand that `Unf` node. Similarly we define a tableau to be closed if there exists a `DerTree` that is generated through correct rule application and all of its leaves are `Clf`.

However, as `DerTree` was defined recursively as a list of `DerTree`s, Coq was unable to automatically define a useful inductive scheme. We use the following induction scheme instead when using induction on the `DerTree` type.

```
Fixpoint DerTree_induction
  (PT : DerTree -> Type)
  (PL : list DerTree -> Type)
  (f_Clf : PT Clf)
  (f_Unf : forall x, PT (Unf x))
  (f_Der : forall x r l, PL l -> PT (Der x r l))
  (g_nil : PL nil)
  (g_cons : forall x, PT x -> forall xs, PL xs -> PL (cons x xs))
  (t : DerTree) : PT t.
```

This general induction scheme requires two predicates: one which works on a `DerTree` and one which works on a list of `DerTree`s. Using this induction scheme, we are able to use the standard sub-tree induction and depth induction when completing proofs in Coq.

We define the depth of a `DerTree` as one would expect.

```
Fixpoint depthDerTree (T : DerTree) :=
  match T with
  | Clf              => 0
  | Unf _            => 0
  | Der _ _ branches => 1 + maxList (map depthDerTree branches)
  end.
```

Where `maxList` takes the maximum natural number in a list and `map` is defined as the usual map function in functional programming languages.

6 Tableau-based Theorem Prover

To determine the satisfiability of a formula using the tableau method, tableau rules must be applied to determine if the tableau generated by the formula can be made closed. At each application of a rules, three main considerations need to be made [6, 9]. We denote any non-closed leaf in a tableau tree to be a goal of a tableau tree.

- **Node-choice:** determining which goal of a tableau to apply a rule to.
- **Rule-choice:** determining which rule to apply a goal.
- **Formula-choice:** determining which formula in a goal to apply a rule to.

Similarly, we create functions in Coq to allow the user to make each of these decisions. We note that any node in a `DerTree` which is of the type `Unf` is a goal of the `DerTree`. To define the functions to allow the user to make these choices, we define how to generate applicable partitions, how to apply a rule to a tableau node, and how to traverse a `DerTree` to make the choices in Coq.

6.1 Generating Applicable Partitions

When we apply a rule to a node of a tableau tree, we wish to generate all possible applications of the rule. It follows that each possible application of the rule is a formula-choice. To define rule application in Coq, the notion of a partition in definition (6) is encoded as a list of pairs.

```
Definition Partition := list (PropF * PropF).
```

Given a `Partition` π, the associated partition map is the map defined by each element of π. If (a, b) is an element of π, then in the map of π, a maps to b. Given the partial function defined in this way, if a `PropF` formula is not in the domain of the function, it is mapped to itself to give us a total function. For a `Partition` π we denote the partial function generated by π as $\mathcal{F}_{\pi}^{P}$ and the total function generated by π as $\mathcal{F}_{\pi}^{T}$.

We additionally define what it means to create a well-formed partition.

1. An empty partition is well-formed.
2. Let `xs` be a well-formed partition. `(a, b) :: xs` is well-formed if `a` is not in the domain or the co-domain of $\mathcal{F}_{\texttt{xs}}^{P}$.

Furthermore, we define a well-formed extension of a well-formed partition π_1 with another well-formed partition π_2 as the following. The extension of the partition π_1 with respect to the partition π_2 is the list π_1 appended to the list π_2. This extension is well-formed if $\text{Image}(\mathcal{F}_{\pi_1}^{P}) \cap \text{Domain}(\mathcal{F}_{\pi_2}^{P}) = \emptyset$ and $\text{Domain}(\mathcal{F}_{\pi_1}^{P}) \cap \text{Domain}(\mathcal{F}_{\pi_2}^{P}) = \emptyset$. This is equivalent to the notion of repeatedly adding elements to form a well-formed partition.

We define a function `extendPartition` in Coq to define well-formed partition extension.

```
Fixpoint extendPartition (p1 p2 : Partition) : option Partition.
```

The function returns `None` if the extension results in a non-well-formed partition, otherwise it returns the well-formed extension.

We define `partition_help` in Coq,

```
Fixpoint partition_help (scheme : PropF) (propset : PropFSet)
  (pi : Partition) : list (Partition).
```

`partition_help` generates all applicable partition tuples of a formula `scheme` with respect to a set of formulae `propset`. The function then returns all well-formed extensions of `pi` with respect to each applicable partition tuple generated to form a list. If all the applicable partitions are unable to extend `pi` to give a well-formed partition, the function returns an empty list.

With `partition_help` we iteratively define a process to generate all applicable partition tuples of a set of formulae with respect to another set of formulae. `getPartitions` is used to find the applicable partitions of a numerator of a rule with respect to a node of the tableau tree.

```
Fixpoint getPartitions_help (schema propset : PropFSet)
  (acc : Partition) : list Partition :=
    match schema with
    | nil   => acc :: nil
    | s::ss => flat_map
                (fun pi => getPartitions_help ss propset pi)
                  (partition_help s propset acc)
    end.

Definition getPartitions (schema propset : PropFSet) :=
  getPaririton_help schema propset acc.
```

Where `flat_map` is provided by Coq.

6.2 Tableau Rule Application

Given a `Rule` and a `Partition` we define a function which creates a `DerTree` inner node with respect to a `PropFSet`.

We first define `applyPartition`, which evaluates the map defined by a partition on a set of formulae.

```
Fixpoint applyPartition (propset : PropFSet) (pi : Partition).
```

This is simply the element-wise evaluation of $\mathcal{F}^T_{\texttt{pi}}$ on a `propset`. We similarly define `denoApply` which additionally takes into account the case in which the denominator of a rule is a terminating symbol. In this case, `denoApply` returns a terminating symbol.

Additionally we define the following function.

```
Fixpoint derTreeAppend (rule : Rule) (propset : PropFSet)
  (branches : list TableauNode) (acc : list DerTree)
  : option DerTree.
```

`derTreeAppend` defines a inner node with `rule` and set `propset` with the children generated by `branches` (with an accumulator to turn `TableauNode`s into `DerTree`s).

With these functions, we define `applyPartitionRuleD` which makes a `DerTree` inner node which is the result of a tableau rule being applied to a `PropFSet` (where the formula-choice of the rule application is defined by a given partition).

```
Definition applyPartitionRuleD (rule : Rule) (propset : PropFSet)
  (pi : Partition) : option DerTree.
```

The function assumes that the argument `pi` is a partition which will correctly allow the set of formulae `propset` be applied to tableau rule `rule`. The choice of partition `pi` determines the formula-choice when the rule is applied to the propset. Assuming that the arguments allow for valid rule application, if the

denominator of `rule` is a terminating symbol, `applyPartitionRuleD` returns a `Clf` node. Otherwise, the function will return a `Der` node with `propset`, `rule` and a list of children `Unf` nodes. The children nodes are the result of `rule` being applied to `propset` (with the formula-choice being decided by `pi`).

6.3 Traversing DerTree

To expand a tableau through rule application, we must traverse the `DerTree` to make a node-choice. To do this we define the function `getGoals` which returns all the goals of a `DerTree`.

```
Fixpoint getGoals (T : DerTree) : list DerTree.
```

We define a function which will identify which branch in a list of branches (which is in order from left-to-right in a tableau) has the n^{th} goal.

```
Definition traverseToNG (Ts : list DerTree) (n : nat) :
  option (list DerTree * DerTree * list DerTree * nat)
```

The function returns a tuple containing the following: the branches to the left of the branch which contains the n^{th} goal, the branch which contains the n^{th} goal, the branches to the right of the branch which contains the n^{th} goal, and the updated number of the goal with respect to the branch containing the goal.

We define a function which applies a function of type `DerTree -> DerTree` to the n^{th} goal of a `DerTree`.

```
Definition toBranchNG (T : DerTree) :
  (DerTree -> DerTree) -> nat -> DerTree.
```

We also define a function to find the n^{th} goal in a `DerTree`.

```
Definition getNGoal (T : DerTree) : nat -> option DerTree.
```

Both `toBranchNG` and `getNGoal` was defined using well-founded induction using the depth order relation (3).

We define the function to apply a rule to the n^{th} goal of a `DerTree`.

```
Definition applyRtoNG (T : DerTree) (rule : Rule) (n : nat).
```

`applyRtoNG` generates all possible ways a `Rule` can be applied to the n^{th} goal of a `DerTree`. The function gets the n^{th} goal of the `DerTree T` using `getNGoal`. It then generates all possible applicable partitions of the `Rule rule` with respect to the set of formulae `propset` with `getPartitions`. `applyPartitionRuleD` is then used to generate new nodes which are the n^{th} goal applied with the tableau rule `rule` (with respect to the formula-choice chosen by each partition generated by `getPartitions`). The function then returns a list of the `DerTree` T with the goal node replaced by each of the new nodes generated.

The function allows the user to make a rule-choice and a node-choice when applying a rule to the tableau. The remaining choice to be made is the formula-choice when applying a rule. `applyRtoNG` returns a list of all possible applications

of a rule on the n^{th} goal of a tableau. Thus to account for a formula-choice, we just pick from the returning list of `applyRtoNG`. We define the following function to select from a list of results.

```
Fixpoint pickNFApply (results : option (list DerTree)) (n : nat)
  : DerTree.
```

6.4 CPL Example

We demonstrate how `applyRtoNG` and `pickNFApply` can be used to prove the validity of formulae in CPL by proving that the following formula is valid.

$$(A \wedge (A \implies B)) \implies B \tag{4}$$

We convert the tableau rules for CPL detailed in section (3) to the `Rule` type in Coq, only including the principal formulae of each tableau rule. We alias each of the rules as the following in Coq.

```
Inductive CRule :=
  | BotC : CRule
  | IdC  : CRule
  | OrC  : CRule
  | AndC : CRule.
```

We define the function `getCRule` of type `CRule -> Rule`. This function returns the `Rule` tuple associated to each of the tableau rules aliased by the inductive type `CRule`. We define `applyCRtoNG` to be a specific implementation of `applyRtoNG` for `CRule`. To determine the validity of (4), we look at the formula's negation. The negation of (4) is converted into negation normal form such that the tableau calculus defined can be used to check the satisfiability of the formula. The procedure to convert the negation of (4) is defined in definition (14).

$$A \wedge ((\neg A \vee B) \wedge \neg B) \tag{5}$$

Thus, with (5), we have the following proof in Coq for tableau.

```
Definition cpl_example :=
  ((# "A") ∧ ((¬ (# "A")) ∨ (# "B")) ∧ (¬ (# "B"))).

Definition step1 := Unf (cpl_example :: nil).
Definition step2 := pickNFApply (applyCRtoNG step1 AndC 1) 1.
Definition step3 := pickNFApply (applyCRtoNG step2 AndC 1) 1.
Definition step4 := pickNFApply (applyCRtoNG step3 OrC 1) 1.
Definition step5 := pickNFApply (applyCRtoNG step4 IdC 1) 1.
Definition step6 := pickNFApply (applyCRtoNG step5 IdC 1) 1.
```

We can check the process of each step by using `Compute` with the function `getGoals` to check what is needed to be proven at each step, that is the leaves of the tableau tree which are not closed. By running `Compute getGoals step6.` we get `= nil : list DerTree` as the output showing that the tableau is closed. Thus it follows that as the negation of (4) results in a closed tableau, it is unsatisfiable. Thus as expected (4) is shown to be a theorem.

7 Future Work

In this paper, we have described a framework in Coq which can be extracted to give functions which allows the user to guide a tableau proof. This system this provides is an interactive theorem prover. There are two logical ways to extend the framework: first is to add automation and second is to add support for additional logics.

To extend the framework for automation, we need to consider the node-choice, rule-choice and formula-choice as defined in section (6). We can follow methods used by the Tableau Work Bench [6] to make decisions about these choices. Node-choice can be handled by a depth-first search, in the tableau when a rule is applied, the left most branch is searched first. Formula-choice is only important if the rules are invertible. If a rule is not invertible (definition (9)), a backtracking approach to rule application is needed. We further discuss the use of backtracking below when dealing with additional logics. To determine the rule-choice, the Tableau Work Bench requires users to input a strategy. This directs the proof search of the Tableau Work Bench, guiding which rule should be applied at a particular step in a tableau proof. That is, in the Tableau Work Bench the user gives the system a strategy in which rules will be evaluated to expand a tableau. An additional requirement of implementing automation in this framework would be to ensure that the proof search method terminates. Thus for the implementation in Coq, for each strategy a user provides, the proof search process of the tableau specified by the strategy must be proven to terminate by either the framework or the user. In general, proving that a strategy is terminating automatically cannot be automated. The process requires defining a well-founded relation in which the proof search function decrements at each recursive call, section (4.1).

To extend the framework for logics other than CPL, we must deal with rules which are not invertible. As mentioned above, formula-choice needs to be considered when rules are not invertible. If the application of an invertible rule leads to a closed tableau, the choice of formula in which the rule is applied to does not matter. However, in the case in which a tableau rule is not invertible, an incorrect selection of formulae in the application of a rule can cause a branch of the tableau to be impossible to close, even if there exists a combination of rule application which can make the branch closed. Thus, to ensure that a branch is truly unable to be closed, all combinations of formula-choice for a non-invertible rule must be tested. This can be done through backtracking. Whenever, a non-invertible rule results in an unclosed branch where no additional rules can be applied, the tableau backtracks. The tableau goes back to the state of the non-invertible rule and a different formula-choice is made until it either gives a closed branch, or there are no more formula-choices to be made, confirming that the branch is indeed opened.

8 Conclusion

In this paper we describe the first steps in implementing a framework which allows users to synthesise tableau-based theorem provers for various logics. We outline the differences which need to be considered when implementing the framework in Coq instead of typical programming languages. Specifically, we demonstrate how the framework can be used to make a tableau-based theorem prover for classical propositional logic and how it can be used to prove that a formula is a theorem in classical propositional logic.

References

1. D'Agostino, M., Gabbay, D.M., Hähnle, R., Posegga, J.: Handbook of tableau methods. Springer Science & Business Media (2013)
2. Barras, B., Boutin, S., Cornes, C., Courant, J., Filliatre, J.C., Gimenez, E., Herbelin, H., Huet, G., Munoz, C., Murthy, C., et al.: The Coq proof assistant reference manual: Version 6.1. PhD thesis, Inria (1997)
3. Tsarkov, D., Horrocks, I.: Fact++ description logic reasoner: System description. Automated reasoning (2006) 292–297
4. Hustadt, U., Schmidt, R.A.: Mspass: Modal reasoning by translation and first-order resolution. In: TABLEAUX. Volume 1847., Springer (2000) 67–71
5. del Cerro, L.F., Fauthoux, D., Gasquet, O., Herzig, A., Longin, D., Massacci, F.: Lotrec: the generic tableau prover for modal and description logics. In: IJCAR. Volume 1., Springer (2001) 453–458
6. Abate, P.: The Tableau Workbench: a framework for building automated tableau-based theorem provers. Citeseer (2007)
7. Castilho, M.A., del Cerro, L.F., Gasquet, O., Herzig, A.: Modal tableaux with propagation rules and structural rules. Fundamenta Informaticae **32**(3, 4) (1997) 281–297
8. Letouzey, P.: Extraction in coq: An overview. Logic and Theory of Algorithms (2008) 359–369
9. Kelly, J.D.: A revised tactic semantics for the tableau work bench. (2009)
10. Barthe, G., Forest, J., Pichardie, D., Rusu, V.: Defining and reasoning about recursive functions: a practical tool for the coq proof assistant. In: FLOPS. Volume 3945., Springer (2006) 114–129
11. van Doorn, F.: Propositional calculus in coq. arXiv preprint arXiv:1503.08744 (2015)
12. Dawson, J.E., Goré, R.: A new machine-checked proof of strong normalisation for display logic. Electronic Notes in Theoretical Computer Science **78** (2003) 20–39