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. Theorem provers which utilise the tableau method is a common method in which this is done for a variety of logics. We describe a framework in Coq which allows users to synthesise tableau-based theorem provers for various logics. The main novelty of using this framework as a tool to create theorem provers is the following. Firstly, to synthesis an interactive theorem prover using the framework, only the grammar of the formula and rule set of the logics is needed. Secondly, that the underlying implementation of the rule set used to define the logics of the theorem prover is in Coq. Subsequently, this allows the framework and the user to prove various properties about the synthesised theorem prover in Coq, such as 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++ and MSPASS (REFERENCE). 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 and the Tableau Work Bench (REFERENCE). (NEED DESCRIPTION OF LoTREC AND THE DIFFERENCE TO WHAT IT DOES AND WHAT IM DOING). 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.

Although both LoTREC and the Tableau Work Bench allows a user to input tableau rules to define a logic system as a tableau system, there is no guarantees that the tableau rules correctly associates to the logic system 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 is still no guarantee that LoTREC nor the Tableau Work Bench correctly translates these set of rules so the tableau method correctly represents the logic. Furthermore, there is no way to prove properties regarding proof search when using these the LoTREC and 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 and additionally allows the user to prove additional properties about the tableau search procedure of determining if a formula is a theorem. 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 (REFERENCE Extraction in Coq: an Overview). This allows us to prove various properties about the search procedure defined in Coq and its corresponding extracted program which determines if a formula is a theorems. The second reason is to couple the implementation of the theorem prover and proofs about the associated tableau calculi being defined for the theorem prover. That is, within the Coq environment, a user can prove that the rule set they give to the framework describes the logic. Then the framework can guarantee that the generated theorem prover correctly associates to the logic the user is trying to describe.

Although providing additional guarantees about the generated theorem prover requires additional input from the user, many of these additional requirements are required for the logic and its tableau calculi of the theorem prover being implemented to be useful. 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 said to describe. However, the proof for soundness and completeness is required to show any usefulness for a logic and its tableau calculi.

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 formulas in a logic and a set of 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. Further more, 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 a tableau calculus are expressed as sets, multi-sets or lists depending on the logic being expressed. We will express the tableau rules as a list as we are primarily working with classical propositional logic. A rule is composed of a numerator and a denominator. A numerator $\mathcal{N}$ is a set of formulae in the logical system L. A denominator is either a set of branches, $\mathcal{D}_i$, which are each sets of formulae in L or the symbol $\times$ signifying a closed tableau, indicated the termination of a branch. These rules are typically written as the following where (??) represents a rule with a denominator as a set of formulae and (??) represents a rule which results in a closed branch,

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

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.

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 2. *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 p, $\pi(p)$, to be the substitution of the propositional variables in p with respect to π.

Definition 3. *Formulae Instance*

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

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 Γ is an instance of rule R.

Definition 4. *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.

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. We describe the

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

Definition 5. *Closed Tableau*

A tableau is closed if each of its leaf nodes are closed, end in terminating symbol $\times$.

Definition 6. *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.*

Definition 7. *A formula* Γ *is unsatisfiable if there exists a tableau of* Γ *which is closed.*

2.2 Classical Propositional Logic

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

Definition 8. *Syntax of CPL*

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

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

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

$$(\varphi \leftrightarrow \psi) \ = \ ((\varphi \to \psi) \wedge (\psi \to \varphi))$$

Definition 9. *CPL Models and Valuations of CPL Formulae*

We define a CPL Model similarly to (CITE PIETRO and JACK) to be a function $\vartheta : PV \to \{True, False\}$. *The valuation of a formula under a model* ϑ *is defined recursively as the following.*

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

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

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

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

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

Definition 10. *Satisfiability in CPL*

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

Definition 11. *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. With

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 system for CPL we will use requires formulae to be in negation normal form.

Definition 12. *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.*

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

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

Definition 13. *Tableau Calculus for CPL*

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

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

If follows that each of 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 like how the Tableau Work Bench is implemented in O'Caml.

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 to allow for proofs (REFERENCE Defining and Reasoning About Recursive Functions: A Practical Tool for the Coq Proof Assistant). As a result, this additionally requires all functions defined in Coq to be provably terminating, total and deterministic. Totality and deterministic 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. However, in general this requires the user to provide a proof of termination.

To prove termination of general recursive functions, these functions are defined by using well-founded recursion. Defining a function this way requires two steps. The first is to prove a relation is well-founded. The second is to prove that the function has a structurally decreasing argument in each of its recursive calls.

To prove a relation R is well-founded in Coq, we prove in Coq that `well_founded R` holds. Underlying this in Coq, is the notion of accessibility of a relation which is defined as `Acc` in Coq. Well-foundedness is equivalent to the notion that the relation does not have any infinite chains. This follows the intuition that there are no infinitely nested recursive calls when a function respects this well-founded relation, 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 have a structurally decreasing argument with respect to the well-founded relation.

In the implementation of the tableau-based theorem prover synthesiser, we use two main well-founded relations to define general recursive functions: the length order of a list and the depth order of a tree like data structure. We define the relations as the following.

Definition 14. *Length Order Relation (LOR)*

We define the length order relation with respect to the length of lists. That is for lists l_1 and l_2,

$$l_1 \preceq_{LOR} l_2 \iff \textit{the length of } l_1 \leq \textit{the length of } l_2 \tag{3}$$

Definition 15. *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,

$$t_1 \preceq_{DOR} t_2 \iff \textit{the depth of } t_1 \leq \textit{the depth of } t_2 \tag{4}$$

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

4.2 Code Extraction

(REFERENCE Extraction in Coq: na Overview) 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. 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 (**??**).

5.1 CPL in Coq

We define CPL in Coq similarly to definition (**??**) based on (REFERENCE Doorne).

```
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 of 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 can define a propositional variable as a map from almost any decidable type to `PropF`. We use the pre-defined lemma in Coq, `string_eq_dec` such that we can prove that `PropF` is decidable. `PropF` needs to be decidable as we need to compare `PropF` types in the tableau-based theorem prover framework.

5.2 Tableau in Coq

Using the defined notion of CPL in Coq, we define general data structures to express a tableau calculus. We define the rule structure of tableau and a node in tableau.

```
Inductive Results :=
  | Closed.
Definition PropFSet := list PropF.
Definition Numerator := PropFSet.
Definition Denominator := sum (list PropFSet) Results.

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

Notably, we define `Rule` in this context to be the 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.

```
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 is open. A tableau of this form is defined as not closed if there exists no rule in the tableau calculus which can expand a `Unf` node. Similarly we define a tableau to be closed if there exists a `DerTree` that is generated through correct rule application and where 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, we use the general induction scheme to define induction on the `DerTree` type as the standard structural induction on a generic rose-tree. That is induction where we prove the proposition holds for leaves as the base case and for the inductive step, we assume that the proposition holds for all elements of a list of `DerTree` and prove that it holds for a node with that list of `DerTree`s as its children. We also use the other standard induction scheme for rose-trees where we for the base case we prove the proposition holds for the leaves again, but for the inductive case we assume that the proposition is true for all `DerTree`s with a tree depth less than or equal to a list of `DerTree`s and prove that it holds for a node with that list of `DerTree`s as its children.

We also note 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

At each step of determining the satisfiability of a formula using tableau, three main considerations need to be made. We denote any non-closed leaf in a tableau tree to be a goal of a tableau tree.

– **Node-choice:** determining which of goal of the tableau tree to apply a rule to.

– **Rule-choice:** determining which rule to apply to the goal.
– **Formula-choice:** determining which formula in the goal to apply the 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 first define how to generate partitions as per definition (**??**), how to apply a rule to a tableau node, and how to traverse a `DerTree` to make the choices in Coq.

6.1 Generating 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 (**??**) 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, $a \mapsto b$. Given this 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 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 also define `partition_help` in Coq.

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

The function generates all applicable partition tuples of a formula `scheme` with respect to a set of formulae `propset` and then attempts to form a well-formed extension of the partition `pi`. Only the well-formed are returned. That is, if all none of the applicable partitions extend `pi` in a well-formed way the function returns an empty list.

Now with `partition_help` we can iteratively define a process to generate all applicable partition tuples of a set of formulae with respect to another set of formulae. This will be 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) :=
  getParititons_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 will expand a node in a `DerTree` with respect to a `PropFSet`.

We first define `applyPartition`, a function 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 `propset`.

We also define the notion of set minus, `removeMultSet`.

```
Fixpoint removeMultSet (remove setprop : PropFSet) : PropFSet.
```

This function simply returns a `PropFSet` which is `setprop` set subtract `remove`.

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 respect to the accumulator.

Thus with these functions we define the following function which defines rule application on a set to define an inner node of a tableau tree.

```
Definition applyPartitionRuleD (rule : Rule) (propset : PropFSet)
  (pi : Partition) : option DerTree :=
    let inst := applyPartition (getNumerator rule) pi in
    let X := removeMultSet inst propset in
    match pi with
    | nil => None
    | _ => match (getDenominator rule) with
           | inr res => Some Clf
           | res     => derTreeAppend rule propset
                          (tableauAppend X (denoApply pi res)) nil
           end
    end.
```