Abstract

I prove the equivalence of tableau calculus and sequent calculus.

1 Introduction

It is essential to prove or disprove that a formula is a theorem in many areas of applied logic. This is done through automatic reasoning, theorem provers. One such versatile and successful type of theorem provers are tableau-based theorem provers. These are theorem prover that utilise the semantic tableau, a procedure which can determine if a formula is satisfiable, and hence if the formula is a theorem. We aim to build a tool that given a set of tableau rules, among other parameters, it will automatically generate a corresponding tableau-based theorem provers which is proven to be correct. We aim to do this for classical propositional logic first in the hopes that this can be extend for more complex logical systems in the future.

In this project we have the following,

1. Explored the formalisation of the equivalence of sequent calculi and tableau calculi in Coq.
2. Created a tableau-based theorem prover for classical propositional logic using Coq.

In the first part of the project, the formalisation of sequent calculi and tableau calculi was explored and how it could be encoded into Coq. Sequent calculi, like tableau calculi, can be utilised to determine whether a formula is satisfiable. Gentzen first introduced the "calculus of sequents" in "Untersuchungen uber das logische Schliessen" (Investigations into logical deduction - gerhard gentzen)). It is now a common system to use when reasoning about satisfiability of formula in classical propositional logic. It follows that there is a strong relationship between the notion of a closed tableau in the tableau calculus and a sequent being derivable in the calculus of sequents. By exploring the equivalence of sequent calculus and tableau calculus, a transformation from the semantics of sequent calculus and tableau calculus can be made in Coq. With this, userss of of our tool, the automatic synthesiser of tableau-based theorem provers, will not need prerequisite knowledge about the tableau calculus to use the tool. Instead, they can use the semantics of the common sequent calculus. (Not too sure if this is correct. Need more citations)

In the second part of the project, we attempt to make a generalised synthesiser for logical systems. We aim to make a tool kit in Coq which allows users to input logical rules and then extract a corresponding theorem provers which is proven to be correct with respect to the logical system the user proves the logical rules corresponds to.

This project is aimed to extend the work done in the Tableau Work Bench (ADD CITATION - Pietro). The Tableau Work Bench, implemented in O'Caml, exists as a "user-friendly framework for building automated tableau-based theorem provers". It allows users to give an input set of rules for their custom logical system in propositional logic to generate a corresponding tableau-based theorem prover. However, the generated tableau-based theorem provers generated are not guaranteed correctness.

Apart from the Tableau Work Bench, to implement a theorem prover for a custom logic system, assuming there is no pre-existing one already, one would have to implement and program the theorem prover from scratch.

Furthermore, if one wants to prove that a logical set of rules corresponds to a particular logical system they would have to prove this separately from their corresponding implementation of a theorem prover. This is a major issue on knowing if the implementation of the theorem prover correctly corresponds to the logical system the user wants it to describe.

We aim to developed a tool which will still allow users to quickly prototype and implement calculi they wish to test like the Tableau Work Bench while also providing an option to have the implementation proven to be correct. To simplify, in this project we only look at classical propositional logic.

2 Background

2.1 Tableau Calculus

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 logical system.

More specifically, a tableau calculus consists of a finite set of rules which describe the logical system, L. The tableau method can determine if formulae are valid with respect to L. Underlying the tableau method is a tree structure, where branches represent a rule application.

Definition 1. *Tableau Rule*

The rules of a tableau are expressed as sets, multi-sets or lists depending on the logics being expressed. We will express the tableau rules as a set as we are working within 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 $\perp$ signifying a closed tableau, indicated the termination of a branch. These rules are typically written as the following,

$$(\rho)\frac{\mathcal{N}}{\mathcal{D}_1|\cdots|\mathcal{D}_n} \qquad\qquad (\rho')\frac{\mathcal{N}}{\perp}$$

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. Then the denominator must be instantiated following the unification of the numerator. Each branch of the denominator acts as subgoals.

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 2. *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.*

A formula can be shown to be unsatisfiable if a tableau exists in which all branches have been expanded to be closed. Thus to show if a formula is valid in classical propositional logic, we show that its negation is unsatisfiable in tableau.

2.2 Sequent Calculus

A sequent calculus is a very useful tool in proof theory. Proving the validity of formulas in a sequent calculus is similar to the tableau method, however the node in a sequent calculus proof are a sequent of formulae instead of a set of formulae.

Definition 3. *Sequent A sequent is an expression of the following form,*

$$\Gamma ==> \Delta$$

Where Γ and Δ are a set, multi-set or list of formulae depending on the calculi being defined. ==> acts as an auxiliary symbol. We will express a sequent with lists.

Γ is denoted as the antecedent and Δ is denoted as the succedent. Both of these expressions could be empty.

Given the sequent,

$$A_1, \ldots, A_n ==> B_1, \ldots, B_n$$

This is equivalent to the following formula,

$$(A_1 \& \ldots \& A_n) \supset (B_1 \vee \ldots \vee B_n)$$

(Define more, seq rules, deduction tree, cut?)

2.3 Classical Propositional Logic

(Define tableau + sequent defs)

3 Proof Theory

The system of sequent calculus has been encoded into Coq based on the rules given by Floris van Doorn. First the notion of a sequent being a tuple of lists is defined, the left side and right side of a sequent. Given this, we encode the notion of sequent being derivable as a direct translation of the sequent rules.

Furthermore, the system of tableau calculus was encoded in a similar manner. A tableau is represented as a list. Then the notion of a closed tableau is established through a direct translation of the tableau rules.

To show that the system of tableau calculus is equivalent to the system of sequent calculus we aim to prove the following,

$$\text{closed } X \iff X = \Gamma \cup \neg\Delta \,\&\, \text{derivable } \Gamma ==> \Delta \tag{1}$$

This is proven to an extent in Coq. Currently the proof is reliant on the exchange lemma for the tableau calculus begin admitted.

The proof is also reliant on a slight modification of rules. However, equivalence of the rules with respect to the ones being used by Floris van Doorn can be achieved through the admissibility of the weakening lemma.

The admissibility of the structure rules, apart from the cut lemma, has been proven with Floris van Doorn's rules for sequent calculus by relying on a proof of the exchange lemma being admitted.

4 Tableau-based Theorem Prover

We define the process of creating a tableau tree through the recursive application of rules on nodes. This function for this process is then extracted using Coq's inbuilt tools to give a program which can be used to check if a formulae is valid with respect to the rules defined in Coq.

4.1 Data Structures

To implement the tableau-based theorem prover for classical propositional logic, we define the following data structures. We make a concious decision to define the data structures we use inside *Type* in Coq. This is because in the extraction of programs provided by Coq, anything defined in *Prop* (and not *Type*) get discarded and subsequently not extracted to a program.

We define a propositional logic formulae similarly to the definition found in Doorn's work (Reference Propositional Calculus in Coq).

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

We use *string* to represent propositional symbols in our formulae. With this, *PropF* can be proven to be decidable under equality using the already proven lemma that strings are decidable, *string_dec*.

General data structures for the tableau calculus are defined using *PropF*. The general data structures are mainly for defining a node of a tableau and the rules of a tableau system.

```
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 a *Rule* as only the principal formulae of a numerator and denominator of a tableau rule.

The decidability of each of these types are proven using induction.

We define the explicit derivation tree of a tableau proof. That is the tree of tableau nodes where each child of a node generated from an application of a tableau rule.

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

The data structure is similar to a general rose-tree. The main difference is that there are two possible leaf choices. One type of a leaf represents a closed branch in the tableau tree. The other type of a leaf represents a node which is not closed and possibly could be expanded by additional rule applications. The inner nodes of a *DerTree* also contains information about the rule which is applied to generate its children. We define that a tableau is closed if there exists a *DerTree* which is generated through correct rule applications and all of its leaves are closed, of type Clf.

Furthermore, as *DerTree* was defined recursively as a list of *DerTree*s, Coq was not able to infer a particularly useful induction scheme, that is an inductive principle used to prove properties about the type *DerTree*.

We prove that the following induction scheme is valid for *DerTree*.

```
Fixpoint DerTree_recursion (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.
```

Simply this recursion scheme is just how one would typically do structural induction on a rose tree, however the proposition for the branches of an inner node must be explicitly stated. That is, it formally states that for some property P, if the property is true for any type of leaf node and if P holds for all the children of an inner node implies it holds for the inner node, then the property P is true for all *DerTree*s.

Again the decidability of *DerTree* is proven using the previous decidability properties and induction using *DerTree_recursion*.

4.2 Implementing a Theorem Prover in Coq

The main difference when implementing a theorem prover in Coq than implementing a theorem prover in another programming language like O'Caml is that all the functions must be proven to be terminating. Functions defined in Coq can be classified in two main groups: functions defined with primitive recursion and functions defined with general recursion.

The case in which a function is defined with primitive recursion, Coq automatically infers its termination requiring no additional input from the user.

However, to define functions with general recursion the user must provide a proof to justify its termination.

This involves proving that a function has a decreasing property in each of its recursive calls with respect to a well-founded relation.

We use two main well-founded relations to prove termination for functions when defining the theorem prover in Coq.

The first is that the relation of a list with respect to length is well-founded.

```
Definition lengthOrder (A : Type) (xs ys : list A) := length xs < length ys.

Lemma lengthOrder_wf : forall A, well_founded (lengthOrder A).
```

The second is that relation of *DerTree* with respect to depth is well-founded. The depth of a *DerTree* is defined as the following.

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

Where *maxList* returns the maximum value of a *list nat*. Hence, we have the well-founded lemma for *depthOrder*.

```
Definition depthOrder T1 T2 := depthDerTree T1 < depthDerTree T2.

Lemma depthOrder_wf : well_founded depthOrder.
```

The two relations were proven to be well-founded through regular induction on lists as provided by Coq for *lengthOrder* and induction using *DerTree_recursion* for *depthOrder*. For the *depthOrder* proof, the explicit branch proposition used was:

$$P(bs) = \forall b,\ depthDerTree\ b < S\ (maxList\ (map\ depthDerTree\ bs)) \rightarrow Acc\ depthOrder\ b$$

Where *Acc* is as defined in Coq.

In the subsequent definitions of function defined using general recursion, the well-founded relation used to define the function will be stated and the function will be defined as if it was a *Fixpoint* function in this paper with keyword *GFixpoint* instead of *Fixpoint*, omitting its actual definition in Coq with the *Fix* operator for clarity.

4.3 Generating a Tableau Through Rule Application

To generate a *DerTree* which expresses a correct tableau, the application of a rule must be defined. The overall process we want defined is that given an initial leaf node with formulae set Γ, *Unf* Γ, through a series of rule applications a new *DerTree* is given which defines a tableau which can be determined to be closed or not. Thus determining if Γ is a satisfiable set of formulae or not assuming the application of rules is exhaustive.

We first define how a rule is applied to a *PropFSet* type before moving on to working with a *DerTree*. When applying a rule to node, a *PropFSet* which is not closed, we must consider the following,

1. If the rule is applicable to the node.
2. Which formulae in the node is the rule being applied to.

To check if a rule is applicable to a node, we must check if there exists a map from propositional variables to formulae such that the rule's numerator under the map is equal to a set of formulae in the node.

Definition 4. *An instance of a rule is denoted as the evaluation of a map to its numerator and denominator such that each of its propositional variables are instantiated. Generally, a formula p is and instance of a formula q if there exists a map ϕ from propositional variables to formulae such that $\phi(p) = q$.*

We denote that a rule R can be applied to a set of formulae Γ if there exists an instance R such that each of the formulae in the numerator of the instance of R exists in Γ. An instance of this type is denoted as an applicable instance of R for set Γ.

A map to give an instance of a rule is defined as a partition. The set of applicable instances of R for a set Γ is denoted as the set of applicable partitions.

If we consider all maps which allow a rule to be applied to a node, this is the combination of all the possible formulae choice we have when applying a rule. We want to get all possible applications of a rule in node to generate all the possible ways a tableau can be expanded.

An application of a rule R to a set of formulae Γ is defined as when all formulae in the numerator of an applicable instance I of R for the set Γ is replaced by the denominator of I for each branch.

To define how a rule is applied to a set of formulae *PropFSet* in Coq, we first define how to generate the list off all applicable partitions for a rule with respect to the *PropFSet*. The notion of a partition is defined as the following.

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

The method of finding the applicable partitions with respect to a numerator of a rule R and a set of formulae Γ is done in an iterative method. For each formulae in the numerator of a rule, we check if there is an instance of it in Γ. We extend this partition for each of the formulae in the numerator of R. This process give us a list of partitions which are all the applicable partitions for the rule R and set Γ. When extending partitions to account for additional formulae in the numerator, we also need to make sure that the partitioning map is consistent. That is if we already map a propositional variable a to a formula p, none of the propositional variables in p gets mapped as a result of the rest of the partition. We implement a function in Coq that generates all the possible consistent partitions with respect to two *PropFSet*s as the following.

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

Definition getPartitions schema propset :=
  (getPartitions_help schema propset nil).
```

Where *partition_help* finds all the maps of instances of s in *propset* which can extend the partition *acc* consistently in each of the recursive calls of *getPartitions_help*. *applyParitition* simply applies the partition *pi* to all formulae in *ss*, updating the propositional variables in propset every recursive call.

getParition_help had to be defined with the *lengthOrder* relation to prove that the *applyPartition ss pi* decreases in size every recursive call to prove termination.

Given a function to generate partitions, we now wish to define a function which allow us to apply rules to the leaves of a *DerTree* data structure, that is expand out a tableau tree. We define the nodes in the *DerTree* which can have rules applied to as the leaves with the *Unf* constructor.

We first define a function which will apply a function from *DerTree* to *DerTree* to the n^{th} un-closed leaf of a *DerTree* data structure. That is we define a function which will apply a change to and un-closed leaf but leave the rest of the tree the same. We denote the n^{th} un-closed leaf as the n^{th} goal which needs to be satisfied when attempting to show a formulae set is unsatisfiable.

To operate on the n^{th} goal of a *DerTree* we need to determine how to traverse a *DerTree* to get to the n^{th} goal. To do this, we define a function to get the goals of a *DerTree* as the following.

```
Fixpoint getGoals (T : DerTree) : list DerTree :=
  match T with
  | Der _ _ branches => match branches with
                        | nil => nil
                        | _ => flat_map getGoals branches
                        end
  | Clf => nil
  | Unf _ => T :: nil
  end.
```

The function ignores all closed leaves as goals as the goal to show that a formula is unsatisfiable is to make all the leaves of the expansion of the tableau closed, that is the closed leaves has already satisfied this condition. To traverse the tree, we must determine which branch to go down when starting from an inner node. To do this, the number of goals in each branch are checked.

```
Fixpoint traverseToNG_help
  (Ts : list DerTree) (n : nat) (acc : list DerTree) :=
    if gt_dec n (length Ts) then None else (
    if eq_nat_dec n 0 then None else (
    match Ts with
    | nil => None
    | x::xs => match x with
               | Clf => traverseToNG xs n (acc ++ (x::nil))
               | _ => let xchild := (length (getGoals x)) in
                      if le_dec n xchild then Some (acc, x, xs, n)
                      else traverseToNG xs (minus n xchild) (acc ++ (x::nil))
               end
    end)).

Definition traverseToNG (Ts : list DerTree) (n : nat) :=
  traverseToNG_help Ts n nil.
```

traverseToNG defines a function which given a list of branches from an inner node and a goal number will return information about the branches with respect to the aimed goal. Assuming that the list of branches has an n^{th} goal, *traverseToNG* returns a tuple containing the branches before the desired goal, the branch containing the n^{th} goal, the branches after the desired goal and the updated goal number with respect to the branch which contains the goal.

(Might add proof to show that this is indeed true)

With *traverseToNG*, we define a function to apply a function from *DerTree* to *DerTree* on the n^{th} goal.

```
GFixpoint toBranchNG (T : DerTree) (f : DerTree -> DerTree) (n : nat) :
  DerTree :=
  match T with
  | Der propset rule branches => match traverseToNG branches n with
                                  | None => T
                                  | Some (acc, x, xs, n') =>
                             Der propset rule (acc ++ (toBranchNG x f n') :: xs)
                                  end
  | _ => f T
  end.
```

The function simply recursively traverses a *DerTree* until a leaf is reached in which its argument f will be evaluated to the leaf giving a new tree. We also define a function to get the n^{th} goal in a *DerTree*.

```
GFixpoint getNGoal (T : DerTree) (n : nat) :=
  match T with
  | Der _ _ branches => match traverseToNG branches n with
                        | None => None
                        | Some (_, x, _, n') => getNGoal x n'
                        end
  | _ => if eq_nat_dec n 0 then None else Some T
  end.
```

These two function are defined using the *depthOrder* relation to show that the *DerTree* argument being passed down every recursive call is decreasing.

With these two functions, the following function is defined to apply a rule to the n^{th} goal.

```
Definition updateLeaf (T : DerTree) : (DerTree -> DerTree) := fun _ => T.

Definition applyRtoNG (T : DerTree) (rule : Rule) (n : nat) : list DerTree :=
  match getNGoal T n with
  | None => None
  | Some goal =>
    match goal with
    | Unf propset =>
      match getPartitions (getNumerator rule) propset with
      | nil => None
      | Pi => match optionSucMap _ _ (applyPartitionRuleD rule propset) Pi with
              | None => None
              | Some newNodes =>
              Some (map (fun x => toBranchNG T x n) (map updateLeaf newNodes))
              end
      end
    | _ => None
    end
  end.
```

Where *optionSucMap* is a mapping functions which only returns successful results, and if there are no successful results it returns *None*. The function does the following,

1. Gets the n^{th} goal of the *DerTree* T, G.

2. Makes sure the goal is infact an un-closed leaf.
3. Generates all applicable partitions of the *Rule rule* with respect to G.
4. Generates all the expansions of the tree with respect to each applicable partition.

With *applyRtoNG*, we can now define a simple interactive theorem prover. When we want to prove that a formula is unsatisfiable using tableau we need to consider three variables when expanding a tableau $DerTree$.

- **Node-choice:** determining which of the goals in the $DerTree$ to expand.
- **Rule-choice:** determining which rule to apply to the goal.
- **Formula-choice:** determining which formula to apply the rule to.

applyRtoNG requires the user to specify the rule and the goal the tableau expands to. It returns a list of $DerTree$s if the rule is applicable to the goal. Each of these results are just the combination of formula-choices for the application of a rule. Hence, by simply defining a function which allows a user to selects one of these results, the user can drive a tableau proof by take care of the possible choices when expanding a tableau. We define the following function so that the user can select a formula-choice after *applyRtoNG* returns all possible formula-choices.

```
Fixpoint pickNFApply (results : option (list DerTree)) (n : nat) : DerTree :=
  match results with
  | None => None
  | Some nil => None
  | Some (x::xs) => match n with
                    | 0 => None
                    | 1 => Some x
                    | S n' => pickNFApply (Some xs) n'
                    end
  end.
```

4.3.1 Example of a User Guided Proof

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

$$(A \wedge (A \implies B)) \implies B \tag{2}$$

The following set of tableau rules for classical propositional logic are used to prove this.

$$(Id)\frac{p\,;\,\neg p\,;\,X}{\bot} \qquad (\wedge)\frac{\phi \wedge \psi\,;\,X}{\phi\,;\,\psi\,;\,X} \qquad (\vee)\frac{\phi \vee \psi\,;\,X}{\phi\,;\,X \,|\, \psi\,;\,X}$$

First, these rules are converted 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 :=
  | IdC : CRule
  | OrC : CRule
  | AndC : CRule.
```

With function *getCRule* : *CRule* → *Rule* which returns the *Rule* tuple associated to each of the tableau rules aliased by the inductive type *CRule*. To determine the validity of Equation 2, we look at the formula's negation. The negation of Equation 2 is converted into negation normal form such that the rules defined are applicable, and thus operate on a logically equivalent formula to the negation of Equation 2 to determine the satisfiability of the formula. The converted formula to negation normal form is the following.

$$A \wedge ((\neg A \vee B) \wedge \neg B) \tag{3}$$

Thus, with Equation 3, we have the following proof in Coq for tableau.

$$Definition\ cpl\_example\ :=\ (\#"A"\ \vee\ ((\neg\,(\#"A"))\ \vee\ (\#"B"))\ \wedge\ (\neg\,(\#"B"))).$$

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

(Different types of rule application, to SetPropF, DerTree etc)

(Well-founded Relations)

(Strategy language)

(Defining a terminating function for tree search)

(Backtracking and further properties)