Extracting the Resolution Algorithm from a Completeness Proof for the Propositional Calculus

Robert Constable and Wojciech Moczydłowski

{rc, wojtek}@cs.cornell.edu
Department of Computer Science, Cornell University, Ithaca, NY 14853, USA

Abstract. We prove constructively that for any propositional formula ϕ in Conjunctive Normal Form, we can either find a satisfying assignment of true and false to its variables, or a refutation of ϕ showing that it is unsatisfiable. This refutation is a resolution proof of $\neg\phi$. From the formalization of our proof in Coq, we extract Robinson's famous resolution algorithm as a Haskell program correct by construction. The account is an example of the genre of highly readable formalized mathematics.

1 Introduction

Recently Jean Gallier [1] gave a simple constructive proof that the resolution method for the propositional calculus is complete – noting that other proofs in the literature [2–5] are by contradiction, hence weaker. Gallier's method is to provide an explicit resolution algorithm and a correctness proof for it.

We establish a slightly stronger result in a different way, by giving a simple constructive proof, formalized in Coq. The Coq system enforces constructivity and extracts from this proof an efficient resolution program (in Haskell or OCaml). Our proof is the kind of argument one would see in a logic textbook. Coq guarantees that it is constructive – which is not as well established in Gallier's presentation. The result is also stronger because executable code is extracted.

Our presentation is an instance of the new genre of *formalized mathematics* which we have been promoting, in both type theory [6, 7] and set theory [8]. A salient feature of this genre is precise clarity. Moreover, the definitions serve a dual purpose, defining data types as well as mathematical concepts. Likewise, the proofs provide justifications as well as algorithms. We strive to make the account more readable than ordinary mathematical texts. If we succeed, then there will be no question that formal text is a more useful way to present certain results in mathematics. The reader can judge these claims directly.

2 The Resolution Method

The resolution method applies to formulas in Constructive Normal Form (CNF), such $(P \vee Q) \wedge (\neg P) \wedge (\neg Q \vee P)$. These are built from variables (atoms) such as $P, Q, R, \ldots$ or negations of them. These positive or negative variables are called *literals*. Disjunctions of them are called *clauses*, so a CNF formula is a conjunction of clauses built out of literals.

Resolution is an attempt to *refute* a CNF formula ϕ. To refute is to show that the formula cannot be true no matter what truth values, $true$ or $false$, are assigned to the variables. What is so noteworthy is that a formula can be refuted by systematically applying one rule to its clauses over and over. The rules is called *resolution*; it resolves two clauses into a new one. Refutation starts with two clauses of ϕ and produces a new one by resolution and then is applied to either original clauses from ϕ or newly created ones.

Our result is that given a CNF formula ϕ, we can either refute it by producing a refutation r or show that it is satisfiable by producing an assignment of truth values, $true$, $false$, to the variables. We also show that if r refutes ϕ, then ϕ is not satisfiable. Thus $\neg\phi$ is valid, and indeed, the refutation r can be seen as a proof of $\neg\phi$.

2.1 Prerequisites

Booleans, denoted by *bool*, consist of two elements: $false$ and $true$. Natural numbers are denoted by *nat*. The standard operations on booleans are called *andb*, *orb* and *negb*. The notation *list A* stands for lists with elements coming from A. We denote the empty list by $[]$ and a list consisting of elements $a_1, a_2, ..., a_k$ by $[a_1, \ldots, a_k]$. The concatenation of lists l_1 and l_2 is denoted by $l_1@l_2$.

We will use the `typewriter font` for concepts in Coq. Our choice of names mostly coincides with its standard library. Lists are an important exception: in Coq, the list $[a_1, a_2, \ldots, a_k]$ is denoted by `a1::a2::...::ak::nil` and $l_1@l_2$ by `l1 ++ l2`. Thus in particular $[a]@l$ is rendered in Coq as `a::l` and the empty list $[]$ by `nil`,

2.2 Literals, clauses, formulas and valuations

We are interested in formulas in Conjunctive Normal Form (CNF). To build their representation in Coq, we represent propositional variables $P, Q, \ldots$ as natural numbers. To fix our attention, in the examples P, Q, R are represented by $1, 2$ and 3, respectively. A *literal* is either an atom or its negation. We represent the former case by labelling the atom with a tag *pos* and the latter by labelling it with a tag *neg*. A *clause* is a list of literals and a (CNF) *formula* is a list of clauses. For example, the formula $\phi \equiv (P \vee R) \wedge \neg P \wedge Q \wedge (P \vee \neg Q \vee \neg R)$ is represented by the list $[[pos\ 1, pos\ 3], [neg\ 1], [pos\ 2], [pos\ 1, neg\ 2, neg\ 3]]$. The function *notlit* for a given literal returns its negation: $notlit(pos\ n) = neg\ n, notlit(neg\ n) = pos\ n$.

The formalization of these definitions in Coq follows. Anything between `(* *)` characters is a comment.

```
(* lit is a disjoint union nat + nat.
   Its components are natural numbers labelled by pos and neg *)
Inductive lit : Set :=
```

```
        pos : nat -> lit
      | neg : nat -> lit.

Definition notLit (l : lit) : lit :=
  match l with
      pos n => neg n
    | neg n => pos n
  end.

Definition clause := list lit.

Definition form := list clause.

Definition P : nat := 1.
Definition Q : nat := 2.
Definition R : nat := 3.
Definition phi : form := (pos  P:: (pos R) :: nil) ::
                         (neg P :: nil) ::
                         (pos Q:: nil) ::
                         (pos P :: (neg Q) ::(neg R) :: nil) ::
                         nil.
```

A *valuation* assigns booleans to atoms in a formula. We represent valuations as lists of natural numbers. The intended meaning, captured in the formal definitions below, is that the valuation v assigns *true* to the atom represented by a number n, if n is an element of v. For example, the valuation $\{P \rightarrow false, Q \rightarrow true, R \rightarrow true\}$ is represented as $[2, 3]$.

```
Definition val := list nat.
```

The value of a literal *pos* n under a valuation v is equal to *true* if n is an element of v. The value of a negated literal, *neg* n, is equal to $false$ if n is not an element of v.

```
(* valLit : lit -> val -> bool *)
Definition valLit (l : lit) (v : val) :=
           match l with
               pos n => elemL nat eqNat n v
             | neg n => negb (elemL nat eqNat n v)
           end.
```

The value of a clause c under the valuation v is an element of *bool*. It is defined by recursion on c. If c is empty, the value is $false$. Otherwise, if c is a list with the first element x and the rest of its elements denoted by xs, it is a disjunction of the value of the literal x and the value of xs. The value of a formula f is defined similarly.

```
(* valClause : clause -> val -> bool
```

```
   valClause nil v = false
   valClause (x::xs) v = orb (valLit x v) (valClause xs v)
*)
Fixpoint valClause (c : clause) (v : val) { struct c } : bool :=
         match c with
              nil => false
            | x :: xs => orb (valLit x v) (valClause xs v)
         end.

Fixpoint valForm (f : form) (v : val) { struct f } : bool :=
         match f with
              nil => true
            | x :: xs => andb (valClause x v) (valForm xs v)
         end.
```

Note that the "empty" formula is always true, while the "empty" clause is always false. We will sometimes say that the valuation v *falsifies* a formula f to mean that $valForm(f)(v) = false$.

A formula f is *satisfiable* if there is a valuation which sets it to true. Or, in other words, if the set of valuations v such that $valForm(f)(v) = true$ is not empty. In Coq's type theory, truth of a proposition is equivalent to the non-emptiness of a corresponding type, so the following definition captures satisfiability correctly.

```
Definition satisfiable (f : form) := { v : val | valForm f v = true }.
```

A reader unfamiliar with type theory can think about `{ v : val | valForm f v = true }` as a (witness-providing) existential quantifier:

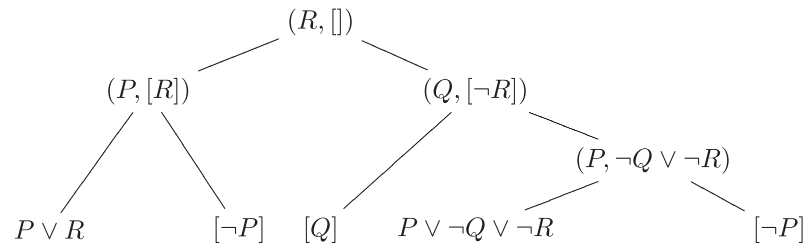
$$\texttt{\{ v : val | valForm f v = true \}} \approx \exists v : val.\ valForm(f)(v) = true$$

2.3 Resolutions

The definition of the resolution tree we use is taken from [1]. Structurally, it is a binary tree. Its leaves are labelled with clauses. Its nodes are labelled with pairs consisting of a literal and a clause.

```
Inductive resol : Set :=
        leaf : clause -> resol
      | node : lit -> clause -> resol -> resol -> resol.
```

For example, the tree T defined as[1]:



is represented as:

```
(* The prefix ex stands for ''example'' *)
Definition exTree : resol :=
 node (pos R) (nil)
   (node (pos P) (pos R::nil)
      (leaf (pos P::(pos R)::nil))
      (leaf (neg P::nil))
   )
   (node (pos Q) (neg R::nil)
      (leaf (pos Q::nil))
      (node (pos P) (neg Q::(neg R)::nil)
         (leaf (pos P::(neg Q)::(neg R)::nil))
         (leaf (neg P::nil))
      )
  ).
```

Note that each node in a tree is labelled with a clause. Let us denote this clause by `clauseR`:

```
Definition clauseR (r:resol) : clause := match r with
     leaf c => c
   | node l c r1 r2 => c
   end.
```

The *premises* of a tree are the clauses at its nodes.

```
Fixpoint premises (r : resol) {struct r}: list clause := match r with
       leaf c => c::nil
     | node l c r1 r2 => (premises r1) ++ (premises r2)
   end.
```

So the premises of T are $[P \vee R, [\neg P], [Q], P \vee \neg Q \vee \neg R, [\neg P]]$:

```
Lemma ex1 :
```

[1] To increase readability, we render clauses with more than one element in a traditional way instead of their list representation.

```
premises exTree =
 (pos P::(pos R)::nil)::
 (neg P::nil)::
 (pos Q::nil)::
 (pos P::(neg Q)::(neg R)::nil)::
 (neg P::nil)::
 nil.
```

The correctness restriction makes the trees more meaningful. Intuitively, each node should correspond to a resolution step. A clause c at the node results from the clauses c_1, c_2 at its children by removing l form c_1, the negation of l from c_2 and retaining the rest of c_1, c_2. So $c = (c_1 - \{l\}) \cup (c_2 - \{\neg l\})$.

In the formal definition, `eqCS` denotes equality of clauses treated as finite sets and `elemC l c` checks whether the literal l is an element of the clause c.

```
Fixpoint correctR (r : resol) : bool := match r with
    leaf c => true
  | node l c r1 r2 =>
       andb (andb (correctR r1) (correctR r2))
            (andb
                 (andb (elemC l (clauseR r1))
                       (elemC (notLit l) (clauseR r2)))
                 (eqCS c (removeC l (clauseR r1) ++
                          (removeC (notLit l) (clauseR r2)))))
         end.
```

Resolutions therefore are these trees that satisfy the correctness condition.

```
Definition res : Set := { r : resol | correctR r = true }.
```

We can easily show that $correctR(exTree) = true$ and define an element $exRes$ of res corresponding to the tree $exTree$.

```
(* p denotes the proof that correctR exTree = true *)
Definition exRes : res := exist _ exTree p.
```

Having defined the resolution trees, we need to relate them to formulas. A tree r *corresponds* to a formula f if the premises of r are a subset of the clauses of f. A tree r *refutes* a formula f if it corresponds to f and its root is labelled with the empty clause. We call a formula *refutable*, if there is a resolution tree which refutes it.

```
(* subL clause eqC f1 f2 checks whether f1 is a subset of f2 *)
(* proj1_sig, given a resolution, returns the underlying tree.
   So proj1_sig exRes = exTree. *)

Definition corresponds (r : res) (f : form) :=
     subL clause eqC (premises (proj1_sig r)) f.
```

```
Definition refutes (r: res) (f : form) :=
 andb (corresponds r f)
      (eqC (clauseR (proj1_sig r)) nil).
```

The proof that the resolution tree $exRes$ refutes ϕ is straightforward.

```
Lemma ex2 : refutes exRes phi = true.
```

2.4 Refuted formulas are unsatisfiable

To show that we have chosen our definitions correctly, we prove the following theorem:

Theorem 1. *For all resolutions r and formulas f, if r refutes f, then f is not satisfiable.*

In other words:

```
Theorem resres : forall (r : res) (f : form),
     refutes r f = true ->
          satisfiable f -> False.
```

Proof (Sketch). Take a resolution r, suppose that r refutes f and take a valuation v which sets f to $true$. We know that the root of r is labelled by the empty clause. In this situation we can find a clause c in the premises of r such that v sets c to false. We do this by starting from the root of r and proceeding down, choosing at the node labelled by l the left child if $v(l) = false$ and the right one otherwise. We take c to be the clause labelling the leaf we end at. Furthermore, we collect on the way literals in the following way: if a node is labelled by l and $v(l) = false$, we collect l; otherwise we collect the negation of l. Let d denote the resulting clause. We can easily prove that c is a subclause of d and that v falsifies d. Thus also $v(c) = false$.

Since r refutes f, the clause c is among the clauses of f. It is therefore easy to see that the value of f under v must be false as well. Therefore $valForm(f)(v) = false$, by the assumption about v we also have $valForm(f)(v) = true$ and since $true \neq false$, we get the claim.

We will now show how to make this proof precise. First, we formalize the process of "proceeding down, starting from the root of r", which resulted in the clause c. For the examples, we use our valuation $\{P \to false, Q \to true, R \to true\}$.

```
Fixpoint contraClause (r : resol) (v : val) { struct r} : clause :=
     match r with
         leaf c => c
        |node l c r1 r2 =>
              if valLit l v then contraClause r2 v
                            else contraClause r1 v
```

```
     end.

Definition exVal : val := Q::R::nil.

Lemma ex3 : contraClause exTree exVal = pos P::(neg Q)::(neg R)::nil.
```

Lemma 1. *If r corresponds to f, then the clause picked using $contraClause$ is equal to one of the clauses of f.*

```
Lemma cc1 : forall (r : res) (f : form) (v : val),
       corresponds r f = true ->
           elemL _ eqC (contraClause (proj1_sig r) v) f = true.
```

Second, we formalize "collecting literals on the way":

```
Fixpoint lontraClause (r : resol) (v : val) { struct r} : clause :=
     match r with
         leaf c => nil
        |node l c r1 r2 =>
              if valLit l v then (notLit l)::lontraClause r2 v
                            else l::(lontraClause r1 v)
     end.

Lemma ex4 : lontraClause exTree exVal = neg R::(neg Q)::(pos P)::nil.
```

Lemma 2. *For any resolution r and valuation v, v falsifies* `lontraClause r v`.

```
Lemma lv1 : forall (r : res) (v : val),
     valClause (lontraClause (proj1_sig r) v) v = false.
```

These definitions enable us to state and prove the main lemma:

```
Lemma lc1 : forall (r : res) (f : form) (v : val),
   let tree := proj1_sig r in
        subL _ eqLit (contraClause tree v)
             ((clauseR tree) ++ (lontraClause tree v)) = true.
```

Proof. Straightforward induction on r.

With Lemma $lc1$ at hand, we show

```
Lemma lv2 : forall (r : res) (f : form) (v : val),
     refutes r f = true ->
          valClause (contraClause (proj1_sig r) v) v = false.
```

Proof. Since r refutes f, the root of r is labelled by the empty clause. Let c denote $contraClause(r)(v)$ and let l denote $lontraClause(r)(v)$. By Lemma $lc1$, $c \subseteq l$. By Lemma $lv1$, $valClause(l)(v) = false$. The claim easily follows.

Now we can show formally what we just sketched at the beginning of this section:

The proof of Theorem 1 Take any f, r refuting f and a valuation v. Suppose the value of f under v is *true*. Let c denote $contraClause(r, v)$. By Lemma $lv2$, the value of c under v is false. By Lemma $cc1$, c is one of the clauses of f, thus also the value of f under v is false, which contradicts our assumption. Therefore f is not satisfiable.

2.5 Graft and percolate

There are two operations on the trees which we will use in the proof of the final theorem. Both are taken from [1].

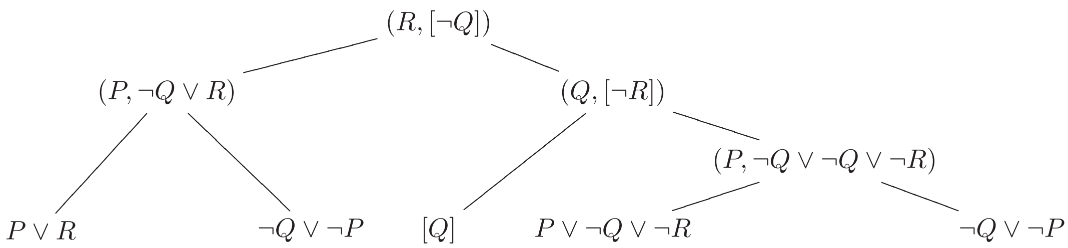
The *percolate* function takes a tree r, a clause c and a literal l as its arguments. It appends l to all clauses in the premises r which are equal (as finite sets) to c. It further "percolates" l up the tree: it travels towards the root and appends l to all clauses labelling nodes on the way. It stops when it either reaches the root or the node which utilizes l in the resolution step. The formal definition follows; the helper function `percolate0` returns additionally a boolean value, set to `false` if the percolating process is to continue and to `true` otherwise.

```
(* (fun x y z => M) is Coq's notation for functional abstraction:
   (fun x y z => M) is a function which given arguments x y z returns M *)
Fixpoint percolate0 (r : resol) (a : clause) (l : lit)
                            { struct r} : resol * bool :=
 match r with
    leaf c => if eqC c a then (leaf (l::c), false) else (r, true)
   |node resL c r1 r2 =>
      (fun p1 p2 cr  => (node resL (fst cr) (fst p1) (fst p2), (snd cr)))
        (percolate0 r1 a l)
        (percolate0 r2 a l)
        (let p1 := percolate0 r1 a l in
         let p2 := percolate0 r2 a l in
         let b1 := snd p1 in
         let b2 := snd p2 in
         let eqll := eqLit resL l in
         let eqlnl := eqLit resL (notLit l) in
           match (b1, b2) with
            (true, true) => (c, true)
           |(true, false) => if eqlnl then (c, true)
                             else ((l::c), false)
           |(false, true) => if eqll  then (c, true)
                                     else ((l::c), false)
           |(false, false) => ((l::c), false)
          end)
    end.
```

```
Definition percolate (r : resol) (a : clause) (l : lit) : resol :=
  fst (percolate0 r a l).

Definition exPerc := percolate exTree (neg P::nil) (neg Q).
```

For example, `exPerc` is the following tree. The $\neg Q$ attached to the left $[\neg P]$ clause percolated to the top, while the process of percolating the one attached to the right $[\neg P]$ clause was stopped at the node labelled with $(P, \neg Q \vee \neg Q \vee \neg R)$.



The main fact about the percolate operation is that it preserves the correctness condition of resolution trees:

```
Lemma percolateCorrect : forall (r : resol) (a : clause) (l : lit),
       correctR r = true -> correctR (percolate r a l) = true.
```

The *graft* operation takes two trees r, s and attaches s to any leaf of r labelled with the clause equal to the root clause of r.

```
Fixpoint graft (r s : resol) { struct r} : resol :=
 match r with
     leaf c => if eqC (clauseR s) c then s else r
    |node l c r1 r2 => node l c (graft r1 s) (graft r2 s)
 end.
```

Grafting preserves the correctness condition as well.

```
Lemma graftCorrect : forall (r s : resol),
  correctR r = true ->
  correctR s = true ->
    correctR (graft r s) = true.
```

2.6 The completeness theorem

We are now ready to present the completeness theorem. It will be proved by measure induction on formulas. The *measure* of a formula $f = [c_1, \ldots, c_k]$ is defined as the number of disjunction symbols in the formula: $measure(f) = \Sigma_{i=1\ldots k} pred(length(c_k))$, where *length* for a given list returns its length and *pred* denotes the total predecessor function on natural numbers: $pred(0) = 0, pred(n+1) = n$.

```
Fixpoint measure (f : form) : nat :=
          match f with
                  nil => 0
                | x :: xs => pred (length x) + (measure xs)
          end.
```

We call a formula *one-literal* if all its clauses have at most one literal. Any formula of measure 0 is one-literal:

```
Lemma l0 : forall f : form, measure f = 0 -> onelL lit f = true.
```

For any formula f, we provide a definition of a predicate stating that either f is satisfiable or refutable:

```
Definition fPred (f : form) : Set :=
   sum (satisfiable f) ({ r : res | refutes r f  = true}).
```

We first tackle the base case of the inductive argument:

```
Lemma l3 : forall f : form, onelL lit f = true  -> fPred f.
```

Proof. By induction on f (as a list). If f is empty, then it is trivially satisfiable with any valuation as a witness. For the inductive step, suppose we have the claim for f. We need to show it for f extended with any clause a. So suppose $[a]@f$ is one-literal. Then obviously so is f, so $fPred(f)$ holds. We have two cases to consider:

– f is satisfiable. Let v be the valuation which sets f to true. We know that a has at most one literal. If a is empty, then the resolution tree consisting of a single leaf labelled with the empty clause refutes f (recall that the value of the empty clause is $false$). Otherwise, it consists of a single literal l. Let $notl$ denote the negation of l. We have 3 subcases to consider:
  - $[notl]$ is one of the clauses of f. Then the resolution tree with two leaves labelled by $[l]$ and $[notl]$ and the node labelled with the pair $(l, [])$ refutes f.
  - $[l]$ is one of the clauses of f. Then $[a]@f$ is satisfiable by v as well, as v must set the value of $[l]$ to $true$.
  - Neither $[notl]$ nor $[l]$ is a clause of f. Then $[a]@f$ is satisfiable by v extended to set a to true.

– There is a resolution r refuting f. Then it is easy to see that r refutes $[a]@f$ as well.

Finally, we prove the main theorem.

Theorem 2. *Any formula is either satisfiable or refutable.*

```
Theorem t : forall f : form, fPred f.
```

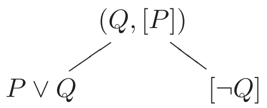
Proof. We proceed by measure induction on f using the function $measure$. The case where $measure(f) = 0$ is handled by Lemmas l0 and l3. Otherwise, $measure(f) > 0$. This implies that there is a clause c in f with at least two literals. Thus we can write c as $[l_1, l_2]@x_0$ for some literals l_1, l_2 and clause x_0. Let Δ denote the formula resulting by removing all occurences of c from f. Let f_1 denote the formula $[l_1]@\Delta$ and let f_2 denote the formula $[l_2, x_0]@\Delta$. It is easy to see that the measures of both f_1 and f_2 are smaller than the measure of f. By the inductive hypothesis, we therefore have several cases to consider.

- f_1 is satisfiable. Then there is a valuation v setting f_1 to $true$. Thus v must set $[l_1]$ and Δ to true, so it also sets $[l_1, l_2]@x_0$ and Δ to true, which implies that the value of f under v is $true$, so f is satisfiable.
- f_2 is satisfiable. Reasoning in the same way, we can derive satisfiability of f.
- There are resolution trees r_1, r_2 refuting f_1 and f_2, respectively. Let r_p denote $percolate(r_2)([l_2]@x_0)(l_1)$. Therefore, r_p results from r_2 by appending l_1 to all leaves labelled by $[l_2]@x_0$ and percolating it up. Since r_2 is a refutation tree, this means that the clause at the root of r_p is either the empty clause or the clause $[l_1]$. In the former case, by Lemma $percolateCorrect$, it is easy to see that r_p refutes f. In the latter, we have two subcases to consider:
  - $[l_1]$ is not among the premises of r_1. In this case, the premises of r_1 are a subset of Δ, so also a subset of f, so r_1 already refutes f.
  - $[l_1]$ is among the premises of r_1. Let r_g denote $graft(r_1)(r_p)$. Therefore, r_g results by replacing every leaf in r_1 labelled with $[l1]$ by r_p. Then r_g refutes f. To show this, we need to show that:
    - r_g corresponds to f. This means that the premises of r_g are a subset of the clauses of f. But note that the premises of r_1 are a subset of $\Delta \cup [l_1]$. By grafting r_p onto r_1, the leaves labelled by $[l_1]$ disappear replaced by r_p. As the premises of r_p are a subset of $\Delta \cup [l_1, l_2]@x_0$, the premises of r_g are a subset of $\Delta \cup [l_1, l_2]@x_0$ and thus also a subset of f.
    - The root of r_g is labelled by the empty clause. This follows trivially by r_1 refuting f_1 and the definition of the grafting function.
    - The tree r_g is a correct resolution tree. This follows easily by Lemmas $graftCorrect$ and $percolateCorrect$.

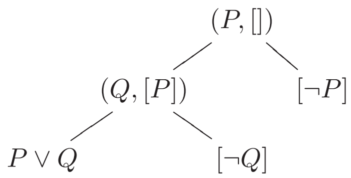
Let us see how the proof works for the formula $f \equiv (P \vee Q) \wedge (\neg P) \wedge (\neg Q)$. Obviously, ϕ is not one-literal. We take $c = P \vee Q$. Then $l_1 = P$, $l_2 = Q$, $x_0 = []$, $\Delta = (\neg P) \wedge (\neg Q)$, $f_1 = P \wedge \neg P \wedge \neg Q$, $f_2 = Q \wedge \neg P \wedge \neg Q$. The application of the inductive hypothesis to f_2 and f_1 results in two refutation tree r_2 and r_1:

$$\begin{array}{ccc} & r_2 & \\ & (Q, []) & \\ / & & \backslash \\ [Q] & & [\neg Q] \end{array} \qquad \begin{array}{ccc} & r_1 & \\ & (P, []) & \\ / & & \backslash \\ [P] & & [\neg P] \end{array}$$

Then the tree r_p is:



Since the clause $[P]$ is obviously not empty, we need to perform the graft operation which results in a tree g, which can easily seen to refute f.



3 Extraction

Now we can reap the benefits of the constructive proof and formalization in a theorem prover as Underwood did for the tableaux method [9]. Coq provides a program extraction capability, which makes it possible to extract a program from our proof. The program, given a representation of a formula f in a CNF form, returns either a valuation satisfying f or a resolution tree refuting f. The extracted program is correct by construction — there is no need for testing for eventual bugs, as the program is *guaranteed* to compute the refuting resolution or the satisfying valuation.

We have chosen to extract a program in a functional programming language Haskell. Coq also offers choices of other programming languages (Scheme, ML). We now show examples of interaction with our program loaded in a Haskell interpreter. The examples correspond to the lemmas and theorems we have proven.

We can check directly the truth of several lemmas in the interpreter. To make the output more readable, we have added a function which renders lists in the style used in this paper. We use standard notation for natural numbers, rendering $S(S(0))$ as simply 2. The `Main>` string in the examples is the standard prompt of the Haskell interpreter.

```
Main> premises exTree
[[Pos 1, Pos 3], [Neg 1], [Pos 2], [Pos 1, Neg 2, Neg 3], [Neg 1]]
Main> refutes exRes phi
True
Main> contraClause exTree exVal
[Pos 1, Neg 2, Neg 3]
Main> lontraClause exTree exVal
[Neg 3, Neg 2, Pos 1]
Main> exPerc
Node (Pos 3) [Neg 2] (Node (Pos 1) [Neg 2, Pos 3] (Leaf [Pos 1, Pos 3])
(Leaf [Neg 2, Neg 1])) (Node (Pos 2) [Neg 3] (Leaf [Pos 2]) (Node (Pos 1)
[Neg 2, Neg 2, Neg 3] (Leaf [Pos 1, Neg 2, Neg 3]) (Leaf [Neg 2, Neg 1])))
Main>
```

Finally, let us see the application of the completeness theorem:

```
Main> f
[[Pos 1, Pos 2], [Neg 1], [Neg 2]]
Main> t f
Inr (Node (Pos 1) [] (Node (Pos 2) [Pos 1] (Leaf [Pos 1, Pos 2]) (Leaf [Neg 2]))
(Leaf [Neg 1]))
Main> let Inr x = t f in refutes x f
True
```

To give the reader a glimpse of the actual Haskell code, we show the Haskell definition of the function t corresponding to Theorem 2.

```
t :: Form -> FPred
t f =
  induction_ltof1 measure
   (\x h ->
    sumbool_rec
     (\_ -> l3 x)
     (\_ ->
      and_rec (\_ _ ->
        case lm1 x of
          Nil -> false_rec __
          Cons l x0 ->
            (case x0 of
               Nil -> false_rec __
               Cons l2 x1 ->
                let delta = removeL eqC (Cons l (Cons l2 x1)) x in
                 sum_rec
                   (\a -> Inl (sat1 l (Cons l (Cons l2 x1)) x a))
                   (\b0 -> sum_rec
                     (\a -> Inl (sat2 (Cons l2 x1) (Cons l (Cons l2 x1)) x a))
                     (\b1 ->
                       Inr (
                         let d1 = proj1_sig b1 in
                         let d11 = percolate d1 (Cons l2 x1) l in
                          sumbool_rec
                           (\_ -> d11)
                           (\_ ->
                             sumbool_rec
                              (\_ -> graft b0 d11)
                              (\_ -> b0)
                              (btf (elemL eqC (Cons l Nil) (premises b0))))
                           (eq_rec (clauseR d1) (p1 d1 (Cons l2 x1) l) Nil)))
                     (h (Cons (Cons l2 x1) delta) __))
                   (h (Cons (Cons l Nil) delta) __))))
     (case measure x of
        O -> Left
        S n -> Right))
   f
```

4 Conclusion

While proofreading this paper, we have discovered several times mistakes in our informal presentation thanks to the included Coq and Haskell code. This is an example of successful *paper verification* using a proof assistant.

Richard Eaton and the second author have also done this proof in Nuprl [6] and are writing a comparison of the systems in their ability to support readable formalized mathematics. This forthcoming article will discuss several subtle points about extraction and how to guide it to produce code as efficient as what can be written directly. At this point, we only mention that one of the most important differences between Coq and Nuprl, namely the treatment of equality, played no role in our developments. As the proofs use only very basic properties of equality, the extensionality of Nuprl and intensionality of Coq did not influence our development.

References

1. Gallier, J.H.: The completeness of propositional resolution: A simple and constructive proof. Logical Methods in Computer Science **2**(5) (2006) 1–7
2. Chang, C.C., Keisler, H.J.: Model Theory. Volume 73 of Studies in Logic and the Foundations of Mathematics. North-Holland, Netherlands (1973)
3. Robinson, J.: A machine oriented logic based on the resolution principle. Journal of the Association of Computing Machinery **12** (1965) 23–41
4. Lewis, H.R., Papadimitriou, C.H.: Elements of the Theory of Computation. Prentice-Hall, Englewood Cliffs, New Jersey (1994)
5. Gallier, J.H.: Logic for Computer Science, Foundations of Automatic Theorem Proving. Harper and Row, NY (1986)
6. Constable, R.L., Allen, S.F., Bromley, H.M., Cleaveland, W.R., Cremer, J.F., Harper, R.W., Howe, D.J., Knoblock, T.B., Mendler, N.P., Panangaden, P., Sasaki, J.T., Smith, S.F.: Implementing Mathematics with the Nuprl Proof Development System. Prentice-Hall, NJ (1986)
7. Constable, R.L., Howe, D.J.: Implementing metamathematics as an approach to automatic theorem proving. In Banerji, R.B., ed.: Formal Techniques in Artificial Intelligence: A Source Book. Elsevier Science Publishers (North-Holland) (1990) 45–76
8. Constable, R., Moczydłowski, W.: Extracting Programs from Constructive HOL Proofs via IZF Set-Theoretic Semantics. In: Proceedings of 3rd International Joint Conference on Automated Reasoning (IJCAR 2006). Volume 4130 of Lecture Notes in Computer Science., Springer (2006) 162–176
9. Underwood, J.L.: The tableau algorithm for intuitionistic propositional calculus as a constructive completeness proof. In: Proceedings of the Workshop on Theorem Proving with Analytic Tableaux, Marseille, France. (1993) 245–248 Available as Technical Report MPI-I-93-213 Max-Planck-Institut für Informatik, Saarbrücken, Germany.