Chapter 1

Resolution Method

In this chapter, we introduce another proof system more suitable for practical applications, called the *resolution method.* This method is the foundation of, among others, *logical programming*, *automated theorem proving*, or *software verification.* In this chapter, we limit ourselves to the resolution method in propositional logic, but later, in Chapter **??**, we will introduce the concept of *unification*, which allows us to search for resolution proofs in predicate logic.

The resolution method works with propositions in *conjunctive normal form (CNF).* Recall that every proposition can be converted to CNF. This conversion is, in the worst case, of exponential time copmlexity (there are even propositions whose shortest CNF equivalent is exponentially longer), but in practice, this is not an issue.

Similar to the tableau method, the resolution method is based on proof by contradiction, i.e., we add the *negation* of the proposition we want to prove to the theory in which we are proving (both converted to CNF), and show that this leads to a contradiction.

To find the contradiction, the resolution method uses a single inference rule called the *resolution rule.* This is a special case of the *cut rule*, which states: *"From the propositions $\varphi \vee \psi$ and $\neg\varphi \vee \chi$, we can infer the proposition $\psi \vee \chi$,"* written as:

$$\frac{\varphi \vee \psi, \neg\varphi \vee \chi}{\psi \vee \chi}$$

In the *resolution rule*, which we will demonstrate shortly, φ is a *literal*, and ψ, χ are *clauses.*

Exercise 1.1. Show that the cut rule is *sound.* (What does it mean?)

1.1 Set Representation

First, we introduce a more compact notation for CNF propositions, called *set notation.* For it would be impractical to represent CNF propositions as strings including brackets and logical symbols.

- Recall that a *literal* ℓ is a propositional variable or its negation, and that $\overline{\ell}$ denotes the *opposite literal* to ℓ.
- A *clause* C is a finite set of literals. The *empty clause*, which is never satisfied,[1] is denoted by $\square$.

[1]It represents the disjunction of an empty set of literals, meaning none of the disjuncts are satisfied.

- A *(CNF) formula* S is a (finite, or even infinite) set of clauses. The *empty formula* $\emptyset$ is always satisfied.[2]

Remark 1.1.1. Note that a *CNF formula* can also be an *infinite* set of clauses. If we convert an infinite propositional theory to CNF, we take (in set representation) all infinitely many clauses as elements of a single formula (set). In practical applications, the formula is (almost always) finite.

In set notation, models correspond to sets of literals that contain exactly one of the literals $p, \neg p$ for each propositional variable p:

- A *(partial) assignment* $\mathcal{V}$ is any set of literals that is *consistent*, i.e., it does not contain both a literal and its negation.
- An assignment is *complete* if it contains either the positive or negative literal for each propositional variable.
- An assignment $\mathcal{V}$ *satisfies* a formula S, written $\mathcal{V} \models S$, if $\mathcal{V}$ contains some literal from each clause in S, i.e.,
$$\mathcal{V} \cap C \neq \emptyset \text{ for each } C \in S.$$

Example 1.1.2. The proposition $\varphi = (\neg p_1 \vee p_2) \wedge (\neg p_1 \vee \neg p_2 \vee p_3) \wedge (\neg p_3 \vee \neg p_4) \wedge (\neg p_4 \vee \neg p_5) \wedge p_4$ is written in set notation as:

$$S = \{\{\neg p_1, p_2\}, \{\neg p_1, \neg p_2, p_3\}, \{\neg p_3, \neg p_4\}, \{\neg p_4, \neg p_5\}, \{p_4\}\}$$

The assignment $\mathcal{V} = \{\neg p_1, \neg p_3, p_4, \neg p_5\}$ satisfies S, written $\mathcal{V} \models S$. It is not complete, but we can extend it with any literal for p_2: both $\mathcal{V} \cup \{p_2\} \models S$ and $\mathcal{V} \cup \{\neg p_2\} \models S$ hold. These two complete assignments correspond to the models $(0, 1, 0, 1, 0)$ and $(0, 0, 0, 1, 0)$.

1.2 Resolution Proof

First, we define one inference step in the resolution proof, the *resolution rule*, which we apply to a pair of clauses; the result is a clause called the *resolvent*, which is a logical consequence of the original pair of clauses:

Definition 1.2.1 (Resolution Rule). Given clauses C_1 and C_2 and a literal ℓ such that $\ell \in C_1$ and $\bar{\ell} \in C_2$, the *resolvent* of the clauses C_1 and C_2 *over the literal* ℓ is the clause

$$C = (C_1 \setminus \{\ell\}) \cup (C_2 \setminus \{\bar{\ell}\}).$$

So, we remove the literal ℓ from the first clause and the literal $\bar{\ell}$ from the second clause (both had to be there!) and take the union of all remaining literals to be the resulting resolvent. Using the symbol $\dot{\sqcup}$ for disjoint union, we could also write:

$$C_1' \cup C_2' \text{ is the resolvent of the clauses } C_1' \dot{\sqcup} \{\ell\} \text{ and } C_2' \dot{\sqcup} \{\bar{\ell}\}$$

Example 1.2.2. From the clauses $C_1 = \{\neg q, r\}$ and $C_2 = \{\neg p, \neg q, \neg r\}$, we can derive the resolvent $\{\neg p, \neg q\}$ over the literal r. From the clauses $\{p, q\}$ and $\{\neg p, \neg q\}$, we can derive $\{p, \neg p\}$ over the literal q or $\{q, \neg q\}$ over the literal p (both of which are tautologies).[3]

[2]It represents the conjunction of an empty set of clauses, meaning all clauses in S are satisfied.

[3]We cannot, however, derive $\square$ 'by resolving over p and q simultaneously' (a common mistake). Note that $\{\{p, q\}, \{\neg p, \neg q\}\}$ is not unsatisfiable, e.g., $(1, 0)$ is a model.

Observation 1.2.3 (Soundness of the Resolution Rule)**.** *The resolution rule is sound, i.e., for any assignment $\mathcal{V}$, the following holds:*

$$\textit{If } \mathcal{V} \models C_1 \textit{ and } \mathcal{V} \models C_2, \textit{ then } \mathcal{V} \models C.$$

A resolution proof is defined similarly to a Hilbert proof as a finite sequence of clauses, ensuring the validity of each clause in the sequence: at each step, we can either write an 'axiom' (a clause from S) or a resolvent of some two previously written clauses.

Definition 1.2.4 (Resolution Proof)**.** A *resolution proof* of a clause C from a CNF formula S is a *finite* sequence of clauses $C_0, C_1, \dots, C_n = C$ such that for each i, either $C_i \in S$ or C_i is the resolvent of some C_j, C_k where $j < i$ and $k < i$.

If a resolution proof exists, we say that C is *resolution provable* from S, written $S \vdash_R C$. A *(resolution) refutation* of the CNF formula S is a resolution proof of $\square$ from S, in which case S is *(resolution) refutable.*

Example 1.2.5. The CNF formula $S = \{\{p, \neg q, r\}, \{p, \neg r\}, \{\neg p, r\}, \{\neg p, \neg r\}, \{q, r\}\}$ is (resolution) refutable, one possible refutation is:

$$\{p, \neg q, r\}, \{q, r\}, \{p, r\}, \{\neg p, r\}, \{r\}, \{p, \neg r\}, \{\neg p, \neg r\}, \{\neg r\}, \square$$

A resolution proof has a natural tree structure: the leaves are labeled by axioms and the inner nodes represent individual resolution steps.

Definition 1.2.6 (Resolution Tree)**.** A resolution tree of a clause C from a CNF formula S is a *finite* binary tree with nodes labeled by clauses, where:

- The root is labeled C,
- The leaves are labeled by clauses from S,
- Each inner node is labeled by a resolvent of its child nodes.

Example 1.2.7. The resolution tree of the empty clause $\square$ from the CNF formula S from Example 1.2.5 is:

$$\begin{tikzcd}
\{p, \neg q, r\} \arrow[dr, dash] & & \{q, r\} \arrow[dl, dash] & & & & & \\
& \{p, r\} \arrow[dr, dash] & & \{\neg p, r\} \arrow[dl, dash] & & \{p, \neg r\} \arrow[dr, dash] & & \{\neg p, \neg r\} \arrow[dl, dash] \\
& & \{r\} \arrow[drr, dash] & & & & \{\neg r\} \arrow[dll, dash] & \\
& & & & \square & & &
\end{tikzcd}$$

It is easy to prove the following observation, by induction on the depth of the tree and the length of the resolution proof:

Observation 1.2.8. *A clause C has a resolution tree from a CNF formula S if and only if $S \vdash_R C$.*

Each resolution proof corresponds to a unique resolution tree. Conversely, from a single resolution tree, we can derive multiple resolution proofs: they are given by any traversal of the tree nodes where an inner node is visited only after both of its children have been visited.

We also introduce another concept, called the *resolution closure*, which contains all clauses that can be 'learned' by resolution from a given formula. It is a useful theoretical perspective on resolution; in applications, constructing the entire resolution closure would be impractical.

Definition 1.2.9 (Resolution Closure)**.** The *resolution closure* $\mathcal{R}(S)$ of the formula S is defined inductively as the smallest set of clauses satisfying:

- $C \in \mathcal{R}(S)$ for all $C \in S$,
- If $C_1, C_2 \in \mathcal{R}(S)$ and C is the resolvent of C_1, C_2, then $C \in \mathcal{R}(S)$.

Example 1.2.10. Let us compute the resolution closure of the formula $S = \{\{p, \neg q, r\}, \{p, \neg r\}, \{\neg p, r\}, \{\neg p, \neg r\}$
The clauses from S are in blue; additional clauses are obtained by resolving them in pairs (first with the first, second with the first, second with the second, etc., over all possible literals):

$$\begin{aligned}\mathcal{R}(S) = \{&\{p, \neg q, r\}, \{p, \neg r\}, \{\neg p, r\}, \{\neg p, \neg r\}, \{q, r\},\\ &\{p, \neg q\}, \{\neg q, r\}, \{r, \neg r\}, \{p, \neg p\}, \{r, \neg s\}, \{p, r\}, \{p, q\}, \{r\}, \{p\}\}\end{aligned}$$

1.3 Soundness and Completeness of the Resolution Method

The resolution method is also both sound and complete.

1.3.1 Soundness of Resolution

Soundness can be easily proved by induction on the length of the resolution proof.

Theorem 1.3.1 (Soundness of Resolution)**.** *If a formula S is resolution refutable, then S is unsatisfiable.*

Proof. Suppose $S \vdash_R \square$ and consider a resolution proof $C_0, C_1, \ldots, C_n = \square$. Suppose, for a contradiction, that S is satisfiable, i.e., $\mathcal{V} \models S$ for some assignment $\mathcal{V}$. By induction on i, we prove that $\mathcal{V} \models C_i$. For $i = 0$, this holds because $C_0 \in S$. For $i > 0$, there are two cases:

- $C_i \in S$, in which case $\mathcal{V} \models C_i$ follows from the assumption that $\mathcal{V} \models S$,
- C_i is the resolvent of C_j, C_k, where $j, k < i$: by the induction hypothesis, $\mathcal{V} \models C_j$ and $\mathcal{V} \models C_k$. $\mathcal{V} \models C_i$ follows from the soundness of the resolution rule.

(Alternatively, we could proceed by induction on the depth of the resolution tree.) $\square$

1.3.2 Substitution Tree

In the completeness proof, we need to construct a resolution tree, whose construction is based on the so-called *substitution tree*. By *substituting* a literal into a formula, we mean simplifying the formula under the assumption that the given literal is true. Substitution was already encountered in Section **??** during *unit propagation*: we remove clauses containing this literal and remove the opposite literal from the remaining clauses.

Definition 1.3.2 (Literal Substitution)**.** Given a formula S and a literal ℓ, the *substitution* of ℓ into S is the formula:

$$S^{\ell} = \{C \setminus \{\bar{\ell}\} \mid \ell \notin C \in S\}$$

Observation 1.3.3. *Here we summarize several simple facts about substitution:*

- *S^{ℓ} is the result of* unit propagation *applied to $S \cup \{\{\ell\}\}$.*
- *S^{ℓ} does not contain the literal ℓ or its opposite $\bar{\ell}$ (it does not contain the propositional variable from ℓ at all).*
- *If S did not contain the literal ℓ or its opposite $\bar{\ell}$, then $S^{\ell} = S$.*
- *If S contained the unit clause $\{\bar{\ell}\}$, then $\square \in S^{\ell}$, meaning S^{ℓ} is unsatisfiable.*

The key property of substitution is expressed by the following lemma:

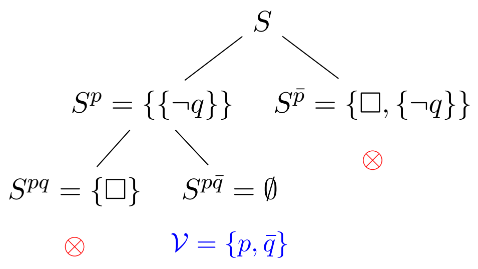
Lemma 1.3.4. *S is satisfiable if and only if S^{ℓ} or $S^{\bar{\ell}}$ is satisfiable.*

Proof. Let $\mathcal{V} \models S$, it cannot contain both ℓ and $\bar{\ell}$ (it must be consistent); without loss of generality, assume $\bar{\ell} \notin \mathcal{V}$, and show that $\mathcal{V} \models S^{\ell}$. Consider any clause in S^{ℓ}. It is of the form $C \setminus \{\bar{\ell}\}$ for a clause $C \in S$ (not containing the literal ℓ). We know that $\mathcal{V} \models C$, but since $\mathcal{V}$ does not contain $\bar{\ell}$, the assignment $\mathcal{V}$ must satisfy some other literal of C, so $\mathcal{V} \models C \setminus \{\bar{\ell}\}$.

Conversely, suppose there exists an assignment $\mathcal{V}$ satisfying S^{ℓ} (again, without loss of generality). Since $\bar{\ell}$ (or ℓ) does not appear in S^{ℓ}, it holds that $\mathcal{V} \setminus \{\bar{\ell}\} \models S^{\ell}$. The assignment $\mathcal{V}' = (\mathcal{V} \setminus \{\bar{\ell}\}) \cup \{\ell\}$ then satisfies each clause $C \in S$: if $\ell \in C$, then $\ell \in C \cap \mathcal{V}'$ and $C \cap \mathcal{V}' \neq \emptyset$, otherwise $C \cap \mathcal{V}' = (C \setminus \{\bar{\ell}\}) \cap \mathcal{V}' \neq \emptyset$ because $\mathcal{V} \setminus \{\bar{\ell}\} \models C \setminus \{\bar{\ell}\} \in S^{\ell}$. We have verified that $\mathcal{V}' \models S$, so S is satisfiable. $\square$

Whether a given *finite* formula is satisfiable can thus be determined recursively (using the *divide and conquer* method) by substituting both possible literals for (some, say the first) propositional variable appearing in the formula and branching the computation. Essentially, this is a similar principle to the DPLL algorithm (see Section **??**). The resulting tree is called a *substitution tree.*

Example 1.3.5. Let's illustrate this concept with an example by constructing a substitution tree for the formula $S = \{\{p\}, \{\neg q\}, \{\neg p, \neg q\}\}$:


S
$S^{p} = \{\{\neg q\}\}$
$S^{\bar{p}} = \{\square, \{\neg q\}\}$
$\otimes$
$S^{pq} = \{\square\}$
$S^{p\bar{q}} = \emptyset$
$\otimes$
$\mathcal{V} = \{p, \bar{q}\}$

Once a branch contains the empty clause $\square$, it is unsatisfiable, and we need not continue in that branch. In the leaves, there are either unsatisfiable theories or empty theories: in the latter case, the sequence of substitutions gives us a satisfying assignment.

From the construction, it is evident how to proceed in the case of a finite formula. However, the substitution tree makes sense, and the following corollary holds, even for infinite formulas:

Corollary 1.3.6. *A formula S (over a countable language) is unsatisfiable if and only if every branch of the substitution tree contains the empty clause $\square$.*

Proof. For a finite formula S, this follows from the above discussion; it can be easily proved by induction on the size of $\mathrm{Var}(S)$:

- If $|\mathrm{Var}(S)| = 0$, we have $S = \emptyset$ or $S = \{\square\}$, in both cases, the substitution tree is one node, and the statement holds.
- In the inductive step, we choose any literal $\ell \in \mathrm{Var}(S)$ and apply Lemma 1.3.4.

If S is infinite and satisfiable, it has a satisfying assignment, which 'matches' the corresponding (infinite) branch in the substitution tree. If S is infinite and unsatisfiable, then by the Compactness Theorem, there is a finite part $S' \subseteq S$ that is also unsatisfiable. Substitution for all variables from $\mathrm{Var}(S')$ will lead to $\square$ in each branch, which will happen after finitely many steps. $\square$

1.3.3 Completeness of Resolution

Theorem 1.3.7 (Completeness of Resolution)**.** *If S is unsatisfiable, it is resolution refutable (i.e., $S \vdash_R \square$).*

Proof. If S is infinite, it has a finite unsatisfiable part S' by the Compactness Theorem. A resolution refutation of S' is also a resolution refutation of S. Suppose S is finite.

The proof is by induction on the number of variables in S. If $|\mathrm{Var}(S)| = 0$, the only possible unsatisfiable formula without variables is $S = \{\square\}$, and we have a one-step proof $S \vdash_R \square$. Otherwise, choose $p \in \mathrm{Var}(S)$. By Lemma 1.3.4, both S^p and $S^{\bar{p}}$ are unsatisfiable. They have one fewer variable, so by the induction hypothesis, there are resolution trees T for $S^p \vdash_R \square$ and T' for $S^{\bar{p}} \vdash_R \square$.

We show how to construct the resolution tree $\widehat{T}$ for $S \vdash_R \neg p$. Similarly, we construct $\widehat{T'}$ for $S \vdash_R p$, and then easily construct the resolution tree for $S \vdash_R \square$: we attach the roots of the trees $\widehat{T}$ and $\widehat{T'}$ as the left and right children of the root $\square$ (i.e., in the last step of the resolution proof, we obtain $\square$ by resolving $\{\neg p\}$ and $\{p\}$).

It remains to show the construction of the tree $\widehat{T}$: the set of nodes and the order remain the same; we only change some clauses in the nodes by adding the literal $\neg p$. At each leaf of the tree T is some clause $C \in S^p$, and either $C \in S$ or it is not, but $C \cup \{\neg p\} \in S$. In the first case, we leave the label unchanged. In the second case, we add the literal $\neg p$ to C *and to all clauses above this leaf.* In the leaves, there are now only clauses from S, in the root, we have changed $\square$ to $\neg p$. And each inner node remains the resolvent of its children. $\square$

Exercise 1.2. The proof of the Completeness Theorem for resolution gives a method for recursively 'growing' a resolution refutation. Think about how to do this and apply it to an example of an unsatisfiable formula.

1.4 LI-Resolution and Horn-SAT

We begin with a different view of the resolution proof, called the *linear proof.*

1.4.1 Linear Proof

In addition to the resolution tree, a resolution proof can also be organized as a *linear proof*, where in each step we have one *central* clause, which we resolve with a *side* clause, which is either one of the previous central clauses or an axiom from S. The resolvent then becomes the new central clause.[4]

Definition 1.4.1 (Linear Proof)**.** A *linear proof* (by resolution) of a clause C from a formula S is a finite sequence

$$\begin{bmatrix} C_0 \\ B_0 \end{bmatrix}, \begin{bmatrix} C_1 \\ B_1 \end{bmatrix}, \dots, \begin{bmatrix} C_n \\ B_n \end{bmatrix}, C_{n+1}$$

where C_i are called *central* clauses, C_0 is the *initial* clause, $C_{n+1} = C$ is the *final* clause, B_i are *side* clauses, and it holds that:

- $C_0 \in S$, for $i \leq n$, C_{i+1} is the resolvent of C_i and B_i,
- $B_0 \in S$, for $i \leq n$, $B_i \in S$ or $B_i = C_j$ for some $j < i$.

A *linear refutation* of S is a linear proof of $\square$ from S. A linear proof can be illustrated as follows:

$$\begin{tikzcd}
C_0 \arrow[r, dash] & C_1 \arrow[r, dash] & C_2 \arrow[r, dash] & \cdots\cdots\cdots \arrow[r, dash] & C_n \arrow[r, dash] & C_{n+1} \\
B_0 \arrow[ru, dash] & B_1 \arrow[ru, dash] & & B_{n-1} \arrow[ru, dash] & B_n \arrow[ru, dash] &
\end{tikzcd}$$

Example 1.4.2. Let's construct a linear refutation of the formula $S = \{\{p, q\}, \{p, \neg q\}, \{\neg p, q\}, \{\neg p, \neg q\}\}$ (i.e., a linear proof of $\square$ from S). A linear proof might look like this:

$$\begin{tikzcd}
\{p, q\} \arrow[r, dash] & {\color{blue}\{p\}} \arrow[r, dash] & \{q\} \arrow[r, dash] & \{\neg p\} \arrow[r, dash] & \square \\
\{p, \neg q\} \arrow[ru, dash] & \{\neg p, q\} \arrow[ru, dash] & \{\neg p, \neg q\} \arrow[ru, dash] & {\color{red}\{p\}} \arrow[ru, dash] &
\end{tikzcd}$$

The last side clause ${\color{red}\{p\}}$ (in red) is not from S, but is equal to the previous central clause (in blue).

Exercise 1.3. Convert the linear proof from Example 1.4.2 into a resolution tree.

Remark 1.4.3. C has a linear proof from S if and only if $S \vdash_R C$.

A resolution tree can easily be produced from a linear proof by induction on the length of the proof: the base case is obvious, and if there is a side clause B_i that is not an axiom from S, then $B_i = C_j$ for some $j < i$ and we only need to attach the resolution tree for proving C_j from S instead of B_i. Notice that this also implies the *soundness* of linear resolution.

We will not present the proof of the reverse implication. It follows from the *completeness* of linear resolution, whose proof can be found in the textbook *A. Nerode, R. Shore: Logic for Applications* [1].

[4]While the construction of the resolution tree can be easily described recursively, the linear proof better corresponds to a procedural computation. It is only about finding a suitable side clause.

1.4.2 LI-Resolution

In a general linear proof, each subsequent side clause can either be an axiom from S or one of the previous central clauses. If we forbid the latter option and require that all side clauses must be from S, we get the so-called *LI (linear-input) resolution*:

Definition 1.4.4 (LI-Proof)**.** An *LI-proof* (by resolution) of a clause C from a formula S is a linear proof

$$\begin{bmatrix} C_0 \\ B_0 \end{bmatrix}, \begin{bmatrix} C_1 \\ B_1 \end{bmatrix}, \ldots, \begin{bmatrix} C_n \\ B_n \end{bmatrix}, C$$

in which each side clause B_i is an axiom from S. If an LI-proof exists, we say that C is *LI-provable* from S, and we write $S \vdash_{LI} C$. If $S \vdash_{LI} \square$, then S is *LI-refutable.*

Remark 1.4.5. An LI-proof directly gives a resolution tree (all leaves are axioms) in a special form that we might call a 'hairy path'. Conversely, from a resolution tree in the form of a hairy path, we immediately obtain an LI-proof: the vertices on the path are central clauses, the hairs are side clauses.

While *linear resolution*[5] is just another view of the general resolution proof, *LI-resolution* introduces a significant restriction: we lose *completeness* (not every unsatisfiable formula has an LI-refutation). On the other hand, LI-proofs are simpler to construct.[6]

1.4.3 Completeness of LI-Resolution for Horn Formulas

As we will now show, LI-resolution is *complete for Horn formulas.* As we will see in the next section, it is the basis of Prolog interpreters, which work with Horn formulas. First, let us recall the terminology related to hornness and also programs, in set representation:

- A *Horn clause* is a clause containing at most one positive literal.
- A *Horn formula* is (finite or even infinite) a set of Horn clauses.
- A *fact* is a positive unit (Horn) clause, i.e., $\{p\}$, where p' is a propositional variable.
- A *rule* is a (Horn) clause with exactly one positive and at least one negative literal.
- Rules and facts are called *program clauses.*
- A *goal* is a non-empty (Horn) clause without a positive literal.[7]

We will find the following simple observation useful:

Observation 1.4.6. *If a Horn formula S is unsatisfiable and $\square \notin S$, then it contains a fact and a goal.*

Proof. If it does not contain a fact, we can evaluate all variables to 0; if it does not contain a goal, we evaluate to 1. $\square$

[5]That is, a proof system based on finding *linear* proofs or refutations.

[6]In each step, we only have to choose from clauses in S, not from previously proven central clauses.

[7]Recall that we prove by *contradiction*, so the *goal* is the negation of what we want to prove.

Now we will state and prove the Completeness Theorem for LI-Resolution for Horn formulas. The proof also provides a guide on how to construct an LI-refutation, based on the process of unit propagation. This procedure is illustrated in the example below, which you can follow along with the reading of the proof.

Theorem 1.4.7 (Completeness of LI-Resolution for Horn Formulas)**.** *If a Horn formula T is satisfiable, and $T \cup \{G\}$ is unsatisfiable for a goal G, then $T \cup \{G\} \vdash_{LI} \square$, by an LI-refutation that starts with the goal G.*

Proof. Similarly to the Completeness Theorem for Resolution, we can assume by the Compactness Theorem that T is finite. The proof (construction of the LI-refutation) will be done by induction on the number of variables in T.

By Observation 1.4.6, T contains a fact $\{p\}$ for some propositional variable p. Since $T \cup \{G\}$ is unsatisfiable, according to Lemma 1.3.4, $(T \cup \{G\})^p = T^p \cup \{G^p\}$ is also unsatisfiable, where $G^p = G \setminus \{\neg p\}$.

If $G^p = \square$, then $G = \{\neg p\}$, $\square$ is the resolvent of G and $\{p\} \in T$, and we have a one-step LI-refutation of $T \cup \{G\}$ (this is the base case of the induction).

Otherwise, we use the induction hypothesis. Note that the formula T^p is satisfiable (by the same assignment as T, because it must contain p due to the fact $\{p\}$, so it does not contain $\neg p$) and has fewer variables than T. Thus, by the induction hypothesis, there exists an LI-derivation of $\square$ from $T^p \cup \{G^p\}$ starting with $G^p = G \setminus \{\neg p\}$.

We construct the required LI-refutation of $T \cup \{G\}$ starting with G (similarly to the proof of the Completeness Theorem for Resolution) by adding the literal $\neg p$ to all leaves that are not already in $T \cup \{G\}$ (i.e., they were created by removing $\neg p$), and to all vertices above them. This gives us $T \cup \{G\} \vdash_{LI} \neg p$, and finally we add the side clause $\{p\}$ and derive $\square$. $\square$

Example 1.4.8. Consider a (satisfiable, Horn) theory T, written in set representation as the formula $T = \{\{p, \neg r, \neg s\}, \{\neg q, r\}, \{q, \neg s\}, \{s\}\}$. Suppose we want to prove that $p \wedge q$ holds in the theory T.[8] In the resolution method, we consider the goal $G = \{\neg p, \neg q\}$ and show that $T \cup \{G\} \vdash_{LI} \square$.

Following the guide from the proof, we find a fact in the formula T, and perform unit propagation in both T and the goal G. We repeat the process until the formula is empty:

- $T = \{\{p, \neg r, \neg s\}, \{\neg q, r\}, \{q, \neg s\}, \{s\}\}$, $G = \{\neg p, \neg q\}$
- $T^s = \{\{p, \neg r\}, \{\neg q, r\}, \{q\}\}$, $G^s = \{\neg p, \neg q\}$
- $T^{sq} = \{\{p, \neg r\}, \{r\}\}$, $G^{sq} = \{\neg p\}$
- $T^{sqr} = \{\{p\}\}$, $G^{sqr} = \{\neg p\}$
- $T^{sqrp} = \emptyset$, $G^{sqrp} = \square$

Now, we construct the resolution refutation in reverse:

- $T^{sqrp}, G^{sqrp} \vdash_{LI} \square$:

$$\square$$

[8] In Prolog, we would pose the 'query': `?-p,q`.

- $T^{sqr}, G^{sqr} \vdash_{LI} \square$:

$$\begin{tikzcd}
\{\neg p\} \arrow[r, dash] & \square \\
\{p\} \arrow[ur, dash] &
\end{tikzcd}$$

- $T^{sq}, G^{sq} \vdash_{LI} \square$:

$$\begin{tikzcd}
\{\neg p\} \arrow[r, dash] & \{\neg r\} \arrow[r, dash] & \square \\
\{p, \neg r\} \arrow[ur, dash] & \{r\} \arrow[ur, dash] &
\end{tikzcd}$$

- $T^{s}, G^{s} \vdash_{LI} \square$:

$$\begin{tikzcd}
\{\neg p, \neg q\} \arrow[r, dash] & \{\neg q, \neg r\} \arrow[r, dash] & \{\neg q\} \arrow[r, dash] & \square \\
\{p, \neg r\} \arrow[ur, dash] & \{\neg q, r\} \arrow[ur, dash] & \{q\} \arrow[ur, dash] &
\end{tikzcd}$$

- $T, G \vdash_{LI} \square$

$$\begin{tikzcd}
\{\neg p, \neg q\} \arrow[r, dash] & \{\neg q, \neg r, \neg s\} \arrow[r, dash] & \{\neg q, \neg s\} \arrow[r, dash] & \{\neg s\} \arrow[r, dash] & \square \\
\{p, \neg r, \neg s\} \arrow[ur, dash] & \{\neg q, r\} \arrow[ur, dash] & \{q, \neg s\} \arrow[ur, dash] & \{s\} \arrow[ur, dash] &
\end{tikzcd}$$

1.4.4 Prolog Program

Although the real power of Prolog comes from *unification* and resolution in predicate logic, we will show how Prolog uses the resolution method with a *propositional* program. Adaptation to predicates will be straightforward later.

A *program* in Prolog is a Horn formula containing only *program clauses*, i.e., *facts* or *rules.* A *query* is a conjunction of facts, the negation of the query is the *goal.*

Example 1.4.9. As an example of a Prolog program, we will use the theory (formula) T and the query $p \wedge q$ from Example 1.4.8. For instance, the clause $\{p, \neg r, \neg s\}$, which is equivalent to $r \wedge s \rightarrow p$, is written in Prolog as: `p:-r,s.`

```
p:-r,s.
r:-q.
q:-s.
s.
```

And we pose the query to the program:

```
?-p,q.
```

Corollary 1.4.10. *Let P be a program and $Q = p_1 \wedge \cdots \wedge p_n$, and denote $G = \{\neg p_1, \ldots, \neg p_n\}$ (i.e., $G \sim \neg Q$). The following conditions are equivalent:*

- $P \models Q$,
- $P \cup \{G\}$ *is unsatisfiable,*
- $P \cup \{G\} \vdash_{LI} \square$*, and there is an LI-refutation starting with the goal* G*.*

Proof. The equivalence of the first two conditions is by contradiction, and the equivalence of the second and third is by the Completeness Theorem for LI-Resolution for Horn formulas. (Note that the Program is always satisfiable.) $\square$

Bibliography

[1] Anil Nerode and Richard A. Shore. *Logic for Applications.* Springer Science & Business Media, December 2012. Google-Books-ID: 90HhBwAAQBAJ.