St. Francis Xavier University
Department of Computer Science

CSCI 544: Computational Logic
Lecture 9: Predicate Logic IV—Resolution
Fall 2025

1 Functions, Terms, and Normal Forms

As we have seen throughout our study of predicate logic, we're able to express many more complex statements and properties using predicates than we could using propositions. For example, we can express the property of the "greater than" relation being transitive using the formula

$$\forall x\ \forall y\ \forall z\ (G(x,y) \wedge G(y,z) \Rightarrow G(x,z))$$

and the interpretation $\mathscr{I} = (\mathbb{Z}, \{>\}, \{\})$. Here, we are asserting that if $x > y$ and $y > z$, where $x, y, z \in \mathbb{Z}$, then it is the case that $x > z$.

In a past lecture, we made an offhand comment that terms appearing in a formula could be constants, variables, or *functions*. However, thus far, we haven't really spent time on learning how to handle functions as terms. In this section, we will direct our focus to that topic, and in doing so we'll be able to express even more properties; for example,

$$\forall x\ \forall y\ \forall z\ (x > y) \Rightarrow (x + z > y + z),$$

which is another property of the "greater than" relation that uses the addition ($+$) function.

Let's begin by formally defining the notion of a *term*.

Definition 1 (Term)**.** Let F be a set of function symbols, where each symbol has an associated arity (denoted by a superscript). A term is defined recursively as follows:

- Any constant, variable, or 0-ary function symbol $f^0 \in F$ is a term; and
- If $f^n \in F$ is an n-ary function symbol where $n > 0$, and $\{t_1, t_2, \ldots, t_n\}$ are terms, then $f^n(t_1, t_2, \ldots, t_n)$ is a term.

We can then generalize our earlier definition of an atomic formula to be any n-ary predicate P followed by a list of n term arguments t_i; that is, $P(t_1, t_2, \ldots, t_n)$.

Likewise, we generalize our earlier definition of an interpretation to take into account functions as terms. Let U be a set of formulas where $\{P_1, \ldots, P_k\}$ are predicate symbols, $\{f_1^{n_1}, \ldots, f_\ell^{n_\ell}\}$ are function symbols, and $\{a_1, \ldots, a_m\}$ are constants appearing in formulas of U. An interpretation is then a tuple

$$\left(D, \{R_1, \ldots, R_k\}, \{F_1^{n_1}, \ldots, F_\ell^{n_\ell}\}, \{d_1, \ldots, d_m\}\right),$$

where D is a nonempty domain, R_i is an n-ary relation on D assigned to the n-ary predicate P_i, $F_j^{n_j}$ is an n_j-ary function on D assigned to the n_j-ary function symbol $f_j^{n_j}$, and d_k is an element of D assigned to the constant a_k.

Example 2. Consider the formula $A = \forall x\ \forall y\ (P(x,y) \Rightarrow P(f(x,a), f(y,a)))$. This formula is true under the interpretation $\mathscr{I} = (\mathbb{Z}, \{\leq\}, \{+\}, \{1\})$. To see why this is the case, consider the result of assigning arbitrary values m and n to x and y, respectively:

$$f(x,a) \to +(x,a) \to +(m,1) \to m+1; \text{ and}$$
$$f(y,a) \to +(y,a) \to +(n,1) \to n+1.$$

If P is assigned the relation $\leq$, then the formula A expresses the fact that $m \leq n$ implies $(m+1) \leq (n+1)$, which is correct.

1.1 Prenex Normal Form

When we discussed resolution for the first time, we defined a special form of propositional logic formulas known as conjunctive normal form. A formula in conjunctive normal form is one consisting of some number of subformulas, themselves consisting of literals joined by disjunctive connectives, that are all joined by conjunctive connectives.

Now that we're using predicate logic with its added quantifier symbols, it would be nice to have a similar normal form for our predicate logic formulas. Unfortunately, we can't directly apply the notion of conjunctive normal form to our predicate logic formulas, since that normal form doesn't specify how we need to handle quantifier symbols.

Since quantifiers can appear anywhere in a predicate logic formula, it would make sense for a "predicate-focused" normal form to normalize the positions of every quantifier; say, by moving all quantifiers to the front of the formula. If we were to do that, then we would end up with a formula in *prenex normal form*.[1]

Definition 3 (Prenex normal form)**.** A formula is in prenex normal form, or PNF, if it is of the form

$$Q_1 x_1 \; Q_2 x_2 \; \ldots \; Q_n x_n \; B,$$

where each Q_i is either a universal ($\forall$) or existential ($\exists$) quantifier, each x_i is the variable being quantified by Q_i, and B is a quantifier-free formula.

In prenex normal form, we sometimes refer to $Q_1 x_1 \; Q_2 x_2 \; \ldots \; Q_n x_n$ as the *prefix* and to B as the *matrix* (although B is obviously not a matrix in the linear algebra sense). Naturally, every quantifier-free formula is already in prenex normal form.

Example 4. Each of the following formulas is in prenex normal form:

$$\forall x \; \forall y \; \neg(P(x) \Rightarrow Q(y)); \quad \forall x \; \exists y \; R(x, y); \quad S(x, y).$$

On the other hand, the following formulas are not in prenex normal form:

$$\forall x \; P(x) \vee \forall x \; Q(x); \quad \neg \forall x \; R(x, y).$$

Just as we could do with formulas in conjunctive normal form, we can write PNF formulas in their clausal form if we wish. To do so, we simply take the matrix of the PNF formula and express that matrix as a set of clauses.

As you might also expect, we can develop a procedure for converting arbitrary logical formulas into their prenex normal form equivalents. Much like our procedure for conversion to conjunctive normal form, we need only follow four steps.

Theorem 5. *Every formula can be converted to an equivalent formula in prenex normal form.*

Proof. We prove this theorem by constructing a method to transform formulas into their prenex normal form. This method makes use of the logical equivalences we established in an earlier lecture.

Given an arbitrary formula, apply the following steps:

1. Eliminate all occurrences of the implication ($\Rightarrow$) and biconditional ($\Leftrightarrow$) connectives from the formula. For this, we use the following logical equivalences:

$$A \Rightarrow B \equiv \neg A \vee B$$
$$A \Leftrightarrow B \equiv (\neg A \vee B) \wedge (A \vee \neg B)$$
$$A \Leftrightarrow B \equiv (A \wedge B) \vee (\neg A \wedge \neg B)$$

[1] The word "prenex" might sound modern or futuristic, but in fact, it comes from the Latin word *praenexus*, meaning "tied up/bound up in front". Naturally, this refers to the quantifiers in the formula, which are all moved to the front in prenex normal form.

2. Move all negations inward such that they only appear as part of literals. For this, we use De Morgan's laws together with the following logical equivalences:

$$\begin{aligned} \neg\neg A &\equiv A \\ \neg\exists x\ A(x) &\equiv \forall x\ \neg A(x) \\ \neg\forall x\ A(x) &\equiv \exists x\ \neg A(x) \end{aligned}$$

3. Standardize the variables apart, when necessary. In essence, this step ensures that the individual clauses in a formula containing bound variables that are repeated across clauses will be changed to have distinct variables without changing the semantics of the formula.

   For example, consider the formula

$$\forall x\ (P(x) \Rightarrow Q(x)) \wedge \exists x\ Q(x) \wedge \exists z\ P(z) \wedge \exists z\ (Q(z) \Rightarrow R(x)).$$

   This formula contains four clauses, but these clauses share the same bound variables x and z. We can standardize the variables apart by assigning new variables to some (but not necessarily all) of these bound variables. In doing so we obtain the resultant formula

$$\forall u\ (P(u) \Rightarrow Q(u)) \wedge \exists v\ Q(v) \wedge \exists w\ P(w) \wedge \exists z\ (Q(z) \Rightarrow R(x)).$$

4. Move all quantifiers to the front of the formula. For this, we use the identities involving quantifiers that we established in an earlier lecture together with the following logical equivalences, where in each case the variable x does not occur in A:

$$\begin{aligned} A \wedge \exists x\ B(x) &\equiv \exists x\ (A \wedge B(x)) \\ A \wedge \forall x\ B(x) &\equiv \forall x\ (A \wedge B(x)) \\ A \vee \exists x\ B(x) &\equiv \exists x\ (A \vee B(x)) \\ A \vee \forall x\ B(x) &\equiv \forall x\ (A \vee B(x)) \end{aligned}$$

   Each of these logical equivalences shows that if the truth value of A does not depend on the value of x, then we can quantify A over x while not affecting its truth value.

At this point, our formula is now in prenex normal form. □

Example 6. Consider the formula $A = \forall x\ (\exists y\ R(x,y) \wedge \forall y\ \neg S(x,y) \Rightarrow \neg(\exists y\ R(x,y) \wedge P))$. We will convert this formula to an equivalent formula in prenex normal form.

We begin by removing the lone occurrence of the implication operator, which produces the formula

$$\forall x\ (\neg(\exists y\ R(x,y) \wedge \forall y\ \neg S(x,y)) \vee \neg(\exists y\ R(x,y) \wedge P)).$$

We then move all negations inward until they apply to literals, which produces the formula

$$\forall x\ (\forall y\ \neg R(x,y) \vee \exists y\ S(x,y) \vee \forall y\ \neg R(x,y) \vee \neg P).$$

Now, observe that each of the three quantified clauses within the outermost parentheses use the same bound variable y. We must therefore standardize the variables apart, which produces the formula

$$\forall x\ (\forall u\ \neg R(x,u) \vee \exists v\ S(x,v) \vee \forall w\ \neg R(x,w) \vee \neg P).$$

Finally, we move all quantifiers to the front of the formula, which gives us the resultant prenex normal form formula

$$\forall x\ \forall u\ \exists v\ \forall w\ (\neg R(x,u) \vee S(x,v) \vee \neg R(x,w) \vee \neg P).$$

1.2 Skolem Normal Form

Going one step further, once we have a formula in prenex normal form, we can optionally remove all of the existential quantifiers through a process known as *Skolemization*. This procedure, named for the Norwegian mathematician Thoralf Skolem, produces a formula that is in $\exists$-free prenex normal form (alternatively called *Skolem normal form*). Although converting a formula to one without existential quantifiers may not seem immediately useful, it is often the first step employed by an automated theorem prover.

Suppose that we're given a formula of the form

$$\forall x_1\ \forall x_2\ \cdots \forall x_n\ \exists y\ A,$$

where A is some subformula possibly containing quantifiers itself. The expression $\exists y\ A$ appears in the scope of each of the previously quantified variables x_1 through x_n, and we can think of each instance of y in terms of a function of these variables; that is, in terms of $f(x_1, \ldots, x_n)$. Such a function is referred to as a *Skolem function* or, when there are no arguments, a *Skolem constant*.

In essence, the process of Skolemization replaces each existentially quantified variable with its corresponding Skolem function or constant; thus, the above formula would become $\forall x_1\ \forall x_2\ \cdots \forall x_n\ A'$, where $A' = A[y/f(x_1, \ldots, x_n)]$.

Unfortunately, the formula obtained through Skolemization is not always logically equivalent to the original formula, since it's possible that more than one instance of the existentially quantified variable could satisfy the formula. When we Skolemize the formula, on the other hand, we instantiate one instance of the existentially quantified variable in terms of the universally quantified variables. For our purposes, though, this doesn't matter; it is still possible to establish the weaker property of satisfiability in general.

Proposition 7. *Let A be a formula containing existential quantifiers. Then there exists a formula A' in Skolem normal form that is satisfiable if and only if A is satisfiable.*

Much like converting to prenex normal form, the process of Skolemization can be distilled into a few steps.

Theorem 8. *Every formula in prenex normal form can be converted to an equivalent formula in Skolem normal form.*

Proof. We prove this theorem by constructing a method to transform formulas into their Skolem normal form. If the given formula is not in prenex normal form to begin with, follow the procedure in the proof of Theorem 5 to put it into that form. Then, apply the following steps:

1. Set $i = 1$.
2. Suppose $A_i = \forall x_1\ \forall x_2\ \cdots \forall x_n\ \exists y\ A$, where A is some subformula possibly containing quantifiers itself. Repeat until all existential quantifiers are removed:
   (a) If $n = 0$, then A_i is of the form $\exists y\ A$. Take $A_{i+1} = A'$, where A' is obtained from A by replacing all occurrences of y with some Skolem constant c.
   (b) If $n > 0$, then take $A_{i+1} = \forall x_1\ \forall x_2\ \cdots \forall x_n\ A'$, where A' is obtained from A by replacing all occurrences of y with the Skolem function $f(x_1, \ldots, x_n)$.
   (c) Increment i by 1.

At this point, our formula is now in Skolem normal form. □

Example 9. Consider the formula $A = \exists x\ \forall y\ \forall z\ \exists t\ P(x, y, z, t)$. We will convert this formula to its Skolem normal form.

Observe that the leftmost existential quantifier, $\exists x$, is not preceded by any universal quantifier. Thus, we can replace x by a Skolem constant c. This gives us the formula $A_2 = \forall y\ \forall z\ \exists t\ P(c, y, z, t)$.

Now, there are two universal quantifiers preceding the next existential quantifier, $\exists t$. Thus, the Skolem function corresponding to t must have two arguments, y and z. Substituting t with $f(y, z)$, we obtain $A_3 = \forall y\ \forall z\ P(c, y, z, f(y, z))$, which is in Skolem normal form.

2 Resolution

When we first introduced the method of resolution, we saw that it served as a decision procedure to test the unsatisfiability of a formula in propositional logic, and we proved that the method was both sound and complete. Now that we're studying resolution in the context of predicate logic, we're still able to prove the soundness and completeness of the method as we did before, but unfortunately we lose the nice property of resolution being a decision procedure. This property no longer holds because, much like with the method of semantic tableaux, we have no guarantee that the method of resolution will halt when we provide a predicate logic formula as input.

2.1 Ground Resolution

For our first foray into predicate logic resolution, we will restrict ourselves to considering only *ground clauses*, which are terms that do not contain any variables. Restricting ourselves in this way allows us to reuse and extend many of the results we established in the propositional logic case, which will come in handy as we build up to the general method of resolution for predicate logic.

As before, the method of ground resolution relies on one rule, stated as follows.

Ground resolution rule. Suppose C_1 and C_2 are ground clauses where $\ell \in C_1$ and $\ell^{\mathrm{C}} \in C_2$ for some literal ℓ. In this case, C_1 and C_2 are said to clash on the complementary pair of literals ℓ and ℓ^{C}. From this, we can determine the resolvent of C_1 and C_2,

$$\mathrm{Res}(C_1, C_2) = (C_1 \setminus \{\ell\}) \cup (C_2 \setminus \{\ell^{\mathrm{C}}\}).$$

Comparing the ground resolution rule to our original resolution rule in the propositional logic case, we see that the two are essentially identical. What is also identical between the two methods is the following result:

Theorem 10. *Clauses C_1 and C_2 are satisfiable if and only if the resolvent $C = \mathrm{Res}(C_1, C_2)$ is satisfiable.*

Proof. Omitted. □

Lastly, the algorithm to perform ground resolution on a predicate logic formula is basically the same as our original algorithm for propositional logic formulas. Given a set of ground clauses, we repeatedly apply the ground resolution rule, and we will eventually arrive at an empty clause if and only if the set of ground clauses (and, hence, the original formula) is unsatisfiable.

So, if everything is the same between the propositional logic resolution method and the predicate logic ground resolution method, why don't we take the easy option and just use the same method for both logical systems? As it turns out, the method of ground resolution isn't so useful for predicate logic formulas, since the set of ground clauses is infinite. This is a consequence of us drawing our constants from a countably infinite set, as we remarked in our discussion on the method of semantic tableaux.

Now, where do we go from here? Naturally, we must move beyond considering only ground clauses and reintroduce variables to our formulas. While we can't exactly find clashing pairs of clauses by looking at variables alone, we can employ substitution of terms into variables to produce ground clauses and, from there, determine any clashes that arise.

2.2 Substitution and Unification

Before we see how to use substitution in our resolution procedure, let us extend the definition of substitution itself that we first introduced in our discussion on the semantics of predicate logic. In lieu of our previous

definition, which only allows for one substitution of a term for a variable, here we will define substitution as a set of mappings (akin to multiple substitutions at once).

Definition 11 (Substitution)**.** A substitution of terms for variables is a set $\{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$, where each x_i is a distinct variable and each t_i is a term not equal to x_i.

We can naturally define the *empty substitution* to be the empty set.

If we have some *expression*—that is, a term, a literal, a clause, or a set of clauses—then we can apply a substitution to that expression. If we denote our substitution by θ and our expression by E, then we can obtain an *instance* $E\theta$ of E by simultaneously replacing each occurrence of the variables x_i in E with the corresponding terms t_i.

Example 12. Consider the expression $E = P(x, y) \vee \neg Q(f(x))$ together with the substitution

$$\theta = \{x \leftarrow a, y \leftarrow f(x)\}.$$

Applying this substitution to our expression, we obtain

$$E\theta = P(a, f(x)) \vee \neg Q(f(a)).$$

Note that we don't replace the occurrence of x in $P(a, f(x))$ with a, since we must apply all substitutions simultaneously.

Just like we can compose functions in mathematics, so too can we compose substitutions. If we have two substitutions

$$\theta = \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\} \text{ and}$$
$$\sigma = \{y_1 \leftarrow s_1, \ldots, y_n \leftarrow s_n\},$$

then the composition of these substitutions is given by

$$\theta\sigma = \{x_i \leftarrow t_i\sigma \mid x_i \neq t_i\sigma\} \cup \{y_j \leftarrow s_j \mid y_j \neq x_i \text{ for all } x_i\}.$$

In other words, we apply the substitution σ to each of the terms t_i given by θ, and we append to this the substitutions produced by σ whose variables didn't already appear in θ.

Example 13. Consider the expression $E = \{P(u, v, x, y, z)\}$ together with the substitutions

$$\theta = \{x \leftarrow f(y), y \leftarrow f(a), z \leftarrow u\} \text{ and}$$
$$\sigma = \{y \leftarrow g(a), u \leftarrow z, v \leftarrow f(f(a))\}.$$

Combining these substitutions gives us

$$\theta\sigma = \{x \leftarrow f(g(a)), y \leftarrow f(a), u \leftarrow z, v \leftarrow f(f(a))\}.$$

Observe that we omitted the substitution $(z \leftarrow u)\sigma$, since that would produce $z \leftarrow z$, and this substitution isn't permitted under our definition. We can then apply this substitution to our expression to obtain

$$(E\theta)\sigma = \{P(z, f(f(a)), f(g(a)), f(a), z)\}.$$

It's worth noting that the composition of substitutions is associative (i.e., $\theta(\sigma\lambda) = (\theta\sigma)\lambda$ for all substitutions θ, σ, and λ), but is not generally commutative (i.e., $\theta\sigma \neq \sigma\theta$ for all substitutions θ and σ).

Now, let's connect substitution to our resolution procedure. Recall that our primary motivation for revisiting substitution was to extend our ground resolution procedure, since ground resolution on its own wasn't particularly useful for predicate logic formulas. If we consider sets of clauses involving variables, we can't immediately identify whether there exists a complementary pair of literals. Following an appropriate substitution, however, clashes can become more easily identifiable.

Example 14. Consider the set of clauses $\{P(f(x), g(y)), \neg P(f(f(a)), g(z))\}$. At first glance, it appears that the two clauses in this set don't clash. However, if we were to apply the substitution

$$\theta = \{x \leftarrow f(a), y \leftarrow a, z \leftarrow a\},$$

then we would obtain the set of clauses $\{P(f(f(a)), g(a)), \neg P(f(f(a)), g(a))\}$, which obviously clashes, and we can use ground resolution on this modified set of clauses to discover this clash.

In our general resolution procedure, we can make use of substitutions to "draw out" complementary pairs of literals. We refer to a substitution used in this way as a *unifier*.

Definition 15 (Unifier)**.** Let $U = \{A_1, \ldots, A_n\}$ be a set of clauses. A unifier θ is a substitution where $A_1\theta = \cdots = A_n\theta$. The most general unifier for U is a unifier μ with the property that any unifier θ of U can be expressed as a composition involving μ; that is, $\theta = \mu\lambda$ for some other substitution λ.

Thus, we can see that the substitution θ in Example 14 is a unifier for the given set of clauses. In most cases, it is possible for us to find a unifier for any set of clauses. The only situations in which it is impossible to find a unifier is (i) when the set of clauses contains predicate symbols that are different, such as $\{P(x), Q(x)\}$; and (ii) when the set of clauses contains a variable that appears both independently and within another term, such as the variable x in the set $\{P(a, x), P(a, f(x))\}$.

Having defined all of the necessary concepts we need, we can develop an algorithmic method of performing *unification* on a set of clauses. If we want to unify, for example, the clauses $P(t_1, t_2, \ldots, t_n)$ and $P(s_1, s_2, \ldots, s_n)$, then we must unify each of the pairs of terms t_1 and s_1 through t_n and s_n. We can denote this concisely by the system of *term equations*

$$\begin{aligned} t_1 &= s_1; \\ t_2 &= s_2; \\ &\vdots \\ t_n &= s_n. \end{aligned}$$

If we are given a system of term equations, then unification will bring these equations into their *solved form*, where

- all term equations are of the form $x_i = t_i$, where x_i is a variable and t_i is a term not containing x_i; and
- every variable x_i appearing on the left-hand side of some term equation does not appear in any other term equation.

With this solved form, we can obtain a unifier directly from the substitution $\{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$ arising from each solved term equation.

Our unification algorithm takes as input a system of term equations and transforms those equations to their solved form by applying the following steps wherever applicable:

1. For any term equation $t = x$, where t is a term that is not a variable and x is a variable, transform this equation to one of the form $x = t$.
2. For any term equation of the form $x = x$, where x is a variable, remove this equation from the system.
3. For any term equation of the form $t' = t''$, where t' and t'' are terms that are not variables:
   (a) If the outermost function symbols of t' and t'' are not the same, then halt and report that the system is not unifiable.
   (b) Otherwise, replace the term equation $f(t'_1, \ldots, t'_k) = f(t''_1, \ldots, t''_k)$ by the k term equations $t'_1 = t''_1$ through $t'_k = t''_k$.

4. For any term equation of the form $x = t$, where the variable x appears elsewhere in the system:
   (a) If x occurs in t and x and t are not the same, then halt and report that the system is not unifiable.
   (b) Otherwise, replace all occurrences of x in all other term equations by t.

Example 16. Consider the following system of term equations:

$$\{g(y) = x; \quad f(x, h(x), y) = f(g(z), w, z)\}.$$

Following our unification algorithm, we can apply Step 1 with the first equation, followed by applying Step 3(b) with the second equation. This will produce the system

$$\{\boldsymbol{x = g(y)}; \quad \boldsymbol{x = g(z)}; \quad \boldsymbol{h(x) = w}; \quad \boldsymbol{y = z}\}.$$

From here, apply Step 4(b) with the second equation to obtain the system

$$\{\boldsymbol{g(z)} = g(y); \quad x = g(z); \quad h(\boldsymbol{g(z)}) = w; \quad y = z\}.$$

Next, apply Step 3(b) with the first equation to obtain the system

$$\{\boldsymbol{z = y}; \quad x = g(z); \quad h(g(z)) = w; \quad y = z\}.$$

Now, apply Step 4(b) with the fourth equation to replace the variable y in the first equation with z. This allows us to use Step 2 to remove the resultant equation $z = z$ and obtain the system

$$\{x = g(z); \quad h(g(z)) = w; \quad y = z\}.$$

Finally, apply Step 1 to the second equation to obtain the system

$$\{x = g(z), \quad \boldsymbol{w = h(g(z))}; \quad y = z\}.$$

This system of term equations is now in its solved form, and from this we can obtain the unifier

$$\theta = \{x \leftarrow g(z), w \leftarrow h(g(z)), y \leftarrow z\}.$$

2.3 General Resolution

Combining ground resolution with our unification algorithm gives us a general resolution procedure for predicate logic formulas. Our general resolution rule will be largely the same as our ground resolution rule, but with the addition of unifiers. In this rule, we will use the notation $L = \{\ell_1, \ldots, \ell_n\}$ to denote a set of literals, while the similar notation $L^{\mathrm{C}} = \{\ell_1^{\mathrm{C}}, \ldots, \ell_n^{\mathrm{C}}\}$ will denote the set of complementary literals.

General resolution rule. Suppose that C_1 and C_2 are clauses having no variables in common.[2] Let $L_1 = \{\ell_1^1, \ldots, \ell_m^1\} \subseteq C_1$ and $L_2 = \{\ell_1^2, \ldots, \ell_n^2\} \subseteq C_2$ be subsets of literals where L_1 and L_2^{C} can be unified by a most general unifier μ. In this case, C_1 and C_2 are said to clash on the sets of literals L_1 and L_2. From this, we can determine the resolvent of C_1 and C_2,

$$\mathrm{Res}(C_1, C_2) = (C_1\mu \setminus L_1\mu) \cup (C_2\mu \setminus L_2\mu).$$

Further note that, whenever we apply the general resolution rule to clauses, identical literals are collapsed when we take the union of the two clauses. This process is sometimes referred to as *factoring*.

Example 17. Suppose we have the clauses $C_1 = \{P(f(x), y)\}$ and $C_2 = \{\neg P(x, a)\}$. Since C_1 and C_2 share variables, we must first standardize the variables apart to obtain the clause $C_2' = \{\neg P(z, a)\}$.

These two clauses have the most general unifier

$$\mu = \{z \leftarrow f(x), y \leftarrow a\}.$$

We can then apply the general resolution rule to these clauses to find that $\mathrm{Res}(C_1, C_2) = \square$.

[2] If C_1 and C_2 share some variable, then we can rectify this issue by standardizing the variables apart.

Our technique for performing general resolution, given an input predicate logic formula in clausal form, is summarized in the following algorithm.

Algorithm 1: Method of general resolution

$S_0 \leftarrow S$
$i \leftarrow 0$
while S_i exists **do** ▷ I.e., while we can apply the resolution rule, assuming S_i is constructible
  $C_1, C_2 \leftarrow$ some pair of clashing clauses in S_i
  $C \leftarrow \text{Res}(C_1, C_2)$ ▷ we may need to standardize variables apart here
  if $C = \square$ **then**
    return S is not satisfiable
  $S_{i+1} \leftarrow S_i \cup \{C\}$
if $S_{i+1} = S_i$ for all pairs of clashing clauses in S_i **then**
  return S is satisfiable

Example 18. Let's use our algorithm to determine whether the set of clauses

1. $\neg P(x), Q(x), R(x, f(x))$
2. $\neg P(x), Q(x), S(f(x))$
3. $T(a)$
4. $P(a)$
5. $\neg R(a, y), T(y)$
6. $\neg T(x), \neg Q(x)$
7. $\neg T(x), \neg S(x)$

is satisfiable. Each of the following lines gives the resolvent of two clashing clauses, as well as their most general unifier.

	Resolvent	**MGU**	**Clauses**
8.	$\neg Q(a)$	$x \leftarrow a$	3, 6
9.	$\neg P(a), S(f(a))$	$x \leftarrow a$	2, 8
10.	$\neg P(a), R(a, f(a))$	$x \leftarrow a$	1, 8
11.	$S(f(a))$	—	4, 9
12.	$R(a, f(a))$	—	4, 10
13.	$T(f(a))$	$y \leftarrow f(a)$	5, 12
14.	$\neg T(f(a))$	$x \leftarrow f(a)$	7, 11
15.	$\square$	—	13, 14

Since we were able to obtain the empty clause $\square$ from the clashing clauses $T(f(a))$ and $\neg T(f(a))$ via the resolution rule, we conclude that the original set of clauses is not satisfiable.