

Introduction to Logic

Resolution

Michael Genesereth
Computer Science Department
Stanford University

Problem With Natural Deduction

Assumption

$p(a)$
 $p(f(a))$
 $p(x) \Rightarrow q(x)$
 $\sim p(x) \Rightarrow q(x)$

Universal Elimination

$\forall x. \forall y. q(x, y)$
 $q(a, a)$ $q(a, [c])$
 $q(a, b)$ $q(a, [c])$
 $q(a, f(a))$ $q(a, f([c]))$
... ...

One of the problems of natural deduction systems like Fitch is that there are rules that require that we choose from infinitely many possible instances. For example, the Assumption rule allows us to assume any legal sentence. And the universal elimination rule allows us to substitute almost any legal term for the variables in universally quantified sentences.

Resolution Principle

The *Resolution Principle* is a rule of inference. Using the Resolution Principle *alone*, it is possible to build a proof system (called *Resolution*) that can prove everything that can be proved in Fitch (without domain closure or induction).

There is no need to make arbitrary assumptions or replacements.

The search space using the Resolution Principle is better controlled than that of natural deduction systems.

Relies on functional notation.

Resolution is a proof system for Herbrand Logic that does not have this problem. As we shall see, Herbrand Resolution allows us to derive conclusions from premises without making arbitrary substitutions.

Like Propositional Resolution, Herbrand Resolution relies on a single rule of inference called the Resolution Principle. Using the resolution principle alone (without axiom schemata or other rules of inference), it is possible to build a reasoning system that is able to prove everything that can be proved in Fitch. Moreover, the search space using the resolution principle is smaller than the search space for generating Fitch proofs.

Programme

Clausal Form

Unification

Resolution Rule of Inference

Unsatisfiability

Logical Entailment

Answer Extraction

This lesson is devoted entirely to resolution. We start with a look at clausal form, a variation of the language of Herbrand logic. We then discuss unification, which is a key to the power of Herbrand resolution. And then we examine the resolution rule of inference in detail. Finally, we show how the rule is used in determining unsatisfiability, in checking logical entailment, and in extracting answers to fill-in-the blanks type questions.

Clausal Form

Clausal Form

Resolution works only on expressions in *clausal form*.

Fortunately, it is possible to convert any set of sentences into an equally satisfiable set of expressions in clausal form.

As with propositional resolution, Herbrand Resolution works only on expressions in clausal form. Fortunately, it is possible to convert any set of Herbrand Logic sentences into an equally satisfiable set of expressions in clausal form.

Clausal Form

A *literal* is either an atomic sentence or a negation of an atomic sentence.

$$p(a), \neg p(y)$$

A *clausal sentence* is either a literal or a disjunction of literals.

$$p(a), \neg p(y), p(a) \vee \neg p(y)$$

A *clause* is a set of literals.

$$\{p(a)\}, \{\neg p(y)\}, \{p(a), \neg p(y)\}$$

The definitions here are analogous to those for Propositional resolution. A literal is either a relational sentence or a negation of a relational sentence. A clausal sentence is either a literal or a disjunction of literals. A clause is a set of literals and, as in propositional logic, represents a disjunction of the literals in the set.

Empty Sets

The empty clause $\{\}$ is unsatisfiable.

Why? It is equivalent to an empty disjunction.

As with Propositional clausal form, the empty set is also a clause. It is equivalent to an empty disjunction and, therefore, is unsatisfiable.

Inseado

Implications Out:

$$\varphi_1 \Rightarrow \varphi_2 \rightarrow \neg \varphi_1 \vee \varphi_2$$

$$\varphi_1 \Leftrightarrow \varphi_2 \rightarrow (\neg \varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \neg \varphi_2)$$

The procedure for converting relational sentences to clausal form is similar to that for Propositional Logic. It goes by the catchy name of Inseado, which is an acronym for the steps in the procedure. Some of the rules are the same as those of propositional logic. However, there are a few additional rules to deal with variables and quantifiers.

In the first step (Implications out), we eliminate all occurrences of the $\Rightarrow$ and $\Leftrightarrow$ operators by substituting equivalent sentences involving only the $\&$, $\vee$, and $\sim$ operators.

Inseado

Implications Out:

$$\begin{aligned}\varphi_1 \Rightarrow \varphi_2 &\rightarrow \neg \varphi_1 \vee \varphi_2 \\ \varphi_1 \Leftrightarrow \varphi_2 &\rightarrow (\neg \varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \neg \varphi_2)\end{aligned}$$

Negations In:

$$\begin{aligned}\neg \neg \varphi &\rightarrow \varphi \\ \neg(\varphi_1 \wedge \varphi_2) &\rightarrow \neg \varphi_1 \vee \neg \varphi_2 \\ \neg(\varphi_1 \vee \varphi_2) &\rightarrow \neg \varphi_1 \wedge \neg \varphi_2 \\ \neg \forall v. \varphi &\rightarrow \exists v. \neg \varphi \\ \neg \exists v. \varphi &\rightarrow \forall v. \neg \varphi\end{aligned}$$

In the second step (Negations in), negations are distributed over other logical operators and quantifiers until each such operator applies to a single atomic sentence. The replacement rules shown here do the job. We eliminate double negations. We use deMorgan's laws to distribute negations over conjunctions and disjunctions; and we distribute negations over quantifiers. When this step is done, any remaining negations are applied directly to atomic sentences.

Inseado

Standardize variables

$$\forall x. p(x) \vee \forall x. q(x) \rightarrow \forall x. p(x) \vee \forall y. q(y)$$

In the third step (Standardize variables), we rename variables so that each quantifier has a unique variable, i.e. the same variable is not quantified more than once within the same sentence. For example, in the case shown here, we rename the variable x in the second disjunct to y.

Inseado

Standardize variables

$$\forall x. p(x) \vee \forall x. q(x) \rightarrow \forall x. p(x) \vee \forall y. q(y)$$

Existentials Out (Outside in)

$$\exists x. p(x) \rightarrow p(a)$$

In the fourth step (Existentials out), we eliminate all existential quantifiers. The method for doing this is a little complicated, and there are two cases.

If an existentially quantified sentence does not contain any free variables and does not occur within the scope of a universal quantifier, we simply drop the existential quantifier and replace all occurrences of the quantified variable by a new object constant; i.e., one that is NOT part of the language. The constant used to replace the existential variable in this case is called a Skolem constant. In the example shown here we have replaced x by a (on the assumption that a was not in the original language).

Inseado

Standardize variables

$$\forall x.p(x) \vee \forall x.q(x) \rightarrow \forall x.p(x) \vee \forall y.q(y)$$

Existentials Out (Outside in)

$$\exists x.p(x) \rightarrow p(a)$$

$$\forall x.(p(x) \wedge \exists z.q(x, y, z)) \rightarrow \forall x.(p(x) \wedge q(x, y, f(x, y)))$$

If an existential quantified sentence contains free variables or lies within the scope of any universal quantifiers, there is the possibility that the value of the existential variable depends on the values of the associated universal variables. Consequently, we cannot replace the existential variable with a constant. Instead, the general rule is to drop the existential quantifier and to replace the associated variable by a term formed from a new function symbol applied to the variables associated with the free variables and universally quantified variables in the target. Any function defined in this way is called a Skolem function. The following example illustrates this transformation. It assumes f is not part of the language. In the case shown here, the existentially quantified sentence contains a universally quantified variable x and a free variable y , and so we create a term by taking f and applying it to x and y .

Note that, properly, our Skolem terms should be enclosed in brackets. However, for simplicity, we drop the brackets in our examples in this lecture.

Inseado

Alls Out

$$\forall x.(p(x) \wedge q(x, y, f(x, y))) \rightarrow p(x) \wedge q(x, y, f(x, y))$$

In the fifth step (Alls out), we drop all universal quantifiers. Because the remaining variables at this point are universally quantified, this does not introduce any ambiguities.

Inseado

Alls Out

$$\forall x.(p(x) \wedge q(x, y, f(x, y))) \rightarrow p(x) \wedge q(x, y, f(x, y))$$

Distribution

$$\varphi_1 \vee (\varphi_2 \wedge \varphi_3) \rightarrow (\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \varphi_3)$$

$$(\varphi_1 \wedge \varphi_2) \vee \varphi_3 \rightarrow (\varphi_1 \vee \varphi_3) \wedge (\varphi_2 \vee \varphi_3)$$

$$\varphi \vee (\varphi_1 \vee \dots \vee \varphi_n) \rightarrow (\varphi \vee \varphi_1 \vee \dots \vee \varphi_n)$$

$$(\varphi_1 \vee \dots \vee \varphi_n) \vee \varphi \rightarrow (\varphi_1 \vee \dots \vee \varphi_n \vee \varphi)$$

$$\varphi \wedge (\varphi_1 \wedge \dots \wedge \varphi_n) \rightarrow (\varphi \wedge \varphi_1 \wedge \dots \wedge \varphi_n)$$

$$(\varphi_1 \wedge \dots \wedge \varphi_n) \wedge \varphi \rightarrow (\varphi_1 \wedge \dots \wedge \varphi_n \wedge \varphi)$$

In the sixth step (Disjunctions in), we put the expression into conjunctive normal form, i.e. a conjunction of disjunctions of literals. This can be accomplished by repeated use of the distribution listed here rules.

Inseado

Operators Out

$$\begin{aligned} \varphi_1 \wedge \dots \wedge \varphi_n &\rightarrow \varphi_1 \\ &\dots \\ &\varphi_n \\ \varphi_1 \vee \dots \vee \varphi_n &\rightarrow \{\varphi_1, \dots, \varphi_n\} \end{aligned}$$

In the seventh and final step (Operators out), we eliminate operators by separating any conjunctions into its conjuncts and writing each disjunction as a separate clause.

Example

$$\begin{aligned} &\exists y.(g(y) \wedge \forall z.(r(z) \Rightarrow f(y,z))) \\ \text{I } &\exists y.(g(y) \wedge \forall z.(\neg r(z) \vee f(y,z))) \\ \text{N } &\exists y.(g(y) \wedge \forall z.(\neg r(z) \vee f(y,z))) \\ \text{S } &\exists y.(g(y) \wedge \forall z.(\neg r(z) \vee f(y,z))) \\ \text{E } &g(\text{greg}) \wedge \forall z.(\neg r(z) \vee f(\text{greg},z)) \\ \text{A } &g(\text{greg}) \wedge (\neg r(z) \vee f(\text{greg},z)) \\ \text{D } &g(\text{greg}) \wedge (\neg r(z) \vee f(\text{greg},z)) \\ \text{O } &\{g(\text{greg})\} \\ &\{\neg r(z), f(\text{greg},z)\} \end{aligned}$$

As an example of this conversion process, consider the problem of transforming the following expression to clausal form. The initial expression appears on the top line, and the expressions on the labeled lines are the results of the corresponding steps of the conversion procedure. On the I step, we get rid of the implication operator. The N step has no effect since the only negation in the sentence is already applied to an atomic sentence. The S step has no effect because there are no duplicate variables. On the E step, we notice that the quantified sentence contains no free variables, so we simply replace the quantified variable by a new constant, in this case greg. The A step and the D step have no effect. Finally, on the O step, we convert our conjunction into two separate clauses.

Example

$$\begin{aligned} &\neg \exists y.(g(y) \wedge \forall z.(r(z) \Rightarrow f(y,z))) \\ \text{I } &\neg \exists y.(g(y) \wedge \forall z.(\neg r(z) \vee f(y,z))) \\ \text{N } &\neg \exists y.(g(y) \wedge \forall z.(\neg r(z) \vee f(y,z))) \\ &\forall y. \neg (g(y) \wedge \forall z.(\neg r(z) \vee f(y,z))) \\ &\forall y. (\neg g(y) \vee \neg \forall z.(\neg r(z) \vee f(y,z))) \\ &\forall y. (\neg g(y) \vee \exists z. \neg (\neg r(z) \vee f(y,z))) \\ &\forall y. (\neg g(y) \vee \exists z. (\neg \neg r(z) \wedge \neg f(y,z))) \\ &\forall y. (\neg g(y) \vee \exists z. (r(z) \wedge \neg f(y,z))) \\ \text{S } &\forall y. (\neg g(y) \vee \exists z. (r(z) \wedge \neg f(y,z))) \end{aligned}$$

Here is another example. In this case, the starting sentence is almost the same. The only difference is the leading $\sim$, but the result looks quite different. As before, on the I step, we get rid of the implication operator. This time the N step is more interesting. We need to push the negation sign down through all of the subexpressions. First, distribute over the existential, flipping it to a universal and negating the target. We then distribute the negation over the conjunction, leading to a disjunction of negated disjuncts. We distribute the negation in the second disjunct over the universal quantifier, flipping it to an existential and negating the target. We distribute that negation over the disjunction. Finally, we eliminate the double negation. The S step here has no effect.

Example

$\forall y.(\neg g(y) \vee \exists z.(r(z) \wedge \neg f(y, z)))$

E $\forall y.(\neg g(y) \vee (r(h(y)) \wedge \neg f(y, h(y))))$

A $\neg g(y) \vee (r(h(y)) \wedge \neg f(y, h(y)))$

D $(\neg g(y) \vee r(h(y))) \wedge (\neg g(y) \vee \neg f(y, h(y)))$

O $\{\neg g(y), r(h(y))\}$

$\{\neg g(y), \neg f(y, h(y))\}$

On the E step, we note that the existentially quantified sentence does contain a free variable, so we replace the existentially quantified variable with a term consisting of a Skolem function and the free variable y . On the A step, we drop the universal quantifier. On the D step, we distribute the disjunction over the conjunction. Finally, on the O step, we convert our conjunction of disjunctions to two separate clauses.

Clausal Form

Bad News: The result of converting a set of sentences is not necessarily logically equivalent to the original set of sentences. Why? Introduction of new constants and functions.

Good News: The result of converting a set of sentences is satisfiable in the expanded language if and only if the original set of sentences is satisfiable in the original language. Important because we use satisfiability to determine logical entailment.

Potential problem with domain closure and induction eliminated by keeping new constants separate from old.

In Propositional Logic, the clause set corresponding to any sentence is logically equivalent to that sentence. In Herbrand Logic, this is not necessarily the case. For example, the clausal form of the sentence $\exists x.p(x)$ is $\{p(a)\}$. This is not logically equivalent. It is not even in the same language. Since the clause exists in a language with an additional object constant, there are truth assignments that satisfy the sentence but not the clause. On the other hand, the clause is effectively equivalent to the original sentence in the sense that it is satisfiable in the expanded language if and only if the original sentence is satisfiable in the original language. As we shall see, in resolution, this equivalence of satisfiability is all we need.

Unification

Unification

Unification is the process of determining whether two expressions can be *unified*, i.e. made identical by appropriate substitutions for their variables.

Unification is the process of determining whether two expressions can be unified, i.e. made identical by appropriate substitutions for their variables. As we shall see, making this determination is an essential part of resolution.

Substitutions

A *substitution* is a finite set of pairs of variables and terms, called *replacements*.

$$\{x \leftarrow a, y \leftarrow f(b), v \leftarrow w\}$$

The result of applying a substitution σ to an expression φ is the expression $\varphi\sigma$ obtained from φ by replacing every occurrence of every variable in the substitution by its replacement.

$$p(x, x, y, z)\{x \leftarrow a, y \leftarrow f(b), v \leftarrow w\} = p(a, a, f(b), z)$$

A **substitution** is a finite mapping of variables to terms. In what follows, we write substitutions as sets of replacement rules, like the one shown here. In each rule, the variable to which the arrow is pointing is to be replaced by the term from which the arrow is pointing. In this case, x is to be replaced by a , y is to be replaced by $f(b)$, and v is to be replaced by w .

The result of applying a substitution σ to an expression φ is the expression $\varphi\sigma$ obtained from φ by replacing every occurrence of every variable in the domain of the substitution by the term with which it is associated. For example, applying our substitution to the expression $p(x, x, y, z)$ yields $p(a, a, f(b), z)$.

Cascaded Substitutions

$$r\{x, y, z\}\{x \leftarrow a, y \leftarrow f(u), z \leftarrow v\} = r\{a, f(u), v\}$$

$$r\{a, f(u), v\}\{u \leftarrow d, v \leftarrow e, z \leftarrow g\} = r\{a, f(d), e\}$$

$$r\{x, y, z\}\{x \leftarrow a, y \leftarrow f(d), z \leftarrow e, u \leftarrow d, v \leftarrow e\} = r\{a, f(d), e\}$$

Given two or more substitutions, it is possible to define a single substitution that has the same effect as applying those substitutions in sequence. For example, the substitutions $\{x \leftarrow a, y \leftarrow f(u), z \leftarrow v\}$ and $\{u \leftarrow d, v \leftarrow e\}$ can be combined to form the single substitution $\{x \leftarrow a, y \leftarrow f(d), z \leftarrow e, u \leftarrow d, v \leftarrow e\}$, which has the same effect as the first two substitutions when applied to any expression whatsoever.

Composition of Substitutions

The *composition* of substitution σ and τ is the substitution (written $compose(\sigma, \tau)$ or, more simply, $\sigma\tau$) obtained by

- (1) applying τ to the replacements in σ
- (2) adding to σ pairs from τ with different variables
- (3) deleting any assignments of a variable to itself.

$$\begin{aligned} & \{x \leftarrow a, y \leftarrow f(u), z \leftarrow v\} \{u \leftarrow d, v \leftarrow e, z \leftarrow g\} \\ &= \{x \leftarrow a, y \leftarrow f(d), z \leftarrow e\} \{u \leftarrow d, v \leftarrow e, z \leftarrow g\} \\ &= \{x \leftarrow a, y \leftarrow f(d), z \leftarrow e, u \leftarrow d, v \leftarrow e\} \end{aligned}$$

Computing the composition of a substitution sigma and a substitution tau is easy. There are three steps. (1) First, we apply tau to the replacements in sigma. (2) Then we adjoin to sigma all pairs from tau with different variables, and finally (3) we delete any assignments of variables to themselves.

Unification

A substitution σ is a *unifier* for an expression ϕ and an expression ψ if and only if $\phi\sigma = \psi\sigma$.

$$\begin{aligned} p(x,y)\{x \leftarrow a, y \leftarrow b, v \leftarrow b\} &= p(a,b) \\ p(a,v)\{x \leftarrow a, y \leftarrow b, v \leftarrow b\} &= p(a,b) \end{aligned}$$

If two expressions have a unifier, they are said to be *unifiable*. Otherwise, they are *nonunifiable*.

$$\begin{aligned} & p(x,x) \\ & p(a,b) \end{aligned}$$

A substitution sigma is a **unifier** for an expression phi and an expression psi if and only if $\phi\sigma = \psi\sigma$, i.e. the result of applying sigma to phi is the same as the result of applying sigma to psi. If two expressions have a unifier, they are said to be **unifiable**. Otherwise, they are **nonunifiable**.

NonUniqueness of Unification

Unifier 1:

$$\begin{aligned} p(x,y)\{x \leftarrow a, y \leftarrow b, v \leftarrow b\} &= p(a,b) \\ p(a,v)\{x \leftarrow a, y \leftarrow b, v \leftarrow b\} &= p(a,b) \end{aligned}$$

Unifier 2:

$$\begin{aligned} p(x,y)\{x \leftarrow a, y \leftarrow f(w), v \leftarrow f(w)\} &= p(a, f(w)) \\ p(a,v)\{x \leftarrow a, y \leftarrow f(w), v \leftarrow f(w)\} &= p(a, f(w)) \end{aligned}$$

Unifier 3:

$$\begin{aligned} p(x,y)\{x \leftarrow a, y \leftarrow v\} &= p(a,v) \\ p(a,v)\{x \leftarrow a, y \leftarrow v\} &= p(a,v) \end{aligned}$$

Note that, although this substitution unifies the two expressions, it is not the only unifier. We do not have to substitute b for y and v to unify the two expressions. We can equally well substitute c or d or f(c) or f(w). In fact, we can unify the expressions without changing v at all by simply replacing y by v.

In considering these alternatives, it should be clear that some substitutions are more general than others are. We say that a substitution sigma is **as general as or more general than** a substitution tau if and only if there is another substitution delta such that tau is the composition of sigma and delta. For example, the substitution at the bottom here is more as general as or more general than the other two since each can be obtained by composing the unifier at the bottom with another substitution. And, in fact, it is more general than the other two in that reverse does not hold. There is no way to convert either of the first two unifiers into the third.

Most General Unifier

A substitution σ is a *most general unifier (mgu)* of two expressions if and only if it is as general as or more general than any other unifier.

Theorem: If two expressions are unifiable, then they have an mgu that is unique up to variable permutation.

$$\begin{aligned}p(x,y)\{x \leftarrow a, y \leftarrow v\} &= p(a,v) \\ p(a,v)\{x \leftarrow a, y \leftarrow v\} &= p(a,v)\end{aligned}$$

$$\begin{aligned}p(x,y)\{x \leftarrow a, v \leftarrow y\} &= p(a,y) \\ p(a,v)\{x \leftarrow a, v \leftarrow y\} &= p(a,y)\end{aligned}$$

In resolution, we are interested only in unifiers with maximum generality. A most general unifier, or mgu, sigma of two expressions has the property that it is as general as or more general than any other unifier.

Although it is possible for two expressions to have more than one most general unifier, all of these most general unifiers are structurally the same, i.e. they are unique up to variable renaming. For example, $p(x)$ and $p(y)$ can be unified by either the substitution $\{x \leftarrow y\}$ or the substitution $\{y \leftarrow x\}$; and either of these substitutions can be obtained from the other by applying a third substitution. This is not true of the unifiers mentioned earlier.

Unification Procedure

One good thing about our language is that there is a simple and inexpensive procedure for computing a most general unifier of any two expressions if it exists.

One good thing about our language is that there is a simple and inexpensive procedure for computing a most general unifier of any two expressions if it exists.

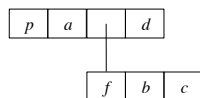
Expression Structure

Each expression is treated as a sequence of its immediate subexpressions.

Linear Version:

$$p(a, f(b, c), d)$$

Structured Version:



The procedure assumes a representation of expressions as sequences of subexpressions. For example, the expression $p(a, f(b), z)$ can be thought of as a sequence with four elements, viz. the relation constant p , the object constant a , the term $f(b)$, and the variable z . The term $f(b)$ can in turn be thought of as a sequence of two elements, viz. the function constant f and the object constant b .

Unification Procedure

- (1) If two expressions being compared are identical, succeed.
- (2) If neither is a variable and at least one is a constant, fail.
- (3) If at least one of the expressions is a variable, proceed as described shortly.
- (4) If both expressions are sequences, iterate across the expressions, comparing as described above.

We start the procedure with two expressions and a substitution, which is initially the empty substitution. We then recursively process the two expressions, comparing the subexpressions at each point. Along the way, we expand the substitution with variable assignments as described below. If, we fail to unify any pair of subexpressions at any point in this process, the procedure as a whole fails. If we finish this recursive comparison of the expressions, the procedure as a whole succeeds, and the accumulated substitution at that point is the most general unifier.

If the two expressions being compared are identical, we succeed. (2) If neither is a variable and at least one is a constant, we fail. (3) If one of the expressions is a variable, we proceed as described shortly. (4) If both expressions are sequences, we iterate across the expressions, comparing as just described.

Dealing With Variables

If one of the expressions is a variable, check whether the variable has a binding in the current substitution.

- (a) If so, try to unify the binding with the other expression.
- (b) If no binding, check whether the other expression contains the variable. If the variable occurs within the expression, fail; otherwise, set the substitution to the composition of the old substitution and a new substitution in which variable is bound to the other expression.

If one of the expressions is a variable, we first check whether the variable has a binding in the current substitution. If so, we try to unify the binding with the second expression. If there is no binding, we check whether the second expression contains the variable. If the variable occurs within the expression, we fail; otherwise, we set the substitution to the composition of the old substitution and a new substitution in which we bind the variable to the second expression.

Example

Call: $p(x,b), p(a,y), \{\}$
Call: $p, p, \{\}$
Exit: $\{\}$
Call: $x, a, \{\}$
Exit: $\{\}\{x \leftarrow a\} = \{x \leftarrow a\}$
Call: $b, y, \{x \leftarrow a\}$
Exit: $\{x \leftarrow a\}\{y \leftarrow b\} = \{x \leftarrow a, y \leftarrow b\}$
Exit: $\{x \leftarrow a, y \leftarrow b\}$

As an example, consider the computation of the most general unifier for the expressions $p(x,b)$ and $p(a,y)$ with the initial substitution $\{\}$. A trace of the execution of the procedure for this case is shown here. We show the beginning of a comparison with a line labelled Call together with the expressions being compared and the input substitution. We show the result of each comparison with a line labelled Exit. The indentation shows the depth of recursion of the procedure. On the first nested step, we compare the first element of the first expression with the first element of the second expression. Since they are identical, we succeed with no changes to our substitution. We then compare the variable x to the constant a , leading to a binding of x to a . Finally, we compare b to y . In this case, we compose the current variable assignment with an assignment of y to b to produce the joint assignment $x \leftarrow a$ and $y \leftarrow b$. Since we have reached the ends of the two expressions, we are done.

Example

Call: $p(x,x), p(a,y), \{\}$
 Call: $p, p, \{\}$
 Exit: $\{\}$
 Call: $x, a, \{\}$
 Exit: $\{x \leftarrow a\} = \{x \leftarrow a\}$
 Call: $x, y, \{x \leftarrow a\}$
 Call: $a, y, \{x \leftarrow a\}$
 Exit: $\{x \leftarrow a\} \{y \leftarrow a\} = \{x \leftarrow a, y \leftarrow a\}$
 Exit: $\{x \leftarrow a, y \leftarrow a\}$
Exit: $\{x \leftarrow a, y \leftarrow a\}$

As another example, consider the process of unifying the expression $p(x,x)$ and the expression $p(a,y)$. A trace is shown here. The main interest in this example comes in the step involving x and y . At this point, x has a binding, so we recursively compare the replacement a to y , which results in binding of y to a .

Example

Call: $p(x,x), p(a,b), \{\}$
 Call: $p, p, \{\}$
 Exit: $\{\}$
 Call: $x, a, \{\}$
 Exit: $\{x \leftarrow a\} = \{x \leftarrow a\}$
 Call: $x, b, \{x \leftarrow a\}$
 Call: $a, b, \{x \leftarrow a\}$
 Exit: *false*
 Exit: *false*
Exit: *false*

Here is an example where we two expressions that do **not** unify. We start as before. This time, when we compare the second x to b , we get a different result. As before, we retrieve the binding of x , viz. a , and try to unify that with b . Since we have two different constants, we fail, and the unification as a whole fails as well.

Example

Call: $p(x,x), p(yf(y)), \{\}$
 Call: $p, p, \{\}$
 Exit: $\{\}$
 Call: $x, y, \{\}$
 Exit: $\{x \leftarrow y\} = \{x \leftarrow y\}$
 Call: $x, f(y), \{x \leftarrow y\}$
 Call: $y, f(y), \{x \leftarrow y\}$
 Exit: *false*
 Exit: *false*
Exit: *false*

One more example. This one is similar to the last except that the second term in the second expression is functional term that contains y . After unifying the first two arguments, we find ourselves trying to unify x and $f(y)$ with a substitution where x is bound to y . This requires us to unify y and $f(y)$. One of the conditions in the unification procedure is that no variable can be bound to complex a term that contains that variable. So the unification fails in this case.

Reason

Circularity Problem:

$$\{x \leftarrow f(y), y \leftarrow f(y)\}$$

Unification Problem:

$$p(x, y) \{x \leftarrow f(y), y \leftarrow f(y)\} = p(f(y), f(y))$$

$$p(y, f(y)) \{x \leftarrow f(y), y \leftarrow f(y)\} = p(f(y), f(f(y)))$$

Why do we have this condition? If we were to allow bindings of variables to expressions that contain those variables, it would lead to circular bindings. This in an of itself is not a problem, though it does raise the question of how many times the substitution needs to be applied.. More importantly, it can create a situation where the substitution does not actually unify the input expressions no matter how many times it is applied. As we see here, applying this substitution does not lead to the same result. And it fails no matter how many times we apply it. The second argument in the second case always has one additional application of f .

Solution

Before assigning a variable to an expression, first check that the variable does not occur within that expression.

This is called, oddly enough, the *occur check* test.

Prolog does not do the occur check (and is proud of it).

This is the reason why we check whether a variable is contained in an expression before creating a binding. This is often called the **occur check**, for obvious reasons. Interestingly, not all logical systems do the occur check. Prolog is notable for not performing the occur check (at least by default in most systems). Programmatically it is useful as a way of generating circular data structure. However, logically, it can lead to errors, and so we always do the occur check in resolution.

Resolution Principle

Propositional Resolution

$$\frac{\begin{array}{l} \{\varphi_1, \dots, \varphi_m\} \\ \{\psi_1, \dots, \neg \varphi, \dots, \psi_n\} \end{array}}{\{\varphi_1, \dots, \varphi_m, \psi_1, \dots, \psi_n\}}$$

In Propositional Logic, if we have a clause containing a literal φ and another clause containing a literal $\neg \varphi$, we can use propositional resolution to derive a new clause by collecting together the literals from both clauses except for the complementary pair of literals.

Resolution (simple version)

$$\frac{\begin{array}{l} \{\varphi_1, \dots, \varphi_m\} \\ \{\psi_1, \dots, \neg \psi, \dots, \psi_n\} \end{array}}{\{\varphi_1, \dots, \varphi_m, \psi_1, \dots, \psi_n\} \sigma}$$

where $\sigma = mgu(\varphi, \psi)$

Resolution for Herbrand Logic is similar except that the positive literal and the target of the negative literal do not need to be identical; it suffices that they are unifiable. A slightly simplified version of the rule is shown here. Given a clause with a literal φ and a second clause with a literal $\neg \psi$ such that φ and ψ have a most general unifier σ , we can derive a new clause by collecting together the remaining literals from each clause and applying σ to the result.

Example

$$\frac{\begin{array}{l} \{p(a, y), r(y)\} \\ \{\neg p(x, b), s(x)\} \end{array}}{\{r(y), s(x)\} \{x \leftarrow a, y \leftarrow b\}} \quad \{r(b), s(a)\}$$

Here is an example. Suppose we have a clause containing the literal $p(a, y)$ and another clause containing the literal $\neg p(x, b)$. We can unify $p(a, y)$ and $p(x, b)$ with the most general unifier x gets a and y gets b . Hence, we can resolve these two clauses. We form a new clause consisting of the remaining literals from each clause, in this case $r(y)$ and $s(x)$; we apply the most general unifier to this clause; and we thereby produce their resolvent, viz. the clause consisting of literals $r(b)$ and $s(a)$. We'll see how to use resolution to solve some interesting problems in the next segment. But first we need to refine our definition of the rule to deal with a couple of shortcomings.

Problem

$$\frac{\begin{array}{l} \{ p(a,x) \} \\ \{ \neg p(x,b) \} \end{array}}{\text{Failure}}$$

Consider the two clauses shown here. Given the meaning of these two clauses, it should be possible to resolve them to produce the empty clause. However, the two atomic sentences do not unify. The variable x must be bound to a and b at the same time.

Resolution (improved)

$$\frac{\begin{array}{l} \{ \varphi_1, \dots, \varphi_m \} \\ \{ \psi_1, \dots, \neg \psi_n \} \end{array}}{\begin{array}{l} \{ \varphi_1 \tau, \dots, \varphi_m \tau, \psi_1, \dots, \psi_n \} \sigma \\ \text{where } \sigma = mgu(\varphi \tau, \psi) \\ \text{where } \tau \text{ is a variable renaming on } \varphi \end{array}}$$

Fortunately, this problem can easily be fixed by extending the Resolution Principle slightly as shown below. Before trying to resolve two clauses, we select one of the clauses and rename any variables the clause has in common with the other clause.

Example

$$\frac{\begin{array}{l} \{ p(a,x) \} \\ \{ \neg p(x,b) \} \end{array}}{\text{Failure}} \quad \frac{\begin{array}{l} \{ p(a,y) \} \\ \{ \neg p(x,b) \} \end{array}}{\{ \} \{ x \leftarrow a, y \leftarrow b \}}$$

Renaming solves our problem case. We simply change the variables in one of our clauses before attempting to resolve them. After renaming, we see that the positive literal and the target of the negative literal can be unified. Hence, they can be resolved, in this case leading to the empty clause.

Problem

$$\begin{aligned} &\{p(x), p(y)\} \\ &\{\neg p(u), \neg p(v)\} \\ &\{p(y), \neg p(v)\} \\ &\{p(x), \neg p(v)\} \\ &\{p(y), \neg p(u)\} \\ &\{p(x), \neg p(u)\} \end{aligned}$$

Unfortunately, we are still not quite done. There is one more technicality that must be addressed to finish the story. As stated, even with variable renaming, the rule is not quite good enough. Given the clauses shown here, we should be able to infer the empty clause $\{\}$; however, this is not possible with the preceding definition. The clauses can be resolved in various ways, but the result is never the empty clause.

Factors

If a subset of the literals in a clause Φ has a most general unifier γ , then the clause Φ' obtained by applying γ to Φ is called a *factor* of Φ .

Clause

$$\{p(x), p(f(y)), r(x, y)\}$$

Factors

$$\begin{aligned} &\{p(f(y)), r(f(y), y)\} \\ &\{p(x), p(f(y)), r(x, y)\} \end{aligned}$$

If a subset of the literals in a clause Φ has a most general unifier γ ; then the clause Φ' obtained by applying γ to Φ is called a **factor** of Φ . For example, the literals $p(x)$ and $p(f(y))$ have a most general unifier $\{x \leftarrow f(y)\}$, so the clause $\{p(f(y)), r(f(y), y)\}$ is a factor of $\{p(x), p(f(y)), r(x, y)\}$. Note that any clause is a trivial factor of itself.

Resolution (final version)

Φ

Ψ

$((\Phi' - \{\phi\})\tau \cup (\Psi' - \{\neg\psi\}))\sigma$

where $\phi \in \Phi'$, a factor of Φ

where $\neg\psi \in \Psi'$, a factor of Ψ

where $\sigma = mgu(\phi\tau, \psi)$

where τ is a variable renaming on φ

Using the notion of factors, we can give a complete definition for the resolution principle. Suppose that capital Φ and capital Ψ are two clauses. If there is a literal ϕ in some factor of capital Φ after renaming and there is a literal $\neg\psi$ in some factor of capital Ψ such that ϕ and $\neg\psi$ have a most general unifier σ , then we can form a resolvent of these two clauses as shown here.

Example

$$\begin{array}{rcl}
 \{p(x), p(y)\} & \text{---} & \{p(x)\} \\
 \{\neg p(u), \neg p(v)\} & \text{---} & \{\neg p(u)\} \\
 \hline
 \{p(y), \neg p(v)\} & & \{\} \\
 \{p(x), \neg p(v)\} & & \\
 \{p(y), \neg p(u)\} & & \\
 \{p(x), \neg p(u)\} & &
 \end{array}$$

Using this enhanced definition of resolution, we can solve the problem mentioned above. Once again, consider the premises $\{p(x), p(y)\}$ and $\{\neg p(u), \neg p(v)\}$. The first premise has the factor $\{p(x)\}$, and the second has the factor $\{\neg p(u)\}$, and these two factors resolve to the empty clause in a single step.

Need for Original Clauses

- | | |
|---------------------------|-------------|
| 1. $\{p(a, y), p(x, b)\}$ | Premise |
| 2. $\{\neg p(a, d)\}$ | Premise |
| 3. $\{\neg p(c, b)\}$ | Premise |
| 4. $\{p(x, b)\}$ | 1, 2 |
| 5. $\{\}$ | 3, 4 |
| | |
| 1. $\{p(a, y), p(x, b)\}$ | Premise |
| 2. $\{\neg p(a, d)\}$ | Premise |
| 3. $\{\neg p(c, b)\}$ | Premise |
| 4. $\{p(a, b)\}$ | Factor of 1 |

Note that we cannot just replace a clause by its factors. We must retain the original clauses as well. The example here shows what goes wrong if we try to replace a clause by its factor. The three premises shown at the top lead to the empty clause after two steps of resolution. Now consider the partial derivation shown at the bottom. Same three clauses. The clause on line 4 is a factor of the clause on line 1. If we were to discard the clause on line 1 and replace it by its factor, we would find ourselves stuck; there would be no further resolutions. The upshot of this is that factoring is useful; it allows us to derive conclusions we would not otherwise be able to produce. However, we must consider resolutions with unfactored clauses as well if we want to be sure to produce all possible conclusions. Okay, that's it for our treatment of the resolution rule. Now, it is time to put it to work solving some logic problems. As it turns out, there is quite a bit we can do with it.

Resolution Reasoning

Reasoning with the Resolution Principle is analogous to reasoning with other rules of inference. We start with premises; we apply the Resolution Principle to those premises; we apply it again to the results of those applications; and so forth until we get to our desired conclusion or we run out of things to do.

Resolution Derivation

A *resolution derivation* of a conclusion from a set of premises is a finite sequence of clauses terminating in the conclusion in which each clause is either a premise or the result of applying the resolution principle to earlier elements of the sequence.

As with Propositional Resolution, we define a resolution derivation of a conclusion from a set of premises to be a finite sequence of clauses terminating in the conclusion in which each clause is either a premise or the result of applying the resolution principle to earlier members of the sequence. And, as with Propositional resolution, we do not use the word proof, because we reserve that word for a slightly different concept, which is discussed in the next segment.

A resolution derivation of a conclusion from a set of premises is a finite sequence of clauses terminating in the conclusion in which each clause is either a premise or the result of applying the resolution principle to earlier elements of the sequence.

Note that our definition of resolution derivation is analogous to our definition of linear proof. However, in this case, we do not use the word proof here, because we are reserving that

Example

- | | | |
|----|---|---------|
| 1. | $\{p(\text{art}, \text{bob})\}$ | Premise |
| 2. | $\{p(\text{art}, \text{bud})\}$ | Premise |
| 3. | $\{p(\text{bob}, \text{cal})\}$ | Premise |
| 4. | $\{p(\text{bud}, \text{coe})\}$ | Premise |
| 5. | $\{\neg p(x, y), \neg p(y, z), g(x, z)\}$ | Premise |

As an example, consider a problem in the area kinship relations. Suppose we know that Art is the parent of Bob and Bud; suppose that Bob is the parent of Cal; and suppose that Bud is the parent of Coe. Suppose we also know that grandparents are parents of parents. Starting with these premises, we can use resolution to conclude that Art is the grandparent of Coe. We start with our five premises - four simple clauses for the four facts about the parent relation p and one more complex clause capturing the definition of the grandparent relation g .

Example

- | | | |
|----|---|---------|
| 1. | $\{p(\text{art}, \text{bob})\}$ | Premise |
| 2. | $\{p(\text{art}, \text{bud})\}$ | Premise |
| 3. | $\{p(\text{bob}, \text{cal})\}$ | Premise |
| 4. | $\{p(\text{bud}, \text{coe})\}$ | Premise |
| 5. | $\{\neg p(x, y), \neg p(y, z), g(x, z)\}$ | Premise |
| 6. | $\{\neg p(\text{bob}, z), g(\text{art}, z)\}$ | 1, 5 |

We start by resolving the clause on line 1 with the clause on line 5 to produce the clause on line 6.

Example

1.	$\{p(art,bob)\}$	Premise
2.	$\{p(art,bud)\}$	Premise
3.	$\{p(bob,cal)\}$	Premise
4.	$\{p(bud,coe)\}$	Premise
5.	$\{\neg p(x,y), \neg p(y,z), g(x,z)\}$	Premise
6.	$\{\neg p(bob,z), g(art,z)\}$	1, 5
7.	$\{g(art,cal)\}$	3, 6

We then resolve the clause on line 3 with this result to derive that conclusion that Art is the grandparent of Cal. Interesting but not what we set out to prove; so we continue the process.

Example

1.	$\{p(art,bob)\}$	Premise
2.	$\{p(art,bud)\}$	Premise
3.	$\{p(bob,cal)\}$	Premise
4.	$\{p(bud,coe)\}$	Premise
5.	$\{\neg p(x,y), \neg p(y,z), g(x,z)\}$	Premise
6.	$\{\neg p(bob,z), g(art,z)\}$	1, 5
7.	$\{g(art,cal)\}$	3, 6
8.	$\{\neg p(bud,z), g(art,z)\}$	2, 5

We next resolve the clause on line 2 with the clause on line 5 to produce the clause on line 8.

Example

1.	$\{p(art,bob)\}$	Premise
2.	$\{p(art,bud)\}$	Premise
3.	$\{p(bob,cal)\}$	Premise
4.	$\{p(bud,coe)\}$	Premise
5.	$\{\neg p(x,y), \neg p(y,z), g(x,z)\}$	Premise
6.	$\{\neg p(bob,z), g(art,z)\}$	1, 5
7.	$\{g(art,cal)\}$	3, 6
8.	$\{\neg p(bud,z), g(art,z)\}$	2, 5
9.	$\{g(art,coe)\}$	4, 8

Then we resolve the clause on line 4 with this result to produce the clause on line 9, which is exactly what we set out to prove.

Example		
1.	$\{p(art,bob)\}$	Premise
2.	$\{p(art,bud)\}$	Premise
3.	$\{p(bob,cal)\}$	Premise
4.	$\{p(bud,coe)\}$	Premise
5.	$\{\neg p(x,y), \neg p(y,z), g(x,z)\}$	Premise
6.	$\{\neg p(bob,z), g(art,z)\}$	1, 5
7.	$\{g(art,cal)\}$	3, 6
8.	$\{\neg p(bud,z), g(art,z)\}$	2, 5
9.	$\{g(art,coe)\}$	4, 8

One thing to notice about this derivation is that there are some useless dead-ends. We first tried resolving the fact about Art and Bob before getting around to trying the fact about Art and Bud. Resolution does not eliminate all search. However, at no time did we ever have to make an arbitrary assumption or an arbitrary choice of a binding for a variable. The absence of such arbitrary choices is why Resolution is so much more focussed than natural deduction systems like Fitch.

Example		
1.	$\{p(art,bob)\}$	Premise
2.	$\{p(art,bud)\}$	Premise
3.	$\{p(bob,cal)\}$	Premise
4.	$\{p(bud,coe)\}$	Premise
5.	$\{\neg p(x,y), \neg p(y,z), g(x,z)\}$	Premise
6.	$\{\neg p(bob,z), g(art,z)\}$	1, 5
7.	$\{g(art,cal)\}$	3, 6
8.	$\{\neg p(bud,z), g(art,z)\}$	2, 5
9.	$\{g(art,coe)\}$	4, 8

Another worthwhile observation about Herbrand Resolution is that, unlike Fitch, Herbrand Resolution frequently terminates even when there is no derivation of the desired result. Suppose, for example, we were interested in Cal is the grandparent of Art from the premises in this case. This sentence, of course, does not follow from the premises. And resolution is sound, so we would never generate this result. The interesting thing is that, in this case, the attempt to derive this result would eventually terminate.

Example		
1.	$\{p(art,bob)\}$	Premise
2.	$\{p(art,bud)\}$	Premise
3.	$\{p(bob,cal)\}$	Premise
4.	$\{p(bud,coe)\}$	Premise
5.	$\{\neg p(x,y), \neg p(y,z), g(x,z)\}$	Premise
6.	$\{\neg p(bob,z), g(art,z)\}$	1, 5
7.	$\{g(art,cal)\}$	3, 6
8.	$\{\neg p(bud,z), g(art,z)\}$	2, 5
9.	$\{g(art,coe)\}$	4, 8

With the premises given, there are a few more resolutions we could do, e.g. resolving the clause on line 1 with the second literal in the clause on line 5. However, having done these additional resolutions, we would find ourselves with nothing left to do; and, unlike Fitch, the process would terminate.

Resolution Not Generatively Complete

Using the Resolution Principle alone, it is not possible to generate every clause that is logically entailed by a set of premises.

Examples:

$$\{\} \models \{p(a), \neg p(a)\}$$

But resolution cannot generate these results.

Unfortunately, like Propositional Resolution, Herbrand Resolution is not generatively complete, i.e. it is not possible to find resolution derivations for all clauses that are logically entailed by a set of premise clauses. For example, the clause $\{p(a), \neg p(a)\}$ is always true, and so it is logically entailed by any set of premises, including the empty set of premises. Herbrand Resolution requires some premises to have any effect. Given an empty set of premises, we would not be able to derive any conclusions, including this valid clause.

Although Herbrand Resolution is not generatively complete, problems like this one are solved by negating the goal and demonstrating that the resulting set of sentences is unsatisfiable.

Unsatisfiability

Demonstrating Unsatisfiability

Start with premises.

Apply resolution repeatedly.

If empty clause generated, the original set is unsatisfiable.

The simplest use of resolution is in demonstrating unsatisfiability. In clausal form, a contradiction takes the form of the empty clause, which is equivalent to a disjunction of no literals. Thus, to automate the determination of unsatisfiability, all we need do is to use resolution to derive consequences from the set to be tested, terminating whenever the empty clause is generated.

Example

- | | | |
|----|-------------------------|---------|
| 1. | $\{p(a,b), q(a,c)\}$ | Premise |
| 2. | $\{\neg p(x,y), r(x)\}$ | Premise |
| 3. | $\{\neg q(x,y), r(x)\}$ | Premise |
| 4. | $\{\neg r(z)\}$ | Premise |

Let's start with a simple example. We have four premises, which are jointly unsatisfiable. $p(a,b) \wedge q(a,c) \wedge p(x,y) \Rightarrow r(x) \wedge q(x,y) \Rightarrow r(x) \wedge \neg r(z)$. The derivation of the empty clause in this case is easy.

Example

- | | | |
|----|-------------------------|---------|
| 1. | $\{p(a,b), q(a,c)\}$ | Premise |
| 2. | $\{\neg p(x,y), r(x)\}$ | Premise |
| 3. | $\{\neg q(x,y), r(x)\}$ | Premise |
| 4. | $\{\neg r(z)\}$ | Premise |
| 5. | $\{q(a,c), r(a)\}$ | 1, 2 |

We resolve the first clause with the second to get the clause shown on line 5.

Example

- | | | |
|----|-------------------------|---------|
| 1. | $\{p(a,b), q(a,c)\}$ | Premise |
| 2. | $\{\neg p(x,y), r(x)\}$ | Premise |
| 3. | $\{\neg q(x,y), r(x)\}$ | Premise |
| 4. | $\{\neg r(z)\}$ | Premise |
| 5. | $\{q(a,c), r(a)\}$ | 1, 2 |
| 6. | $\{r(a)\}$ | 5, 3 |

Next, we resolve the result with the third clause to get the unit clause on line 6. Note that $r(a)$ is the remaining literal from clause 3 after the resolution, and $r(a)$ is also the remaining literal from clause 5 after the resolution. Since these two literals are identical, they appear only once in the result.

Example

1.	$\{p(a,b), q(a,c)\}$	Premise
2.	$\{\neg p(x,y), r(x)\}$	Premise
3.	$\{\neg q(x,y), r(x)\}$	Premise
4.	$\{\neg r(z)\}$	Premise
5.	$\{q(a,c), r(a)\}$	1, 2
6.	$\{r(a)\}$	5, 3
7.	$\{\}$	6, 4

Finally, we resolve this result with the clause on line 4 to produce the empty clause.

Logical Entailment

Provability

A *resolution derivation* of a clause φ from a set Δ of clauses is a sequence of clauses ending in φ in which each item is
(1) a member of Δ or
(2) the result of applying the resolution to earlier items.

A sentence φ is *provable* from a set of sentences Δ by resolution if and only if there is a derivation of the empty clause from the clausal form of $\Delta \cup \{\neg\varphi\}$.

A resolution *proof* is a derivation of the empty clause from the clausal form of the premises and the negation of the desired conclusion.

Example

Everybody loves somebody. Everybody loves a lover.
Show that everybody loves everybody.

$\forall x. \exists y. \text{loves}(x, y)$
 $\forall u. \forall v. \forall w. (\text{loves}(v, w) \Rightarrow \text{loves}(u, v))$
 $\neg \forall x. \forall y. \text{loves}(x, y)$

$\{\text{loves}(x, f(x))\}$

$\{\neg \text{loves}(v, w), \text{loves}(u, v)\}$

$\{\neg \text{loves}(\text{jack}, \text{jill})\}$

As an example of using Herbrand Resolution to determine logical entailment, let's consider a case we saw earlier. The premises are shown below. We know that everybody loves somebody and everybody loves a lover.

Our goal is to show that everybody loves everybody. So, we negate this conclusion and add it to our set of premises.

Finally, we convert to clausal form, leading to the clauses shown below. Note the use of a Skolem function in the first clause and the use of Skolem constants in the clause derived from the negated goal.

Example

- | | | |
|----|---|--------------|
| 1. | $\{\text{loves}(x, f(x))\}$ | Premise |
| 2. | $\{\neg \text{loves}(v, w), \text{loves}(u, v)\}$ | Premise |
| 3. | $\{\neg \text{loves}(\text{jack}, \text{jill})\}$ | Negated Goal |

In order to prove our result, we start with our premises.

Example (continued)

- | | | |
|----|---|--------------|
| 1. | $\{\text{loves}(x, f(x))\}$ | Premise |
| 2. | $\{\neg \text{loves}(v, w), \text{loves}(u, v)\}$ | Premise |
| 3. | $\{\neg \text{loves}(\text{jack}, \text{jill})\}$ | Negated Goal |
| 4. | $\{\text{loves}(u, x)\}$ | 1, 2 |

We resolve the clause on line 1 with the clause on line 2 to produce the clause on line 4.

Example (concluded)

- | | | |
|----|---|--------------|
| 1. | $\{ \text{loves}(x, f(x)) \}$ | Premise |
| 2. | $\{ \neg \text{loves}(v, w), \text{loves}(u, v) \}$ | Premise |
| 3. | $\{ \neg \text{loves}(\text{jack}, \text{jill}) \}$ | Negated Goal |
| 4. | $\{ \text{loves}(u, x) \}$ | 1, 2 |
| 5. | $\{ \}$ | 4, 3 |

Finally, we resolve the clause on line 4 with the clause on line 3 to produce the empty clause and thus prove our result.

Inferential Equivalence

Equivalence Theorem: It is possible to prove a conclusion from a set of premises using Resolution if and only if it is possible to prove that conclusion from those premises using Fitch (without domain closure or induction).

This example shows that resolution is much more focussed than Fitch. The added good news is that it is every bit as powerful. In fact it is possible to derive a conclusion from a set of premises using resolution if and only if it is possible to derive that conclusion using Fitch.

