Logic Programming

Logic programming grew out of work on automatic theorem proving.

It was noticed that by trying to prove a carefully chosen predicate, a proof could provide the solution to a computational problem.

1 First Order Predicate Calculus

In the first order predicate calculus, quantifiers (i.e. $\exists$ and $\forall$) can be applied to variables but not functions or predicates. Also, predicates can't occur in the arguments of other predicates.

In the second order predicate calculus, predicates can take first order predicates as arguments.

Prolog and other logic programming languages are based on the first order predicate calculus.

1.1 Syntax

The syntax of the predicate calculus is:

```
const       → identifier
var         → identifier
fn          → identifier
pred        → identifier
term        → const | var | fn(termlist)
termlist    → term | term, termlist
atom        → pred | pred(termlist)
connectives → ∨ | ∧
quantifier  → (∃ var) | (∀ var)
formula     → atom |¬(formula)
            | quantifier(formula)
            | (formula) connective (formula)
```

Notes on Syntax:

1. In the predicate calculus, you can tell from the form of an identifier whether it is a constant (`const`), a variable (`var`), a function symbol (`fn`) or a predicate symbol (`pred`).

   Predicate names occur only at the top level, and function names never occur at the top level. That is, function names only occur within the arguments of a predicate, and predicate names never occur within an argument of another predicate.

2. In the predicate calculus, the names of function and predicate symbols indicate the number of arguments.

3. Sometimes `const` is not defined, and `fn`s with zero arguments are used instead. You may wish to think of constants as functions with no arguments.

The term *well-formed formula* or *WFF* is used to mean a syntactically correct formula.

The term *literal* is used to mean either an `atom` or the negation of an `atom`.

We may use $\rightarrow$ and $\leftrightarrow$ as connectives. It is understood that $p \rightarrow q$ is just shorthand for $q \vee \neg p$ and $p \leftrightarrow q$ is shorthand for $(p \rightarrow q) \wedge (q \rightarrow p)$.

The propositional calculus is the predicate calculus restricted to predicates that have no arguments. Hence there are no functions or variables, and hence no need for quantifiers.

We will skip parentheses where this won't lead to confusion. The unary operator $\neg$ binds more tightly than $\wedge$ which in turn binds more tightly than $\vee$.

The scope of a quantifier extends as far to the right as possible.

For example,

$$\neg\exists X \neg\forall Y \neg p(X,Y) \vee p(Y,X) \equiv \neg\big(\exists X\big(\neg\big(\forall Y\big(\big(\neg p(X,Y)\big) \vee p(Y,X)\big)\big)\big)\big)$$

and

$$\exists X \forall Y\, p(X,Y) \vee p(Y,X) \wedge q(X,X) \equiv \exists X\big(\forall Y\big(p(X,Y) \vee \big(p(Y,X) \wedge q(X,X)\big)\big)\big)$$

Notes on Prolog Syntax:

1. Prolog doesn't really distinguish between predicates and functions.

   We will keep this distinction while talking about the predicate calculus. However, like Prolog, we will usually use all lower case names for both.

2. Prolog will allow the same name to be used with different numbers of arguments, but considers these to be different. We will use this convention. For example, `f(X,Y)` and `f(Z)` are both terms, but the `f` in the first of these is completely different from the `f` in the second.

3. Prolog uses upper case letters for variables. We will do the same.

4. Warning: Prolog uses the term `atom` to mean a `const` that is not an `integer`. The definition of an `atom` in the predicate calculus is a formula that contains no smaller subformulas. These definitions are not compatible!

1.2 Free and Bound Variables

The *free variables* of a `formula` are defined as follows:

1. All variables in an `atom` are free.
2. The free variables of $\neg$`(`$\mathcal{F}$`)`, for any `formula`, $\mathcal{F}$, are the free variables of $\mathcal{F}$.
3. The free variables of `(`$\mathcal{F}$`)connective(`$\mathcal{G}$`)` are the free variables of $\mathcal{F}$ and the free variables of $\mathcal{G}$.
4. The free variables of `(`$\exists X$`)(`$\mathcal{F}$`)` or `(`$\forall X$`)(`$\mathcal{F}$`)`, for any `var`, X, are the free variables of $\mathcal{F}$ except X (if X happens to be a free variable of $\mathcal{F}$).

An occurrence of X in $\mathcal{G}$ is said to be a *bound occurrence* if it is an occurrence within a subformula of the form $(\exists X)(\mathcal{F})$ or $(\forall X)(\mathcal{F})$. Any other occurrence of X in $\mathcal{G}$ is a *free occurrence*.

A formula is *closed* if it contains no free variables.

From now on we will consider only closed formula (except when we are talking about a subformula of a closed formula).

1.3 Interpretations, Validity and Satisfiability

An *interpretation* I of a WFF consists of

1. a nonempty set U, called the *universe*, which is the set of values that variables may take,
2. for each constant c an element $c_I \in U$, which will be the value of c,
3. for each function symbol f taking n arguments a function $f_I :: U^n \to U$, and
4. for each predicate symbol p taking n arguments an n-ary relation p_I (i.e. a subset of U^n).

For a given interpretation, a closed WFF is either true of false.

1.3.1 Validity and Satisfiability

A WFF is *valid* iff it is true for every possible interpretation.

A WFF is *satisfiable* iff it is true for some interpretation.

Hence, a WFF is not valid if its negation is satisfiable. Similarly, a WFF is not satisfiable if its negation is valid.

1.3.2 Decidability of Validity

The validity of WFFs in the first order predicates calculus is not decidable, but it is partially decidable. This means

1. There does not exist an algorithm that can always determine whether or not a WFF is valid.
2. However, there does exist an algorithm that, when given a valid WFF, can always verify that the WFF is valid. That is, if a WFF is valid, it is possible to construct a proof for it.

If the algorithm that can verify (prove) any valid WFF is given an invalid WFF, it must fail to terminate in some cases.

Of course, this means that we can't put an upper bound on the amount of time that might be required to verify a WFF.

We can prove that a WFF is unsatisfiable by verifying that its negation is valid.

1.4 Clausal Form

A *clause* is a disjunction of literals.

Recall that:

- A *literal* is either an `atom` or the negation of an `atom`.
- An `atom` is a `pred`, optionally with arguments.
- A disjunction is a collection of literals combined with the $\vee$ connective.

In other words, a clause is a WFF with no $\wedge$ connectives, no quantifiers and with negation applied only to atoms.

Implicitly, all variables in a clause are universally quantified.

A WFF in *clausal form* is a conjunction of clauses.

We will be primarily interested in the unsatisfiability of clauses.

Given any WFF, F, it is possible to construct a WFF, G, in clausal form such that G is satisfiable iff F is satisfiable. (Hence, F is unsatisfiable iff G is. However, when F is valid, G may not be valid.)

We will use a slightly different notation for a WFF in clausal form.

In particular, each clause will be represented as a set of literals. Implicitly, the set will denote the disjunction of its elements.

A WFF is a collection of clauses (technically a set of clauses, but generally represented as a list of clauses).

At this point in these notes, we are going to make a change in the coloring convention. From now on, we will use *this color with the math font for the predicate calculus* and `another color with the typewriter font for Prolog code`.

1.4.1 Skolemization

An interpretation assigns a value in the universe to each constant. A WFF such as

$$\ldots \exists X \, \forall Y \; p(f(X, Y), Y, X) \ldots$$

is satisfiable if and only if

$$\ldots \forall Y \; p(f(a, Y), Y, a) \ldots$$

is satisfiable, where a is an unused name for a constant. This is because the WFF is satisfiable iff there *exists* an interpretation, which assigns a value to a (among other things), that makes the WFF true.

Note: The bottom WFF (with the a) need not necessarily be valid, even if the top WFF (with the X) is valid.

Similarly,

$$\ldots \forall W \; \forall X \; \forall Y \; \exists Z \; p(W, X, Y, Z) \ldots$$

can be changed to

$$\ldots \forall W \; \forall X \; \forall Y \; p(W, X, Y, g(W, X, Y)) \ldots$$

where g is an unused name for a function and W, X and Y are the universally quantified variables that may determine the value of Z. That is, W, X and Y are universally quantified variables that where in scope at the point where Z was existentially quantified.

This process for removing existential quantifiers is called *Skolemization*, after the logician Skolem. A constant that replaces an existentially quantified variable is called a *Skolem constant* and a function that is used in replacing a variable is called a *Skolem function*.

As another example, to remove quantifiers from

$$\exists W\ \forall X\ \forall Y\ \exists Z\ p(W, X, Y, Z)$$

we need both a Skolem constant and a Skolem variable, giving

$$\forall X\ \forall Y\ p(a, X, Y, g(X, Y))$$

In general, Skolemization preserves satisfiability but not necessarily validity.

If a WFF is satisfiable, then there exists an interpretation in which it is true. By extending the interpretation for the original WFF by choosing appropriate values for the Skolem constants and functions that replace existentially quantified variables when the WFF is Skolemized, it is possible to construct an interpretation for the Skolemized version of the WFF that is also true.

However, there are more possible interpretations for the Skolemized version of a WFF, since the interpretations include values for the Skolem constants and the Skolem functions.

When the original WFF is valid, the Skolemized version of the WFF, while satisfiable, may not be valid. This is because the Skolemized version may be false for certain choices of the Skolem constants and functions.

For example,

$$\neg p(z) \vee p(s(z)) \vee \exists X(p(X) \wedge \neg p(s(X)))$$

is valid, and hence satisfiable, while the Skolemized version,

$$\neg p(z) \vee p(s(z)) \vee (p(a) \wedge \neg p(s(a))),$$

while still satisfiable, is no longer valid.

For any interpretation of the original WFF, clearly selecting X to be z makes the top WFF true, so the WFF is valid.

For the Skolemized WFF, consider an interpretation where the universe is the set of natural number, z is 0, s is the successor function that adds 1 to its argument, and p is true if and only if its argument is zero. Now let a be 1. The WFF is false for this interpretation, and hence invalid.

1.4.2 Converting to Clausal Form

We proceed through several steps in converting a WFF to clausal form.

1. Move all negation inward, so only `atom`s are negated.

   We do this using the following rules, which you all remember from MACM 101.

   $$\neg(a \wedge b) = \neg a \vee \neg b$$
   $$\neg(a \vee b) = \neg a \wedge \neg b$$
   $$\neg\neg a = a$$
   $$\neg\forall X\ F = \exists X\ \neg F$$
   $$\neg\exists X\ F = \forall X\ \neg F$$

2. Rename variables so that each quantifier has its own unique variable associated with it.

3. Eliminate existential quantifiers ($\exists$s) by Skolemizing the WFF.

4. Convert the WFF to prenex normal form. A WFF is in *prenex normal form* if it consists of a string of quantifiers followed by a quantifier-free formula.

   Since we have previously removed existential quantifiers and all possible name conflicts, we can safely move all the remaining ($\forall$) quantifiers to the left without altering the meaning of the WFF.

5. Remove all quantifiers. Since all variables are now universally quantified, we can make the quantification implicit.

6. Convert the WFF to conjunctive normal form using the following laws.

   $$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$$
   $$(a \wedge b) \vee c = (a \vee c) \wedge (b \vee c)$$

7. Rewrite the WFF as a list of clauses.

1.4.3 Example

$$\forall X\Big(\neg p(X) \rightarrow \exists Y\Big(d(X,Y) \wedge \neg\Big(e(Y,f(X)) \vee e(Y,X)\Big)\Big)\Big) \wedge \neg\forall X\, p(X)$$

0. Get rid of the unofficial connective.

$$\forall X\Big(\neg\neg p(X) \vee \exists Y\Big(d(X,Y) \wedge \neg\Big(e(Y,f(X)) \vee e(Y,X)\Big)\Big)\Big) \wedge \neg\forall X\, p(X)$$

1. Move negation in as far as possible.

$$\forall X\Big(p(X) \vee \exists Y\Big(d(X,Y) \wedge \neg e(Y,f(X)) \wedge \neg e(Y,X)\Big)\Big) \wedge \exists X \neg p(X)$$

2. Rename variables to avoid duplication.

$$\forall X\Big(p(X) \vee \exists Y\Big(d(X,Y) \wedge \neg e(Y,f(X)) \wedge \neg e(Y,X)\Big)\Big) \wedge \exists Z \neg p(Z)$$

3. Skolemize.

$$\forall X\Big(p(X) \vee \Big(d(X,g(X)) \wedge \neg e(g(X),f(X)) \wedge \neg e(g(X),X)\Big)\Big) \wedge \neg p(a)$$

4. Convert to prenex normal form.

$$\forall X\Big(\Big(p(X) \vee \Big(d(X,g(X)) \wedge \neg e(g(X),f(X)) \wedge \neg e(g(X),X)\Big)\Big) \wedge \neg p(a)\Big)$$

5. Eliminate quantifiers.

$$\Big(p(X) \vee \Big(d(X,g(X)) \wedge \neg e(g(X),f(X)) \wedge \neg e(g(X),X)\Big)\Big) \wedge \neg p(a)$$

Last WFF from the previous page.

$$\Big(p(X) \vee \Big(d(X, g(X)) \wedge \neg e(g(X), f(X)) \wedge \neg e(g(X), X)\Big)\Big) \wedge \neg p(a)$$

6. Convert to conjunctive normal form.

$$\Big(p(X) \vee d(X, g(X))\Big) \wedge \Big(p(X) \vee \neg e(g(X), f(X))\Big)$$
$$\wedge \Big(p(X) \vee \neg e(g(X), X)\Big) \wedge \neg p(a)$$

7. Finally, rewrite as a list of clauses.

$$\{p(X), d(X, g(X))\}$$
$$\{p(X), \neg e(g(X), f(X))\}$$
$$\{p(X), \neg e(g(X), X)\}$$
$$\{\neg p(a)\}$$

1.5 Resolution

Resolution is a rule of inference.

$$\frac{p,\ p \rightarrow q}{q} \qquad \text{(Modus ponens)}$$

$$\frac{(p \rightarrow q),\ (q \rightarrow r)}{p \rightarrow r} \qquad \text{(Syllogism)}$$

$$\frac{(p \vee q),\ (\neg p \vee r)}{q \vee r} \qquad \text{(Resolution)}$$

In fact, the first two rules of inference are just special cases or resolution.

Since $p \equiv p \vee false$ and $p \rightarrow q \equiv \neg p \vee q$, we can apply resolution to yield $false \vee q \equiv q$.

Similarly, $p \rightarrow q \equiv \neg p \vee q$ and $q \rightarrow r \equiv \neg q \vee r$ so resolution yields $\neg p \vee r \equiv p \rightarrow r$.

Usually resolution is applied to a pair of clauses, C_1 and C_2, in a set of clauses where the clause (set of literals) C_1 contains some literal, l, such that C_2 contains the negation, $\neg l$, of the literal. In this case, we say that the clauses, C_1 and C_2, *resolve*. The *resolvent* of these clauses is the clause produced by taking the union of the literals in C_1 and C_2 excluding l and $\neg l$. The resolvent is added to the clause set.

Quiz: What does the empty clause, that is the clause represented by the empty set, $\{\}$, mean?

A set of clauses is unsatisfiable if and only if the empty clause (which is false) can be produced through resolution.

We can prove that a WFF is valid by proving that its negation is unsatisfiable.

Let us consider an example. We will start by considering an axiom of the propositional calculus.

$$\Big(a \to (b \to c)\Big) \to \Big((a \to b) \to (a \to c)\Big)$$

To prove that this axiom is valid, using resolution, we will prove that its negation is unsatisfiable. We will also need to transform the negation to clausal form.

$$\neg\Big(\big(a \to (b \to c)\big) \to \big((a \to b) \to (a \to c)\big)\Big)$$

Definition of implication.

$$\neg\Big(\neg\big(a \to (b \to c)\big) \vee \big((a \to b) \to (a \to c)\big)\Big)$$

De Morgan's laws.

$$\Big(\neg\neg\big(a \to (b \to c)\big) \wedge \neg\big((a \to b) \to (a \to c)\big)\Big)$$

Eliminate double negation.

$$\Big(a \to (b \to c)\big) \wedge \neg\big((a \to b) \to (a \to c)\big)\Big)$$

Definition of implication, twice.

$$\Big(\neg a \vee (b \to c)\big) \wedge \neg\big(\neg(a \to b) \vee (a \to c)\big)\Big)$$

De Morgan's laws.

$$\Big(\neg a \vee (b \to c)\big) \wedge \big(\neg\neg(a \to b) \wedge \neg(a \to c)\big)\Big)$$

Eliminate double negation.

$$\Big(\neg a \vee (b \to c)\big) \wedge \big((a \to b) \wedge \neg(a \to c)\big)\Big)$$

Last WFF from the previous page.

$$\Big(\neg a \vee (b \rightarrow c)\Big) \wedge \Big((a \rightarrow b) \wedge \neg(a \rightarrow c)\Big)$$

Definition of implication, three times.

$$\Big(\neg a \vee (\neg b \vee c)\Big) \wedge \Big((\neg a \vee b) \wedge \neg(\neg a \vee c)\Big)$$

De Morgan's laws.

$$\Big(\neg a \vee (\neg b \vee c)\Big) \wedge \Big((\neg a \vee b) \wedge (\neg\neg a \wedge \neg c)\Big)$$

Eliminate double negation.

$$\Big(\neg a \vee (\neg b \vee c)\Big) \wedge \Big((\neg a \vee b) \wedge (a \wedge \neg c)\Big)$$

Getting rid of unneeded parentheses.

$$\Big(\neg a \vee \neg b \vee c\Big) \wedge (\neg a \vee b) \wedge a \wedge \neg c$$

Finally, rewrite as a list of clauses.

$$\{\neg a, \neg b, c\}$$
$$\{\neg a, b\}$$
$$\{a\}$$
$$\{\neg c\}$$

We will now complete this proof using resolution, where the first four clauses are from the previous page.

1. $\{\neg a, \neg b, c\}$
2. $\{\neg a, b\}$
3. $\{a\}$
4. $\{\neg c\}$
5. $\{\neg a, c\}$ from 1 and 2.
6. $\{c\}$ from 3 and 5.
7. $\{\}$ from 4 and 6.

The final clause shows that our negated axiom is unsatisfiable, so the axiom itself is valid. Of course, the axiom is not required if resolution is a permitted rule of inference.

1.6 Unification

1.6.1 Substitution

Let t be any `term`, X be any `var` and C be any clause. By $C[t/X]$ we mean the result of replacing every occurrence of X in C with t.

If C is a clause in a clause set S, then $C \rightarrow C[t/X]$ so $S \Leftrightarrow S \cup \{C[t/X]\}$.

We will call the rule of inference that allows us to do this the *particularization rule* .

Suppose we are given the following axioms or assumptions in the form of a collection of clauses.

$$\{\texttt{add}(\texttt{zero}, W, W)\}$$
$$\{\texttt{add}(\texttt{succ}(X), Y, \texttt{succ}(Z)), \neg\texttt{add}(X, Y, Z)\}$$

If we wish to show that there are values of U and V for which

$$\texttt{add}(U, \texttt{succ}(V), \texttt{succ}(\texttt{zero}))$$

is true, then we can do so by showing that

$$\{\neg\texttt{add}(U, \texttt{succ}(V), \texttt{succ}(\texttt{zero}))\}$$

together with the two given clauses is unsatisfiable.

Note: The scope of a variable is a single clause. I could have used X in place of W, for example, in the first clause above. To avoid confusion I am making all variable names distinct.

This gives us the following collection of clauses.

$$\{\texttt{add}(\texttt{zero}, W, W)\}$$
$$\{\texttt{add}(\texttt{succ}(X), Y, \texttt{succ}(Z)), \neg\texttt{add}(X, Y, Z)\}$$
$$\{\neg\texttt{add}(U, \texttt{succ}(V), \texttt{succ}(\texttt{zero}))\}$$

In order to use resolution we need a pair of clauses such that one clause contains some literal, l, and the other clause contains the literal, $\neg l$. We don't have this. Furthermore, applying the particularization rule to a single clause will not give us this condition.

However, we can use the particularization rule a total of three times on two literals.

The collection of clauses from the previous page is:

$$\{\texttt{add}(\texttt{zero}, W, W)\}$$
$$\{\texttt{add}(\texttt{succ}(X), Y, \texttt{succ}(Z)), \neg\texttt{add}(X, Y, Z)\}$$
$$\{\neg\texttt{add}(U, \texttt{succ}(V), \texttt{succ}(\texttt{zero}))\}$$

If we make the following substitutions to the first and third clauses

$$\{\texttt{add}(\texttt{zero}, W, W)[\texttt{succ}(\texttt{zero})/W]\}$$
$$\{\neg\texttt{add}(U, \texttt{succ}(V), \texttt{succ}(\texttt{zero}))[\texttt{zero}/U][\texttt{zero}/V]\}$$

then the resulting clauses are

$$\{\texttt{add}(\texttt{zero}, \texttt{succ}(\texttt{zero}), \texttt{succ}(\texttt{zero}))\}$$
$$\{\neg\texttt{add}(\texttt{zero}, \texttt{succ}(\texttt{zero}), \texttt{succ}(\texttt{zero}))\}$$

which by resolution give us the empty clause, which means false. Hence, the clause set is unsatisfiable, so the original WFF,

$$\texttt{add}(U, \texttt{succ}(V), \texttt{succ}(\texttt{zero}))$$

is true for some values of U and V. Furthermore, the substitutions produce suitable values for U and V.

Remember that when we write a WFF in clausal form, all variables are implicitly universally quantified. Hence,

$$\forall U\ \forall V\ \neg \mathtt{add}(U, \mathtt{succ}(V), \mathtt{succ}(\mathtt{zero}))$$

is the same as of

$$\neg \exists U\ \exists V\ \mathtt{add}(U, \mathtt{succ}(V), \mathtt{succ}(\mathtt{zero}))$$

which is the negation of

$$\exists U\ \exists V\ \mathtt{add}(U, \mathtt{succ}(V), \mathtt{succ}(\mathtt{zero}))$$

which is what we really wanted to show was true.

What we have show to be unsatisfiable is the following set of clauses, with implicit quantifiers given explicitly.

$$\forall W\ \{\texttt{add}(\texttt{zero}, W, W)\}$$
$$\forall X\ \forall Y\ \forall Z\ \{\texttt{add}(\texttt{succ}(X), Y, \texttt{succ}(Z)), \neg\texttt{add}(X, Y, Z)\}$$
$$\forall U\ \forall V\ \{\neg\texttt{add}(U, \texttt{succ}(V), \texttt{succ}(\texttt{zero}))\}$$

or

$$\forall W\ \texttt{add}(\texttt{zero}, W, W)$$
$$\forall X\ \forall Y\ \forall Z\ (\texttt{add}(X, Y, Z) \rightarrow \texttt{add}(\texttt{succ}(X), Y, \texttt{succ}(Z)))$$
$$\forall U\ \forall V\ \neg\texttt{add}(U, \texttt{succ}(V), \texttt{succ}(\texttt{zero}))$$

That is, for any `add`, `zero` and `succ` that satisfy the first two conditions, there must be some values U and V such that $\texttt{add}(U, \texttt{succ}(V), \texttt{succ}(\texttt{zero}))$.

Finding a substitution that makes two atoms the same is called *unification* and the substitution is called a *unifier* .

We will write either

$$[\texttt{succ}(\texttt{zero})/W; \texttt{zero}/U; \texttt{zero}/V]$$

or

$$[\texttt{succ}(\texttt{zero})/W]; [\texttt{zero}/U]; [\texttt{zero}/V]$$

for the composition of the substitutions

$$[\texttt{succ}(\texttt{zero})/W], [\texttt{zero}/U] \text{ and } [\texttt{zero}/V].$$

A unifier σ for two atoms is a *most general unifier* if, for any other unifier τ, there exists a substitution ω such that $\tau = \sigma; \omega$. Hence, a most general unifier is a "minimal" unifier, that changes as little as possible.

For example, given the atoms $d(f(Y), Y)$ and $d(X, Y)$, $\sigma = [f(Y)/X]$ is a most general unifier. The substitution $\tau = [f(Y)/X; g(a)/Y]$ is another unifier, since

$$d(f(Y), Y)[f(Y)/X; g(a)/Y] = d(f(g(a)), g(a))$$

$$d(X, Y)[f(Y)/X; g(a)/Y] = d(f(g(a)), g(a))$$

In this case, $\tau = \sigma; \omega$ where $\omega = [g(a)/Y]$.

We note that the order of the components in a substitution is important.

$$d(X, Y)[g(a)/Y; f(Y)/X] = d(f(Y), g(a))$$

On the other hand $\tau = [f(Y)/X; g(a)/Y] = [g(a)/Y; f(g(a))/X]$.

Finally, we note that $\sigma' = [f(Z)/X, Z/Y]$ is another most general unifier for the two atoms considered above, so most general unifiers need not be unique.

1.6.2 The Unification Algorithm

There is a simple algorithm to determine whether two atoms can be unified, and to produce a most general unifier if unification is possible.

We assume that the algorithm is given copies of two atoms.

Recall that each variable is local to the clause in which it occurs. That is, for example, the variable X in one clause is different than the variable X in another clause. We will also assume that at the start of each unification step, no two clauses have any variable name in common. It is always possible to make this so by changing the names of variables.

1. Initialize $\sigma = [\,]$.
2. Compare the atoms from left to right (e.g. traverse the expression tree with a preorder traversal) until a difference is found or the atoms are found to be identical.
   (a) If a difference is found in a place where one atom contains a variable, say X, and the other contains a term, t, where t does not contain X, then
      i. Replace σ with $\sigma; [t/X]$
      ii. Apply the substitution $[t/X]$ to both atoms.
      iii. Continue with the preorder traversal of the expression trees of the two atoms as modified by the substitution.

      (Note: The term t may be just a different variable.)
   (b) If any other difference is found, the unification fails.
   (c) If no differences remain, then unification has succeeded and σ is the most general unifier.

1.7 Resolution Theorem Proving

A resolution theorem prover can be used to show that a collection of clauses is unsatisfiable.

In normal use, we usually want to show that a particular consequence follows from a set of assumptions.

We proceed by adding the negation of the consequence to our assumptions and prove that the resulting set of clauses is unsatisfiable. That is, the negation of the consequence cannot be true, so the original (unnegated) consequence must follow from the assumptions, unless our assumptions are unsatisfiable.

As we have seen in an earlier example on page 37 of these notes, the substitutions used when unifying atoms in clauses can produce useful information.

A resolution theorem prover starts with a collection of clauses and repeatedly adds new clauses to the set one at a time, using the particularization rule and resolution, until the empty clause is generated.

The one remaining issue is the order in which new clauses are generated. This, in fact, has been a major focus of research into automatic theorem proving.

1.8 Horn Clauses

In general, some of the atoms in a clause will be negated and others will not be.

Consider a clause of the form

$$\{A_1, A_2, \ldots, A_m, \neg B_1, \neg B_2, \ldots, \neg B_n\}.$$

This may be written in nonclausal form as

$$B_1 \wedge B_2 \wedge \cdots B_n \rightarrow A_1 \vee A_2 \vee \cdots \vee A_m.$$

A *Horn clause* is a clause that contains at most one atom that is not negated. We will restrict our attention to Horn clauses that contain exactly one atom that is not negated. (Some books define a Horn clause to have exactly, rather than at most, one atom that is not negated.)

In the above example, the clause is a Horn clause if $m = 1$.

We will write Horn clauses in the following form:

$$A_1 \leftarrow B_1, B_2, \ldots B_n.$$

A special case of a Horn clause is when $n = 0$. In this case we will write

$$A_1.$$

rather than

$$A_1 \leftarrow .$$

2 Prolog

A Prolog program is a collection of Horn clauses.

For example, the following is a listing of the file `nat.pl`.

```
add(zero,W,W).
add(succ(X),Y,succ(Z)) :- add(X,Y,Z).
```

Remember that A `:-` B_1`,`B_2`,`B_3`.` means $A \vee \neg B_1 \vee \neg B_2 \vee \neg B_3$.

A is called the *head* of the clause and B_1`,`B_2`,`B_3 is called the *body*.

A clause with only a head is called a *fact* and a clause with a body is called a *rule*.

Important: Notice that every line ends with a period (`.`).

2.1 Using Prolog on CSIL Linux Machines

```
6: burton_apple% pl
  • • •
For help, use ?- help(Topic). or ?- apropos(Word).

?- [nat].
% nat compiled 0.00 sec, 580 bytes

Yes
?- add(U,succ(V),succ(zero)).

U = zero
V = zero
```

2.2 Queries and Goals

A *query* , such as

```
?- add(U,succ(V),succ(zero)).
```

is entered interactively. This becomes an initial *goal* .

From the viewpoint of the predicate calculus, a query, Q, entered after the `?-` prompt, is the body of a clause without a head. That is, the query is the clause $\{\neg Q\}$.

With the implicit universal quantification, this is the WFF $(\forall \text{ local variables})(\neg Q)$. Prolog attempts to prove that $\neg Q$ together with the current rules and facts is unsatisfiable.

Of course, $(\forall \text{ local variables})(\neg Q)$ is equivalent to $\neg(\exists \text{ local variables})(Q)$.

With the example goal from the previous page, which was

```
?- add(U,succ(V),succ(zero)).
```

and the contents of `nat.pl`, Prolog determines that no `const zero`, `fn succ`, and `pred add` can satisfy our negated goal as well as our rules and facts. Prolog also finds specific values of `U` and `V` that violate the negated goal

$$\neg(\exists\, \texttt{U}\, \exists\, \texttt{V})(\texttt{add(U,succ(V),succ(zero))})$$

These values of `U` and `V` will then satisfy the original (unnegated) goal.

2.3 Search Straregy in Prolog

With resolution theorem proving, a major concern is the order in which new clauses should be generated from existing clauses using resolution.

Prolog uses a depth first search with backtracking. Clauses are tried in order from top to bottom, and the first clause with a head that can be unified with the current goal is used. The predicates in the clause body are taken as new goals and are considered from left to right. If each goal succeeds (from the Prolog viewpoint) then the original query succeeds (from the Prolog viewpoint). If a goal cannot succeed (from the Prolog viewpoint) then backtracking occurs and we try selecting another clause at the last point where we had more than one possible clause to choose from.

2.4 Other Aspects of Prolog

You need a few other bits of information about Prolog in order to do much that is useful.

A constant is either a number or what prolog calls an *atom*. A Prolog atom is usually a word starting with a lower case letter. It is possible to have atoms that start with upper case letters or other symbols if they are enclosed in single quotes.

```
?- X = name.

X = name

Yes
?- X = 'name'.

X = name

Yes
?- X = 'Name'.

X = 'Name'

Yes
```

Lists are somewhat similar to lists in Haskell.

In Prolog, `[ ]` and `[1,2,3]`, for example, are lists of length zero and three.

If `L` is a list, then `[X|L]` is the list produced by inserting `X` at the start of `L` to produce a new list, while leaving `L` unchanged.

For example, with `list.pl` as follows

```
isEqualTo(X,X).
cons(X,Y,Z) :- isEqualTo([X|Y],Z).
```

we get

```
| ?- [list].
  • • •
yes
| ?- cons(a,[2,3],X).
X = [a,2,3]
yes
| ?- cons([1,2,3],[4,5,6],X).
X = [[1,2,3],4,5,6]
yes
```

Double quotes are used as a special syntax for a list of ascii character codes.

```
?- X = "Hello".

X = [72, 101, 108, 108, 111]
```

Beware of the following:

```
?- X = ['H'|"ello"].

X = ['H', 101, 108, 108, 111]
```

A useful web page, with lots of links, including some to tutorials, may be found at:

```
http://www.cetus-links.org/oo_prolog.html
```

The following also may be useful:

```
http://www.amzi.com/AdventureInProlog/advfrtop.htm
```

I would like to thank Tai Meng for many corrections and suggestions for improvements to these notes. Of course, the remaining errors are my fault.