

COMP4415 Lecture Notes on Logic Programming

1

Michael Maher

May 11, 2009

Logic programming starts with the Horn clause subset of clausal reasoning. This subset is sufficiently expressive for many tasks and, with some restrictions on how resolution can be applied, has efficient implementations and is controllable enough to be used as a programming language. Various extensions of the Horn clause base have been developed for different purposes, and we will look at some of them in this and following lectures. Prolog is the best-known logic programming language. It keeps quite close to the Horn clause base.

1 Notation and Terminology

I will try to conform to the notation and terminology of the textbook. Please let me know when I vary from it. Most of the definitions here are just extensions or reminders of definitions in the textbook.

A clause

$$\{A_1, A_2, \dots, A_n, \neg B_1, \neg B_2, \dots, \neg B_m\}$$

is an abbreviation for the formula

$$A_1 \vee A_2 \vee \dots \vee A_n \vee \neg B_1 \vee \neg B_2 \vee \dots \vee \neg B_m$$

It also can be written as

$$(A_1 \vee A_2 \vee \dots \vee A_n) \leftarrow (B_1 \wedge B_2 \wedge \dots \wedge B_m)$$

where no negation symbols appear. We will often use a simplified form of this, writing

$$A_1, A_2, \dots, A_n \leftarrow B_1, B_2, \dots, B_m.$$

When $m = 0$ the arrow is usually omitted.

There are several categories of clauses that are of interest. A clause is a:

Horn clause if $n \leq 1$. It is either a definite clause or a goal clause.

definite clause if $n = 1$, written $A_1 \leftarrow B_1, B_2, \dots, B_m$.

goal clause (or negative clause) if $n = 0$ written $\leftarrow B_1, B_2, \dots, B_m$.

positive clause if $m = 0$, written $A_1, A_2, \dots, A_n \leftarrow$.

fact if $m = 0$ and $n = 1$, written A_1 .

unit clause if $m + n = 1$, written A_1 or $\leftarrow B_1$.

empty clause if $m = 0$ and $n = 0$. Often denoted by \square .

A set of definite clauses is called a *program* (or *logic program*). We can consider a definite clause as a *rule*. If the rule is $A_1 \leftarrow B_1, B_2, \dots, B_m$ then we refer to A_1 as the *head* and B_1, B_2, \dots, B_m as the *body* of the rule. In logic programming languages, like Prolog, a program consists of a *sequence* of clauses and the order of clauses in the sequence can affect the computation. We won't be concerned much with those issues. Prolog has a specific syntax that does not conform with the way clauses have been written up till now. It distinguishes variables by beginning them with an upper case letter, and replaces \leftarrow by $:-$.

An example (Prolog) logic program. `[]` represents an empty list and `[H|T]` represents a non-empty list where H is the first element and T is the remainder.

```
% append(A, B, C)
% list C is equal to the concatenation of A and B

append([], Y, Y).
append([H|X], Y, [H|Z]) :-
    append(X, Y, Z).
```

Let Σ_F be the set of function (and constant) symbols. The *Herbrand universe* is the set of all variable-free terms. The *Herbrand base* \mathcal{H} is the set of all variable-free atoms. A *Herbrand interpretation* is a subset of the Herbrand base¹. $2^{\mathcal{H}}$ is the set of all subsets of \mathcal{H} , that is, the set of all Herbrand interpretations.

¹Strictly speaking, a Herbrand interpretation is an interpretation where the domain is the Herbrand universe and each function symbol is interpreted as the corresponding term constructor. The set of ground atoms true in the interpretation is the subset of the Herbrand base that is used to represent the Herbrand interpretation.

2 Function-based Semantics of Definite Clauses

Unit resolution is the use of resolution only when one of the clauses is a unit clause (that is, contains only a single literal). Unit resolution is not refutation complete in general, but it is refutation complete for Horn clauses. That is, if a set of Horn clauses is unsatisfiable, unit resolution can generate the empty clause. Let U_P be the set of ground atoms that can be derived from P by unit resolution (and instantiating clauses).

We can produce an equivalent algebraic formulation of U_P , using fixedpoint theory.

2.1 Fixedpoints

Let S be any set (especially, S can be \mathcal{H}).

Let $f : 2^S \rightarrow 2^S$ be a function that maps S to S (and particularly, Herbrand interpretations to Herbrand interpretations).

Definition 1 f is monotonic if, for all subsets I and J of S , if $I \subseteq J$ then $f(I) \subseteq f(J)$.

f is continuous if for every increasing sequence $I_1 \subseteq I_2 \subseteq \dots \subseteq I_j \subseteq \dots$

$$\bigcup_{i=1}^{\infty} f(I_i) = f(\bigcup_{i=1}^{\infty} I_i)$$

Monotonicity is a part of continuity.

Proposition 1 f is monotonic iff for every increasing sequence $\bigcup_{i=1}^{\infty} f(I_i) \subseteq f(\bigcup_{i=1}^{\infty} I_i)$.

The following proposition shows that continuity is a little bit like compactness.

Proposition 2 f is continuous iff f is monotonic and for every I and every $x \in f(I)$ there is a finite subset I' of I such that $x \in f(I')$.

Definition 2 We define the operator \uparrow as follows:

$$\begin{aligned} f \uparrow 0 &= \emptyset \\ f \uparrow (n+1) &= f(f \uparrow n) \\ f \uparrow \omega &= \bigcup_{n=1}^{\infty} f \uparrow n \end{aligned}$$

This sequence can be extended beyond ω using transfinite ordinals.

A subset X of S is a *fixedpoint* of f iff $f(X) = X$.

Theorem 1 *Suppose f is monotonic.*

Then f has a least fixedpoint $lfp(f)$. That is, $lfp(f)$ is a fixedpoint such that, for all fixedpoints X of f , $lfp(f) \subseteq X$.

$lfp(f)$ is also the least x such that $x \supseteq f(x)$.

f also has a greatest fixedpoint $gfp(f)$. $gfp(f)$ is also the greatest x such that $x \subseteq f(x)$.

If f is continuous then $lfp(f) = f \uparrow \omega$.

The x such that $x \subseteq f(x)$ are called *pre-fixedpoints*, and the x such that $x \supseteq f(x)$ are called *post-fixedpoints*.

The definitions and results in this section extend beyond sets of interpretations to a large class of partially ordered sets.

Definition 3 *A set X with a partial order \leq is a complete lattice if*

- *there is a largest element \top and a smallest element \perp of X*
- *for every (possibly infinite) subset Y of X , there is a least upper bound of Y in X (denoted by $\text{lub } Y$ or $\sqcup Y$) and a greatest lower bound of Y in X (denoted by $\text{glb } Y$ or $\sqcap Y$)*

A least upper bound of Y is $x \in X$ such that $\forall y \in Y \ y \leq x$ and if $\forall y \in Y \ y \leq z$ then $x \leq z$, and greatest lower bound has a similar definition.

2^S under the subset ordering forms a complete lattice. All the results above apply to any complete lattice, not just 2^S . For more details see the notes on fixedpoints by Carsten Fritz.

2.2 Function-based Semantics

Let P be a set of definite clauses. The following function is the basis for a functional form of the unit resolution.

Definition 4 *Given a set of definite clauses P , the function $T_P : 2^{\mathcal{H}} \rightarrow 2^{\mathcal{H}}$ is defined as*

$$T_P(I) = \{A \mid A \leftarrow B_1, \dots, B_n \text{ is a ground instance of } P \text{ and } \{B_1, \dots, B_n\} \subseteq I\}$$

It is not difficult to prove that T_P is continuous. The application of T_P to an interpretation I is like a batch application of unit resolution (using the atoms of I) to P , where $T_P(I)$ is the set of generated unit clauses.

3 Model-based Semantics

A *Herbrand model* of P is a Herbrand interpretation that is a model of P . That is, for every ground instance $A \leftarrow B_1, \dots, B_n$ of a clause of P , whenever $\{B_1, \dots, B_n\} \subseteq I$ then $A \in I$.

Proposition 3 *Let P be a set of definite clauses.*

I is a Herbrand model of P iff $T_P(I) \subseteq I$.

In fixedpoint terms, I is a Herbrand model of P iff I is a postfixpoint of T_P .

Proposition 4 *Let P be a set of Horn clauses. For any set of Herbrand models of P , the intersection of those models is also a model of P .*

We define $lm(P)$ to be the intersection of *all* Herbrand models of P . $lm(P)$ is itself a model of P – it is called the least (Herbrand) model of P .

As an easy consequence, we find that Horn clauses entail disjunctions of atoms only in trivial ways.

Proposition 5 *Let P be a set of Horn clauses and A_1 and A_2 be atoms. If $P \models A_1 \vee A_2$ then either $P \models A_1$ or $P \models A_2$*

4 SLD Resolution

A sequence of resolution steps is *linear* if the output of each resolution step is used as an input to the next resolution step. Such a sequence is *input* resolution if the other input comes from the original set of clauses (before any resolution has been done). Thus a linear input refutation of $\leftarrow A$ in a definite clause program P is a sequence of negative clauses D_i where D_0 is $\leftarrow A$ and D_{i+1} is obtained from D_i by resolution with a clause of P , and the final negative clause is the empty clause. This restriction of resolution is called *SLD resolution*. The sequence of steps is called a *derivation*. A derivation of the empty clause from $\leftarrow A$ is called a *successful* derivation for A wrt P . Let SLD_P be the set of ground atoms A such that there is a successful SLD derivation of A in P .

SLD resolution is sometimes called a “top-down” execution whereas unit resolution is called a “bottom-up” execution. If you draw a proof tree for a ground atom A , with the root at the top, then unit resolution works from the bottom and SLD resolution works from the top.

Also notice that SLD resolution is *goal-directed* – the possible resolution steps are restricted by the current negative clause in the sequence – whereas unit resolution is not. (The goal under discussion is to refute the negative clause.) For example, given the definite clauses

$$\begin{array}{l} a \leftarrow b \\ b \\ c \leftarrow d \\ d \end{array}$$

and the negative clause $\leftarrow a$, SLD resolution will not attempt to resolve the last two definite clauses because they never become relevant to the goal. On the other hand, unit resolution is not goal-directed and may resolve these two clauses.

(Goal-directedness is more important in first-order logic. Consider the definite clauses $\{a(X) \leftarrow b(X); b(0); b(1); b(2); \dots b(99)\}$ and the negative clause $\leftarrow a(22)$. SLD resolution will produce the negative clause $\leftarrow b(22)$ which will unify only with one definite clause. Unit resolution can generate the clauses $\{a(0); a(1); a(2); \dots a(99)\}$ before discovering that only one of these clauses is useful in finding a refutation.)

However, unit resolution can also avoid doing work that SLD resolution may not. For example, suppose we add $a \leftarrow e$ to the set of definite clauses. Then SLD resolution will resolve this clause with $\leftarrow a$, whereas unit resolution will not resolve on the extra clause.

4.1 SLD Finite Failure

When employing SLD resolution, there is a nominal *selection rule* (called *computation rule* by Ben-Ari) which chooses the atom in each input negative clause that is used to resolve against. In an SLD derivation, an atom is ignored if it is never selected by the selection rule. A selection rule is *fair* if no SLD-derivation using that rule ignores an atom. Fair SLD resolution is SLD resolution using a fair selection rule.

An SLD derivation *fails* if (a negative clause in) it contains a selected literal that cannot be resolved on by SLD resolution. If all SLD derivations from $\leftarrow A$ fail then there can be no refutations of $\leftarrow A$. If all derivations with fixed selection rule fail by a finite bound, then A is said to be *finitely failed* wrt P . If A is finitely failed wrt P using one selection rule then A is finitely failed wrt P for all fair selection rules.

One way to visualise SLD-resolution is a *SLD-tree*. In an SLD-tree, the initial goal clause is the root of the tree, and every node of the tree is a goal clause. The selection rule determines a literal in each goal clause, and there is a child in the

tree for each resolvent obtained by resolving on the selected literal with a clause of the program.

Finite failure enables us to conclude that there is no way to prove A from P . This property will be used next week.

5 Equivalence of Semantics

These different ways to give a meaning to sets of definite clauses are equivalent.

Theorem 2

$$SLD_P = U_P = lm(P) = lfp(T_P) = T_P \uparrow \omega$$

For Horn clauses, the two restricted forms of resolution – unit resolution and SLD resolution – are refutation complete. This does not hold for general clauses. See the section on refutation completeness, Section 7.

6 Constraint Logic Programming

6.1 Conditional resolution

One way of looking at first-order resolution is that each clause represents the set of all its ground instances (where variables can take values from the Herbrand universe). When doing first-order resolution, we use unification to keep track of the set of ground instances. But messing about with substitutions is complicated and error-prone, so here is an alternative formulation of the idea.

We consider that each clause has a condition which must be satisfied. The conditions will be equations, and we can write such *conditional clauses* in the form $L_1 \vee \dots \vee L_n \leftarrow e$. Initial clauses will have the condition *true* (or $x = x$ if you prefer). Each conditional clause represents the set of ground instances that also satisfy the equations. (So a clause with an unsatisfiable condition represents the empty set of ground instances and consequently can be ignored.)

It should be clear that we can resolve ground instances of $p(s) \vee C_1$ and $\neg p(t) \vee C_2$ if and only if the ground instances of s and t are equal. Rather than using unification and substitutions to express this requirement, we can present it as a condition: the resolvent is $C_1 \vee C_2 \leftarrow s = t$. The resolvent represents all the ground instances that can be obtained from ground instances of the given clauses by resolving on the two distinguished literals. In this way we separate the problem of representing all ground resolvents from the equation-solving (unification) problem. In practice, we will still need to use a unification algorithm to determine whether the resulting conditional clause can be ignored or not.

In general, when resolving the two conditional clauses $p(s) \vee C_1 \leftarrow e_1$ and $\neg p(t) \vee C_2 \leftarrow e_2$ we produce $C_1 \vee C_2 \leftarrow (e_1 \wedge e_2 \wedge s = t)$. It should be obvious that the resolvent represents all the ground clauses that can be obtained from ground instances of the resolvands by resolving on the distinguished literals ($p(s)$ and $\neg p(t)$). In this formulation, unification is simply an implementation technique and is not necessary to understand why first-order resolution works.

If resolution derives a conditional empty clause $\square \leftarrow e$ then each solution of e , applied to all the clauses in the derivation, describes a set of ground instances of the clauses that are unsatisfiable. This is the finite set of ground clauses guaranteed by Herbrand's theorem.

6.2 Conditional SLD resolution

Consider an SLD refutation from an initial goal clause $\leftarrow G$ and a program P that ends with a conditional empty clause $\square \leftarrow e$. We can view the goal clause as a query, asking for what values of its variables \tilde{x} does G hold as a consequence of P . The equations e define the values for which G holds as a result of this refutation (G might hold for other values as a result of other refutations).

The answers to such a query should only involve \tilde{x} , the variables of G , where possible. As long as e has a solution, the restrictions on the other variables are not meaningful to the original query. Thus we only want the effect of e on \tilde{x} , which can be expressed by existential quantification: $\exists_{-\tilde{x}} e$, where $\exists_{-\tilde{x}}$ expresses the existential quantification of all variables *except* \tilde{x} . We can simplify e by finding a solved form using unification, and then eliminating all those equations in the solved form with a quantified variable on the left side. This produces an answer that is as compact as possible. Call this *simplify*(\tilde{x}, e).

A conjunction of equations e to a query G is a *correct answer* if $P \models e \rightarrow G$. That is, in every model of P , e describes some situations where G holds. e is a *computed answer* for G if there is an SLD refutation for $P \cup \{\leftarrow G\}$ that ends with $\square \leftarrow e'$ and $e = \text{simplify}(\tilde{x}, e')$.

Proposition 6 Consider a program P , a query G , and a conjunction of equations e .

- If e is a computed answer then it is correct.
- If e is a correct answer then it can be computed by a SLD-refutation. (that is, there is a computed answer c such that $e \rightarrow \exists_{-\tilde{x}} c$.)

This says that (1) SLD refutations are a sound way of computing answers and (2) that SLD refutations will capture all correct answers. That is, SLD resolution is a sound and complete method for answering queries about a program.

6.3 Constrained resolution

In some circumstances we wish to use variables to range over non-Herbrand values (like numbers or strings) and apply operations and relations that are pre-defined for such values (like $+$, $<$, concatenation, and test for substring). Using the ideas of conditional resolution, we can perform resolution over clauses involving non-Herbrand values and non-Herbrand conditions. (What we lose is the power to use Skolemization and resolution together to determine the satisfiability of arbitrary first-order sentences.) We need to use a different universe for interpretations in place of the Herbrand universe and specify the meaning of new symbols like $+$ and $<$. This is done by a constraint domain.

A *signature* Σ is a set of function and relation symbols, each with an associated arity. It is the set of those symbols we want to have a specified meaning. We assume that the binary relation symbol $=$ is in Σ .

A *constraint domain* over a signature Σ is a pair $(\mathcal{D}, \mathcal{L})$ where \mathcal{D} is a Σ -structure and \mathcal{L} is a set of logical formulas over Σ and a set of variables. A *constraint* is a formula $c \in \mathcal{L}$. Thus \mathcal{L} specifies the syntax of constraints and \mathcal{D} specifies their semantics. We assume $=$ is interpreted as identity in \mathcal{D} and that \mathcal{L} is closed under variable renaming, conjunction, and existential quantification².

We can use a *many-sorted* logic (which is kind of a logic with a type system) if we want to use a logic that has more than one domain. We can also combine Herbrand terms and (say) integers in a larger domain to have a domain where the values are Herbrand terms generated from constants *and* the integers. But we will not look into these complications.

We can now have clauses like

$$p(x) \vee \neg p(y) \leftarrow x < y$$

which represents the infinite set of ground clauses

$$p(1) \vee \neg p(2); p(1) \vee \neg p(3); \dots; p(2) \vee \neg p(99); \dots$$

and we can resolve this clause with the clause $p(9)$ to obtain

$$p(x) \leftarrow x < 9$$

In general we need an algorithm to perform the same role as the unification algorithm in conditional resolution: to simplify the conditions and determine whether the conditions are satisfiable. This algorithm is called a *constraint solver*.

²As a result, the conjunction of constraints is also a constraint. When I refer to a constraint, remember that it might be a quite complicated conjunctive constraint.

6.4 Constraint Logic Programming

Constraint Logic Programming (CLP) consists of logic programming over a non-Herbrand domain. That means we use constrained SLD resolution and unit resolution in place of normal SLD resolution and unit resolution. Program clauses and queries may contain constraints as conditions. In place of the Herbrand universe we have the set of values that variables may take (determined by the constraint domain), and in place of the Herbrand base we have the set

$$\{p(\tilde{d}) \mid p \text{ is a predicate symbol, } \tilde{d} \text{ is a sequence of values}\}$$

Ground instances are obtained by replacing all variables by values and then evaluating any expressions. For example, the ground instances of $p(x + 1) \leftarrow p(x)$ are

$$p(1) \leftarrow p(0); p(2) \leftarrow p(1); \dots$$

As an example of CLP, consider a program for describing the compounding interest on a bank account.

```
%      D: Initial deposit
%      T: Time deposited in months
%      I: Fixed (but compounded) monthly interest rate
%      B: Balance at the end

compound(D, T, I, B) :-
    {T = 0,
     B = D}.
compound(D, T, I, B) :-
    {T >= 1,
     T1 = T - 1,
     A = D*(1 + I/100)},
    compound(A, T1, I, B).
```

The first rule says, if the money is deposited for 0 months then we will not earn any interest: the balance will equal the original deposit. The second rule says, if the money is deposited for 1 or more (T) months then after the first month the account will have A dollars, and we must compound that amount for $T - 1$ months.

Practically all of the results mentioned earlier for logic programming, also hold for CLP. The only exception is Proposition 6, part (2), which says that every correct answer is “covered” by one computed answer. This will not hold for some constraints.

Consider the following program, using a numerical constraint domain

$$\begin{aligned} p(x) &\leftarrow x \leq 5 \\ p(x) &\leftarrow x \geq 5 \end{aligned}$$

This program says (in a slightly indirect way) that p holds for every number. So a correct answer to the query $p(y)$ is *true*. However, the computed answers for the query are $y \leq 5$ and $y \geq 5$. In general, many SLD refutations are needed to “cover” a correct answer.

When Prolog was originally developed, computers could not support a powerful constraint solver. Instead the *is* predicate was introduced to do arithmetic evaluation. Even unification was omitted in favour of a more efficient but incomplete algorithm. Modern Prologs might support these “features” for backward compatibility but they now include arithmetic constraint solvers and correct unification.

7 Refutation Completeness

Refutation completeness is the property of a set of inference rules. It guarantees that any unsatisfiability in the initial set of clauses can be discovered by a refutation. Not all the inference rules we have looked at are refutation complete.

7.1 Factoring

Binary resolution (where we resolve on two literals in different clauses) is not refutation complete. Consider the clauses

$$p(x) \vee p(y); \neg p(z) \vee \neg p(w)$$

Binary resolution can only produce another clause with two literals. Thus it will never produce the empty clause.

We need to add the operation of *factoring* to binary resolution to make it refutation complete. A *factor* is obtained from a clause by unifying two literals (that are unifiable). If the clause is (say) $\neg p(s) \vee \neg p(t) \vee L_1 \vee \dots \vee L_n$ then unifying $\neg p(s)$ and $\neg p(t)$ gives us the mgu θ and the factor is $(\neg p(s) \vee L_1 \vee \dots \vee L_n)\theta$. In the example above, we can unify the two literals in the first clause (to give the factor $p(x)$) and similarly in the second clause we produce the factor $\neg p(z)$. These two clauses can then be resolved by binary resolution to produce the empty clause.

7.2 Unit and Input Resolution

Unit resolution is not refutation complete. Consider the following set S of clauses:

$$p \vee q; \neg p \vee q; p \vee \neg q; \neg p \vee \neg q$$

S is clearly unsatisfiable, but unit resolution cannot find a refutation because there are no unit clauses.

Similarly, input resolution cannot find a refutation because after resolving (say) the first two clauses the resultant clause is a unit clause, and resolving a unit clause with any of the input clauses must produce another unit clause. Thus input resolution will never produce an empty clause.

However, both unit resolution and input resolution (and SLD resolution) are refutation complete for Horn clauses.