Knowledge Representation and Reasoning

Ronald J. Brachman
Hector J. Levesque
In Praise of *Knowledge Representation and Reasoning*

This book clearly and concisely distills decades of work in AI on representing information in an efficient and general manner. The information is valuable not only for AI researchers, but also for people working on logical databases, XML, and the semantic web: read this book, and avoid reinventing the wheel!

**Henry Kautz, University of Washington**

Brachman and Levesque describe better than I have seen elsewhere, the range of formalisms between full first order logic at its most expressive and formalisms that compromise expressiveness for computation speed. Theirs are the most even-handed explanations I have seen.

**John McCarthy, Stanford University**

This textbook makes teaching my KR course much easier. It provides a solid foundation and starting point for further studies. While it does not (and cannot) cover all the topics that I tackle in an advanced course on KR, it provides the basics and the background assumptions behind KR research. Together with current research literature, it is the perfect choice for a graduate KR course.

**Bernhard Nebel, University of Freiburg**

This is a superb, clearly written, comprehensive overview of nearly all the major issues, ideas, and techniques of this important branch of artificial intelligence, written by two of the masters of the field. The examples are well chosen, and the explanations are illuminating.

Thank you for giving me this opportunity to review and praise a book that has sorely been needed by the KRR community.

**William J. Rapaport, State University of New York at Buffalo**

A concise and lucid exposition of the major topics in knowledge representation, from two of the leading authorities in the field. It provides a thorough grounding, a wide variety of useful examples and exercises, and some thought-provoking new ideas for the expert reader.

**Stuart Russell, UC Berkeley**

No other text provides a clearer introduction to the use of logic in knowledge representation, reasoning, and planning, while also covering the essential ideas underlying practical methodologies such as production systems, description logic-based systems, and Bayesian networks.

**Lenhart Schubert, University of Rochester**

Brachman and Levesque have laid much of the foundations of the field of knowledge representation and reasoning. This textbook provides a lucid and comprehensive introduction to the field. It is written with the same clarity and gift for exposition as their many research publications. The text will become an invaluable resource for students and researchers alike.

**Bart Selman, Cornell University**

KR&R is known as “core AI” for a reason — it embodies some of the most basic conceptualizations and technical approaches in the field. And no researchers are more qualified to provide an in-depth introduction to the area than Brachman and Levesque, who have been at the forefront of KR&R for two decades. The book is clearly written, and is intelligently comprehensive. This is the definitive book on KR&R, and it is long overdue.

**Yoav Shoham, Stanford University**
KNOWLEDGE REPRESENTATION AND REASONING
About the Authors

Ron Brachman has been doing influential work in knowledge representation since the time of his Ph.D. thesis at Harvard in 1977, the result of which was the KL-ONE system, which initiated the entire line of research on description logics. For the majority of his career he served in research management at AT&T, first at Bell Labs and then at AT&T Labs, where he was Communications Services Research Vice President, and where he built one of the premier research groups in the world in Artificial Intelligence. He is a Founding Fellow of the American Association for Artificial Intelligence (AAAI), and also a Fellow of the Association for Computing Machinery (ACM). He is currently President of the AAAI. He served as Secretary-Treasurer of the International Joint Conferences on Artificial Intelligence (IJCAI) for nine years. With more than 60 technical publications in knowledge representation and related areas to his credit, he has led a number of important knowledge representation systems efforts, including the CLASSIC project at AT&T, which resulted in a commercially deployed system that processed more than $5 billion worth of equipment orders. Brachman is currently Director of the Information Processing Technology Office at the U.S. Defense Advanced Research Projects Agency (DARPA), where he is leading a new national-scale initiative in cognitive systems.

Hector Levesque has been teaching knowledge representation and reasoning at the University of Toronto since joining the faculty there in 1984. He has published over 60 research papers in the area, including three that have won best-paper awards. He has also co-authored a book on the logic of knowledge bases and the widely used TELL–ASK interface that he pioneered in his Ph.D. thesis. He and his collaborators have initiated important new lines of research on a number of topics, including implicit and explicit belief, vivid reasoning, new methods for satisfiability, and cognitive robotics. In 1985, he became the first non-American to receive the Computers and Thought Award given by IJCAI. He was the recipient of an E.W.R. Steacie Memorial Fellowship from the Natural Sciences and Engineering Research Council of Canada for 1990–1991. He was also a Fellow of the Canadian Institute for Advanced Research from 1984 to 1995, and is a Founding Fellow of the AAAI. He was elected to the Executive Council of the AAAI, and is on the editorial board of five journals. In 2001, Levesque was the Conference Chair of the IJCAI-01 conference, and is currently Past President of the IJCAI Board of Trustees.

Brachman and Levesque have been working together on knowledge representation and reasoning for more than 25 years. In their early collaborations at BBN and Schlumberger, they produced widely read work on key issues in the field, as well as several well-known knowledge representation systems, including KL-ONE, KRYPTON, and KANDOR. They presented a tutorial on knowledge representation at the International Joint Conference on Artificial Intelligence in 1983. In 1984, they coauthored a prize-winning paper at the National Conference on Artificial Intelligence that is generally regarded as the impetus for an explosion of work in description logics and which inspired many new research efforts on the tractability of knowledge representation systems, including hundreds of research papers. The following year, they edited a popular collection, Readings in Knowledge Representation, the first text in the area. With Ray Reiter, they founded and chaired the international conferences on Principles of Knowledge Representation and Reasoning in 1989; these conferences continue on to this day. Since 1992, they have worked together on the course in knowledge representation at the University of Toronto that is the basis for this book.
In Chapter 4, we saw how a Resolution procedure could in principle be used to calculate entailments of any first-order logic KB. But we also saw that in its most general form Resolution ran into serious computational difficulties. Although refinements to Resolution can help, the problem can never be completely eliminated. This is a consequence of the fundamental computational intractability of first-order entailment.

In this chapter, we will explore the idea of limiting ourselves to only a certain interesting subset of first-order logic, where the Resolution procedure becomes much more manageable. We will also see that from a representation standpoint, the subset in question is still sufficiently expressive for many purposes.

5.1 Horn Clauses

In a Resolution-based system, clauses end up being used for two different purposes. First, they are used to express ordinary disjunctions like

[Rain, Sleet, Snow].

This is the sort of clause we might use to express incomplete knowledge: There is rain or sleet or snow outside, but we don't know which. But consider a clause like

[¬Child, ¬Male, Boy].

Although this can certainly be read as a disjunction, namely, “either someone is not a child, or is not male, or is a boy,” it is much more
naturally understood as a *conditional*: “If someone is a child and is male then that someone is a boy.” It is this second reading of clauses that will be our focus in this chapter.

We call a clause like this—containing at most one positive literal—a *Horn clause*. When there is exactly one positive literal in the clause, it is called a *positive* (or *definite*) Horn clause. When there are no positive literals, the clause is called a *negative* Horn clause. In either case, there can be zero negative literals, and so the empty clause is a negative Horn clause. Observe that a positive Horn clause \([\neg p_1, \ldots, \neg p_n, q]\) can be read as “if \(p_1\) and ... and \(p_n\), then \(q\).” We will sometimes write a clause like this as

\[ p_1 \land \ldots \land p_n \Rightarrow q \]

to emphasize this conditional, “if–then” reading.

Our focus in this chapter will be on using Resolution to reason with if–then statements (which are sometimes called “rules”). Full first-order logic is concerned with disjunction and incomplete knowledge in a more general form, which we are putting aside for the purposes of this chapter.

### 5.1.1 Resolution Derivations with Horn Clauses

Given a Resolution derivation over Horn clauses, observe that two negative clauses can never be resolved together, because all of their literals are of the same polarity. If we are able to resolve a negative and a positive clause together, we are guaranteed to produce a negative clause: The two clauses must be resolved with respect to the one positive literal in the positive clause, and so it will not appear in the resolvent. Similarly, if we resolve two positive clauses together, we are guaranteed to produce a positive clause: The two clauses must be resolved with respect to one (and only one) of the positive literals, so the other positive literal will appear in the resolvent. In other words, Resolution over Horn clauses must always involve a positive clause, and if the second clause is negative, the resolvent is negative; if the second clause is positive, the resolvent is positive.

Less obvious, perhaps, is the following fact: Suppose \(S\) is a set of Horn clauses and \(S \vdash c\), where \(c\) is a negative clause. Then there is guaranteed to be a derivation of \(c\) where all the new clauses in the derivation (i.e., clauses not in \(S\)) are negative. The proof is detailed and laborious, but the main idea is this: Suppose we have a derivation with some new positive clauses. Take the last one of these, and call it \(c'\). Since \(c'\) is the last positive clause in the derivation, all of the Resolution steps after \(c'\) produce negative clauses. We now change the derivation so that instead of generating negative clauses using \(c'\), we generate these negative clauses using the positive parents of \(c'\) (which is where all of the literals in \(c'\) come from—\(c'\)
must have only positive parents, because it is a positive clause). We know we can do this because in order to get to the negative successor(s) of \( c' \), we must have a clause somewhere that can resolve with it to eliminate the one positive literal in \( c' \) (call that clause \( d \) and the literal \( p \)). That \( p \) must be present in one of the (positive) parents of \( c' \), so we just use clause \( d \) to resolve against the parent of \( c' \), thereby eliminating \( p \) earlier in the derivation and producing the negative clauses without producing \( c' \). The derivation still generates \( c \), but this time without needing \( c' \). If we repeat this for every new positive clause introduced, we eliminate all of them.

We can go further: Suppose \( S \) is a set of Horn clauses and \( S \vdash c \), where \( c \) is again a negative clause. Then there is guaranteed to be a derivation of \( c \) where each new clause derived is not only negative, but is a resolvent of the previous one in the derivation and an original clause in \( S \). The reason is this: By the earlier argument, we can assume that each new clause in the derivation is negative. This means that it has one positive and one negative parent. Clearly, the positive parent must be from the original set (because all the new ones are negative). Each new clause then has exactly one negative parent. So starting with \( c \), we can work our way back through its negative ancestors and end up with a negative clause that is in \( S \). Then, by discarding all the clauses that are not on this chain from \( c \) to \( S \), we end up with a derivation of the required form.

These observations lead us to the following conclusion:

There is a derivation of a negative clause (including the empty clause) from a set of Horn clauses \( S \) if and only if there is one where each new clause in the derivation is a negative resolvent of the previous clause in the derivation and some element of \( S \).

We will look at derivations of this form in more detail in the next section.

---

### 5.2 SLD RESOLUTION

The observations of the previous section lead us to consider a very restricted form of Resolution that is sufficient for Horn clauses. This is a form of Resolution where each new clause introduced is a resolvent of the previous clause and a clause from the original set. This pattern showed up repeatedly in the examples of Chapter 4, and is illustrated schematically in Figure 5.1.\(^1\)

Let us be a little more formal about this. For any set \( S \) of clauses (Horn or not), an SLD derivation of a clause \( c \) from \( S \) is a sequence of clauses \( c_1, c_2, \ldots, c_n \), such that \( c_n = c \), \( c_1 \in S \), and \( c_{i+1} \) is a resolvent of \( c_i \) and

\(^1\)The pattern appears in Figure 4.4, but not Figure 4.5.
some clause of $S$. We write $S \vdash_{\text{SLD}} c$ if there is an SLD derivation of $c$ from $S$. Notationally, because of its structure, an SLD derivation is simply a type of Resolution derivation where we do not explicitly mention the elements of $S$ except for $c_1$.\footnote{The name SLD stands for Selected literals, Linear pattern, over Definite clauses.} We know that at each step of the way the obvious positive parent from $S$ can be identified, so we can leave it out of our description of the derivation and just show the chain of negative clauses from $c_1$ to $c$.

In the general case, it should be clear that if $S \vdash_{\text{SLD}} []$ then $S \vdash []$. The converse, however, is not true in general. For example, let $S$ be the set of clauses $[p, q], [\neg p, q], [p, \neg q]$, and $[\neg p, \neg q]$. A quick glance at these clauses should convince us that $S$ is unsatisfiable (whatever values we pick for $p$ and $q$, we cannot make all four clauses true at the same time). Therefore, $S \vdash []$. However, to generate $[]$ by Resolution, the last step must involve two complementary unit clauses $[\rho]$ and $[\overline{\rho}]$, for some atom $\rho$. Since $S$ contains no unit clauses, it will not be possible to use an element of $S$ for this last step. Consequently there is no SLD derivation of $[]$ from $S$, even though $S \vdash []$.

In the previous section we argued that for Horn clauses we could get by with Resolution derivations of a certain shape, wherein each new clause in the derivation was a negative resolvent of the previous clause.
in the derivation and some element of \( S \); we have now called such derivations SLD derivations. So although not the case for Resolution in general, it is the case that if \( S \) is a set of Horn clauses, then \( S \vdash [\] \) if and only if \( S \vdash_{SLD} [\] \). So if \( S \) is Horn, then it is unsatisfiable if and only if \( S \vdash_{SLD} [\] \). Moreover, we know that each of the new clauses \( c_2, \ldots, c_n \) can be assumed to be negative. So \( c_2 \) has a negative and a positive parent, and thus \( c_1 \in S \) can be taken to be negative as well. Thus in the Horn case, SLD derivations of the empty clause must begin with a negative clause in the original set.

To see an example of an SLD derivation, consider the first example of Chapter 4. We start with a KB containing the following positive Horn clauses:

\[
\text{Toddler} \\
\text{Toddler} \supset \text{Child} \\
\text{Child} \land \text{Male} \supset \text{Boy} \\
\text{Infant} \supset \text{Child} \\
\text{Child} \land \text{Female} \supset \text{Girl} \\
\text{Female}
\]

and wish to show that KB \( \models \text{Girl} \), that is, that there is an SLD derivation of \( [\] \) from KB together with the negative Horn clause \( [\neg \text{Girl}] \). Because this is the only negative clause, it must be the \( c_1 \) in the derivation. By resolving it with the fifth clause in the KB, we get \( [\neg \text{Child}, \neg \text{Female}] \) as \( c_2 \). Resolving this with the sixth clause, we get \( [\neg \text{Child}] \) as \( c_3 \). Resolving this with the second clause, we get \( [\neg \text{Toddler}] \) as \( c_4 \). And finally, resolving this with the first clause, we get \( [\] \) as the final clause. Observe that all the clauses in the derivation are negative. To display this derivation, we could continue to use Resolution diagrams from Chapter 4. However, for SLD derivations, it is convenient to use a special-purpose terminology and format.

### 5.2.1 Goal Trees

All the literals in all the clauses in a Horn SLD derivation of the empty clause are negative. We are looking for positive clauses in the KB to “eliminate” these negative literals to produce the empty clause. Sometimes, there is a unit clause in the KB that eliminates the literal directly. For example, if a clause like \( [\neg \text{Toddler}] \) appears in a derivation using the earlier KB, then the derivation is finished, because there is a positive clause in the KB that resolves with it to produce the empty clause. We say in this case that the goal \text{Toddler} is solved. Sometimes there is a positive clause that eliminates the literal but introduces other negative literals.
For example, with a clause like \([-\text{Child}]\) in the derivation, we continue with the clause \([-\text{Toddler}]\), having resolved it against the second clause in our knowledge base \(([-\text{Toddler}, \text{Child}]\)). We say in this case that the goal \text{Child} reduces to the subgoal \text{Toddler}. Similarly, the goal \text{Girl} reduces to two subgoals, \text{Child} and \text{Female}, since two negative literals are introduced when it is resolved against the fifth clause in the KB.

A restatement of the SLD derivation is as follows: We start with the goal \text{Girl}. This reduces to two subgoals, \text{Child} and \text{Female}. The goal \text{Female} is solved, and \text{Child} reduces to \text{Toddler}. Finally, \text{Toddler} is solved.

We can display this derivation using what is called a goal tree. We draw the original goal (or goals) at the top, and point from there to the subgoals. For a complete SLD derivation, the leaves of the tree (at the bottom) will be the goals that are solved (see Figure 5.2). This allows us to easily see the form of the argument: We want to show that \text{Girl} is entailed by the KB. Reading from the bottom up, we know that \text{Toddler} is entailed because it appears in the KB. This means that \text{Child} is entailed. Furthermore, \text{Female} is also entailed (because it appears in the KB), so we conclude that \text{Girl} is entailed.

This way of looking at Horn clauses and SLD derivations, when generalized to deal with variables in the obvious way, forms the basis of the programming language PROLOG. We already saw an example of a PROLOG-style definition of addition in Chapter 4. Let us consider another example involving lists. For our purposes, list terms will either be variables, the constant \text{nil}, or a term of the form \text{cons}(t_1, t_2), where \(t_1\) is any term and \(t_2\) is a list term. We will write clauses defining the \text{Append}(x, y, z)\ relation, intended to hold when list \(z\) is the result of appending \(y\) to list \(x\):

\[
\text{Append}(\text{nil}, y, y)
\]

\[
\text{Append}(x, y, z) \Rightarrow \text{Append}(\text{cons}(w, x), y, \text{cons}(w, z))
\]
If we wish to show that this entails

\[ \text{Append}(\text{cons}(a,\text{cons}(b,\text{nil})), \text{cons}(c,\text{nil}), \text{cons}(a,\text{cons}(b,\text{cons}(c,\text{nil})))) \]

we get the goal tree in Figure 5.3. We can also use a variable in the goal and show that the definition entails \( \exists u. \text{Append}(\text{cons}(a,\text{cons}(b,\text{nil})), \text{cons}(c,\text{nil}), u) \). The answer \( u = \text{cons}(a,\text{cons}(b,\text{cons}(c,\text{nil}))) \) can be extracted from the derivation directly. Unlike ordinary Resolution, it is not necessary to use answer predicates with SLD derivations. This is because if \( S \) is a set of Horn clauses, then \( S \models \exists x. \alpha \) if and only if for some term \( t, S \models \alpha t \).

### 5.3 Computing SLD Derivations

We now turn our attention to procedures for reasoning with Horn clauses. The idea is that we are given a KB containing a set of positive Horn clauses representing if–then sentences, and we wish to know whether or not some atom (or set of atoms) is entailed. Equivalently, we wish to know whether or not the KB together with a clause consisting of one or more negative literals is unsatisfiable. Thus the typical case, and the one we will consider here, involves determining the satisfiability of a set of Horn clauses containing exactly one negative clause.\(^3\)

#### 5.3.1 Backward Chaining

A procedure for determining the satisfiability of a set of Horn clauses with exactly one negative clause is presented in Figure 5.4. This procedure

\(^3\)It is not hard to generalize the procedures presented here to deal with more than one negative clause (see Exercise 4). Similarly, the procedures can be generalized to answer entailment questions where the query is an arbitrary (non-Horn) formula in CNF.
input: a finite list of atomic sentences, \( q_1, \ldots, q_n \)

output: YES or NO according to whether a given KB entails all of the \( q_i \)

procedure \( \text{SOLVE}[q_1, \ldots, q_n] \)

\[ \begin{align*}
    & \text{if } n = 0 \text{ then return YES} \\
    & \text{for each clause } c \in \text{KB, do} \\
    & \quad \text{if } c = [q_1, \neg p_1, \ldots, \neg p_m] \\
    & \quad \quad \text{and } \text{SOLVE}[p_1, \ldots, p_m, q_2, \ldots, q_n] \\
    & \quad \text{then return YES} \\
    & \text{end for} \\
    & \text{return NO}
\end{align*} \]

\( \text{FIGURE 5.4} \)

A Recursive Backward-Chaining SLD Procedure

starts with a set of goals as input (corresponding to the atoms in the single negative clause) and attempts to solve them. If there are no goals, then it is done. Otherwise, it takes the first goal \( q_1 \) and looks for a clause in KB whose positive literal is \( q_1 \). Using the negative literals in that clause as subgoals, it then calls itself recursively with these subgoals together with the rest of the original goals. If this is successful, it is done; otherwise it must consider other clauses in the KB whose positive literal is \( q_1 \). If none can be found, the procedure returns NO, meaning the atoms are not entailed.

This procedure is called \textit{backward chaining}, because it works backward from goals to facts in the KB. It is also called \textit{depth-first}, because it attempts to solve the new goals \( p_i \) before tackling the old goals \( q_i \). Finally, it is called \textit{left-to-right}, because it attempts the goals \( q_i \) in order 1, 2, 3, and so on. This depth-first left-to-right backward-chaining procedure is the one normally used by PROLOG implementations to solve goals, although the first-order case obviously requires unification, substitution of variables, and so on.

This backward-chaining procedure also has a number of drawbacks. First, observe that even in the propositional case it can go into an infinite loop. Suppose we have the tautologous \( [p, \neg p] \) in the KB.\(^4\) In this case, a goal of \( p \) can reduce to a subgoal of \( p \), and so on, indefinitely.

Even if it does terminate, the backward-chaining algorithm can be quite inefficient and do a considerable amount of redundant searching. For example, imagine that we have \( 2n \) atoms \( p_0, \ldots, p_{n-1} \) and \( q_0, \ldots, q_{n-1} \),

\(^4\text{This corresponds to the PROLOG program } "p : - p."\)
5.3 Computing SLD Derivations

**input**: a finite list of atomic sentences, $q_1, \ldots, q_n$

**output**: YES or NO according to whether a given KB entails all of the $q_i$

1. if all of the goals $q_i$ are marked as solved, then return YES
2. check if there is a clause $[p, \neg p_1, \ldots, \neg p_n]$ in KB, such that all of its negative atoms $p_1, \ldots, p_n$ are marked as solved, and such that the positive atom $p$ is not marked as solved
3. if there is such a clause, mark $p$ as solved and go to step 1
4. otherwise, return NO

**FIGURE 5.5**
A Forward-Chaining SLD Procedure

and the following $4n - 4$ clauses: For $0 < i < n$,

$$p_{i-1} \Rightarrow p_i$$

$$p_{i-1} \Rightarrow q_i$$

$$q_{i-1} \Rightarrow p_i$$

$$q_{i-1} \Rightarrow q_i$$

For any $i$, both $\text{SOLVE}[p_i]$ and $\text{SOLVE}[q_i]$ will eventually fail, but only after at least $2^i$ steps. The proof is a simple induction argument.\(^5\) This means that even for a reasonably sized KB (say 396 clauses when $n = 100$), an impossibly large amount of work may be required (over $2^{100}$ steps).

Given this exponential behavior, we might wonder if this is a problem with the backward-chaining procedure or another instance of what we saw in the last chapter where the entailment problem itself was simply too hard in its most general form. As it turns out, this time it is the procedure that is to blame.

### 5.3.2 Forward Chaining

In the propositional case, there is a much more efficient procedure to determine if a Horn KB entails a set of atoms, given in Figure 5.5. This is a forward-chaining procedure, because it works from the facts in the KB toward the goals. The idea is to mark atoms as “solved” as soon as we have determined that they are entailed by the KB.

---

\(^5\) The claim is clearly true for $i = 0$. For the goal $p_k$, where $k > 0$, we need to try to solve both $p_{k-1}$ and $q_{k-1}$. By induction, each of these take at least $2^{k-1}$ steps, for a total of $2^k$ steps. The case for $q_k$ is identical.
Suppose, for example, we start with the earlier Girl example. At the outset Girl is not marked as solved, so we go to step 2. At this point, we look for a clause satisfying the given criteria. The clause [Toddler] is one such, because all of its negative literals (of which there are none) are marked as solved. So we mark Toddler as solved and try again. This time we might find the clause [Child, ¬Toddler], and so we can mark Child as solved and try again. Continuing in this way, we mark Female and finally Girl as solved and we are done.

Although this procedure appears to take about the same effort as the backward-chaining one, it has much better overall behavior. Note, in particular, that each time through the iteration we need to find a clause in the KB with an atom that has not been marked. Thus, we will iterate at most as many times as there are clauses in the KB. Each such iteration step may require us to scan the entire KB, but the overall result will never be exponential. In fact, with a bit of care in the use of data structures, a forward-chaining procedure like this can be made to run in time that is linear in the size of the KB, as will be demonstrated in Exercise 1.

5.3.3 The First-Order Case

Thus, in the propositional case at least, we can determine if a Horn KB entails an atom in a linear number of steps. But what about the first-order case? Unfortunately, even with Horn clauses, we still have the possibility of a procedure that runs forever. The example in Figure 4.11, where an infinite branch of resolvents was generated, only required Horn clauses. While it might seem that a forward-chaining procedure could deal with first-order examples like these, avoiding the infinite loops, this cannot be: The problem of determining whether a set of first-order Horn clauses entails an atom remains undecidable. So no procedure can be guaranteed to always work, despite the fact that the propositional case is so easy. This is not too surprising, because PROLOG is a full programming language, and being able to decide if an atom is entailed would imply being able to decide if a PROLOG program would halt.

As with non-Horn clauses, the best that can be expected in the first-order case is to give control of the reasoning to the user to help avoid redundancies and infinite branches. Unlike the non-Horn case, however, Horn clauses are much easier to structure and control in this way. In the next chapter, we will see some examples of how this can be done.

5.4 BIBLIOGRAPHIC NOTES

Horn formulas were first studied by Alfred Horn [190] and are named after him. The SLD Resolution procedure was introduced by Kowalski.
5.5 Exercises

1. Write, test, and document a program that determines the satisfiability of a set of propositional Horn clauses by forward chaining and that runs in linear time, relative to the size of the input. Use the following data structures:

   (a) a global variable STACK containing a list of atoms known to be true, but waiting to be propagated forward;

   (b) for each clause, an atom CONCLUSION, which is the positive literal appearing in the clause (or NIL if the clause contains only negative literals), and a number REMAINING, which is the number of atoms appearing negatively in the clause that are not yet known to be true;

   (c) for each atom, a flag VISITED indicating whether or not the atom has been propagated forward, and a list ON-CLAUSES of all the clauses where the atom appears negatively.

You may assume the input is in suitable form. Include in the documentation an argument as to why your program runs in linear time. (If you choose to use LISP property lists for your data structures, you may assume that it takes constant time to go from an atom to any of its properties.)

2. As noted in Chapter 4, Herbrand's Theorem allows us to convert a first-order satisfiability problem into a propositional (variable-free) one, although the size of the Herbrand base, in general, is infinite. One way to deal with an infinite set \( S \) of clauses is to look at progressively larger subsets of it to see if any of them are unsatisfiable, in which case \( S \) must be as well. In fact, the converse is true: If \( S \) is
unsatisfiable, then some finite subset of $S$ is unsatisfiable too. This is called the \textit{compactness} property of FOL. One way to generate progressively larger subsets of $S$ is as follows:

For any term $t$, let $|t|$ be defined as 0 for variables and constants, and $1 + \max|t_i|$ for terms $f(t_1, \ldots, t_n)$.

Now for any set $S$ of formulas, define $S_k$ to be those elements $\alpha$ of $S$ such that every term $t$ of $\alpha$ has $|t| \leq k$.

(a) Write and test a program that given a finite set $S$ of first-order clauses and a positive number $k$ returns as value $H_k$, where $H$ is the Herbrand base of $S$.

(b) When the original set $S$ is Horn, then for any $k$, your program returns a finite set of propositional Horn clauses. These can be checked for satisfiability using a propositional program like the one in Exercise 1. Briefly compare this way of testing the satisfiability of $S$ to the more standard way using SLD Resolution, as in PROLOG.

3. Consider the more general version of Resolution discussed in Exercise 4 of Chapter 4. Is that generalization required for SLD-resolution? Explain.

4. In this question, we will explore the semantic properties of propositional Horn clauses. For any set of clauses $S$, define $I_S$ to be the interpretation that satisfies an atom $p$ if and only if $S \models p$.

(a) Show that if $S$ is a set of positive Horn clauses, then $I_S \models S$.

(b) Give an example of a set of clauses $S$ where $I_S \not\models S$.

(c) Suppose that $S$ is a set of positive Horn clauses and that $c$ is a negative Horn clause. Show that if $I_S \not\models c$ then $S \cup \{c\}$ is unsatisfiable.

(d) Suppose that $S$ is a set of positive Horn clauses and that $T$ is a set of negative ones. Using part (c), show that if $S \cup \{c\}$ is satisfiable for every $c \in T$, then $S \cup T$ is satisfiable also.

(e) In the propositional case, the normal PROLOG interpreter can be thought of as taking a set of positive Horn clauses $S$ (the program) and a single negative clause $c$ (the query) and determining whether or not $S \cup \{c\}$ is satisfiable. Use part (d) to conclude that PROLOG can be used to test the satisfiability of an arbitrary set of Horn clauses.

5. In this question, we will formalize a fragment of high school geometry. We will use a single binary predicate symbol, which we write here as $\equiv$. The objects in this domain are points, lines, angles,
and triangles. We will use constants only to name the points we need, and for the other individuals we will use function symbols that take points as arguments: first, a function that given two points is used to name the line between them, which we write here as $\overline{AB}$, where $A$ and $B$ are points; next, a function that given three points names the angle between them, which we write here as $\angle ABC$; and finally, a function that given three points names the triangle between them, which we write here as $\triangle ABC$.

Here are the axioms of interest:

- $\equiv$ is an equivalence relation.
- $\overline{XY} \equiv \overline{YX}$.
- $\angle XYZ \equiv \angle ZYX$.
- If $\triangle XYZ \equiv \triangle UVW$, then the corresponding lines and angles are congruent ($\overline{XY} \equiv \overline{UV}$, $\angle XYZ \equiv \angle UVW$, etc.).
- **SAS:** If $\overline{XY} \equiv \overline{UV}$, $\angle XYZ \equiv \angle UVW$, and $\overline{YZ} \equiv \overline{VW}$, then $\triangle XYZ \equiv \triangle UVW$.

(a) Show that these axioms imply that the base angles of an isosceles triangle must be equal, that is, that

$\text{Axioms } \cup \overline{AB} \equiv \overline{AC} \models \angle ABC \equiv \angle ACB$.

Because the axioms can be formulated as Horn clauses and the other two sentences are atomic, it is sufficient to present an SLD derivation.

(b) The theorem in part (a) can also be proven by constructing the midpoint of the side $BC$ (call it $D$), and showing that $\triangle ABD \equiv \triangle ACD$ (by using **SSS**, the fact that two triangles are congruent if the corresponding sides are all congruent). What difficulties do you foresee in automated reasoning with constructed points like this?