Statistical Relational Learning Notes

Ganesh Ramakrishnan and Ashwin Srinivasan

November 20, 2008

Contents

Chapter 1

Sets, Relations and Logic

'Crime is common. Logic is rare. Therefore it is upon the logic rather than the crime that you should dwell.' Sherlock Holmes in Conan Doyle's *The Copper Breeches*.

1.1 Sets and Relations

1.1.1 Sets

A *set* is a fundamental concept in mathematics. Simply speaking, it consists of some objects, usually called its *elements*. Here are some basic notions about sets that you must already know about:

– A set S with elements a, b and c is usually written as $S = \{a, b, c\}$. The fact that a is an element of S is usually denoted by $a \in S$.

– A set with no elements is called the "empty set" and is denoted by $\emptyset$.

– Two sets S and T are equal ($S = T$) if and only if they contain precisely the same elements. Otherwise $S \neq T$.

– A set T is a subset of a set S ($T \subseteq S$) if and only if every element of T is also an element of S. If $T \subseteq S$ and $S \subseteq T$ then $S = T$. Sometimes $T \subseteq S$ may sometimes also be written as $S \supseteq T$. If $T \subseteq S$ and S has at least one element not in T, then $T \subset S$ (T is said to be "proper subset" of S). Again, $T \subset S$ may sometimes be written as $S \supset T$.

We now look at the meanings of the *union*, *intersection*, and *equivalence* of sets. The intersection, or product, of sets S and T, denoted by $S \cap T$ or ST or $S \cdot T$ consists of all elements common to both S and T. $ST \subset S$ and $ST \subseteq T$ for all sets S and T. Now, if S and T have no elements in common, then they are said to be *disjoint* and $ST = \emptyset$. It should be easy for you to see that $\emptyset \subseteq S$ for all S and $\emptyset \cdot S = \emptyset$ for all S. The union, or sum, of sets S and T, denoted by $S \cup T$ or $S + T$, is the set consisting of elements that belong at least to S or T. Once again, it should be a straightforward matter to see $S \subseteq S + T$ and $T \subseteq S + T$ for all S and T. Also, $S + \emptyset = S$ for all S. Finally, if there is a one-to-one correspondence between the elements of set S and set T, then S and T are said to be equivalent ($S \sim T$). Equivalence and subsets form the basis of the definition of an infinte set: if $T \subset S$ and $S \sim T$ then S is said to be an infinite set. The set of natural numbers $\mathcal{N}$ is an example of an infinite set (any set $S \sim \mathcal{N}$ is said to be *countable* set).

1.1.2 Relations

A finite sequence is simply a set of n elements with a $1-1$ correspondence with the set $\{1, \ldots, n\}$ arranged in order of succession (an *ordered pair*, for example, is just a finite sequence with 2 elements). Finite sequences allow us to formalise the concept of a relation. If A and B are sets, then the set $A \times B$ is called the *cartesian product* of A and B and is denoted by all ordered pairs (a, b) such that $a \in A$ and $b \in B$. Any subset of $A \times B$ is a binary relation, and is called a relation from A to B. If $(a, b) \in R$, then aRb means "a is in relation R to b" or, "relation R holds for the ordered pair (a, b)" or "relation R holds between a and b." A special case arises from binary relations within elements of a single set (that is, subsets of $A \times A$). Such a relation is called a "relation in A" or a "relation over A". There are some important kinds of properties that may hold for a relation R in a set A:

Reflexive. The relation is said to be reflexive if the ordered pair $(a, a) \in R$ for every $a \in A$.

Symmetric. The relation is said to be symmetric if $(a, b) \in R$ *iff* $(b, a) \in R$ for $a, b \in A$.

Transitive. The relation is said to be transitive if $(a, b) \in R$ and $(b, c) \in R$, then $(a, c) \in R$ for $a, b, c \in A$.

Here are some examples:

- The relation $\leq$ on the set of integers is reflexive and transitive, but not symmetric.

- The relation $R = \{(1,1), (1,2), (2,1), (2,2), (3,3), (4,4)\}$ on the set $A = \{1, 2, 3, 4\}$ is reflexive, symmetric and transitive.

- The relation $\div$ on the set $\mathcal{N}$ defined as the set $\{(x, y) : \exists z \in \mathcal{N} \text{ s.t. } xz = y\}$ is reflexive and transitive, but not symmetric.

- The relation $\perp$ on the set of lines in a plane is symmetric but neither reflexive nor transitive.

It should be easy to see a relation like R above is just a set of ordered pairs. Functions are just a special kind of binary relation F which is such that if $(a, b) \in F$ and $(a, c) \in F$ then $b = c$. Our familiar notion of a function F from a set A to a set B is one which associates with each $a \in A$ exactly one element $b \in B$ such that $(a, b) \in F$. Now, a function from a set A to itself is usually called a *unary operation* in A. In a similar manner, a *binary operation* in A is a function from $A \times A$ to A (recall $A \times A$ is the Cartesian product of A with itself: it is sometimes written as A^2). For example, if $A = \mathcal{N}$, then addition ($+$) is a binary operation in A. In general, an n-ary operation F in A is a function from A^n to A, and if it is defined for every element of A^n, then A is said to be *closed* with respect to the operation F. A set which is closed for one or more n-ary operations is called an *algebra*, and a sub-algebra is a subset of such a set that remains closed with respect to those operations. For example:

- $\mathcal{N}$ is closed wrt the binary operations of $+$ and $\times$, and $\mathcal{N}$ along with $+, \times$ form an algebra.
- The set $\mathcal{E}$ of even numbers is a subalgebra of algebra of $\mathcal{N}$ with $+, \times$. The set $\mathcal{O}$ of odd numbers is not a subalgebra.
- Let $S \subseteq U$ and $S' \subseteq U$ be the set with elements of U not in S (the unary operation of complementation). Let $U = \{a, b, c, d\}$. The subsets of U with the operations of complementation, intersection and union form an algebra. (How many subalgebras are there of this algebra?)

Equivalence Relations

Any relation R in a set A for which all three properties hold (that is, R is reflexive, symmetric, and transitive) is said to be an "equivalence relation". Suppose, for example, we are looking at the relation R over the set of natural numbers $\mathcal{N}$, which consists of ordered pairs (a, b) such that $a + b$ is even[1] You should be able to verify that R is an equivalence relation over $\mathcal{N}$. In fact, R allows us to split $\mathcal{N}$ into two disjoint subsets: the set of odd numbers $\mathcal{O}$ and the set of even numbers $\mathcal{E}$ such that $\mathcal{N} = \mathcal{O} \cup \mathcal{E}$ and R is an equivalence relation over each of $\mathcal{O}$ and $\mathcal{E}$. This brings us to an important property of equivalence relations:

Theorem 1 *Any equivalence relation E over a set S partitions S into disjoint non-empty subsets $S_1, \ldots, S_k$ such that $S = S_1 \cup \cdots \cup S_k$.*

Let us see how E can be used to partition S by constructing subsets of S in the following way. For every $a \in S$, if $(a, b) \in E$ then a and b are put in the same subset. Let there be k such subsets. Now, since $(a, a) \in E$ for every $a \in S$, every element of S is in some subset. So, $S = S_1 \cup \cdots \cup S_k$. It also follows that the subsets are disjoint. Otherwise there must be some $c \in S_i, S_j$. Clearly, S_i

[1]Equivalence is often denoted by $\approx$. Thus, for an equivalence relation E, if $(a, b) \in E$, then $a \approx b$.

and S_j are not singleton sets. Suppose S_i contains at least a and c. Further let there be a $b \notin S_i$ but $b \in S_j$. Since $a, c \in S_i$, $(a,c) \in E$ and since $c, b \in S_j$, $(c,b) \in E$. Thus, we have $(a,c) \in E$ and $(c,b) \in E$, which must mean that $(a,b) \in E$ (E is transitive). But in this case b must be in the same subset as a by construction of the subsets, which contradicts our assumption that $b \notin S_i$. The converse of this is also true:

Theorem 2 *Any partition of a set S partitions into disjoint non-empty subsets $S_1, \ldots, S_k$ such that $S = S_1 \cup \cdots \cup S_k$ results in an equivalence relation over S.*

(Can you prove that this is the case? Start by constructing a relation E, with $(a,b) \in E$ if and only if a and b are in the same block, and prove that E is an equivalence relation.)

Each of the disjoint subsets $S_1, S_2, \ldots$ are called "equivalence classes", and we will denote the equivalence class of an element a in a set S by $[a]$. That is, for an equivalence relation E over a set S:

$$[a] = \{x : x \in S, (a,x) \in E\}$$

What we are saying above is that the collection of all equivalence classes of elements of S forms a partition of S; and conversely, given a partition of the set S, there is an equivalence relation E on S such that the sets in the partition (sometimes also called its "blocks") are the equivalence classes of S.

Partial Orders

Given an equality relation $=$ over elements of a set S, a partial order $\preceq$ over S is a relation over S that satisfies the following properties:

Reflexive. For every $a \in S$, $a \preceq a$

Anti-Symmetric. If $a \preceq b$ and $b \preceq a$ then $a = b$

Transitive. If $a \preceq b$ and $b \preceq c$ then $a \preceq c$

Here are some properties about partial orders that you should know (you will be able to understand them immediately if you take, as a special case, $\preceq$ as meaning $\leq$ and $\prec$ as meaning $<$):

- If $a \preceq b$ and $a \neq b$ then $a \prec b$
- $b \succeq a$ means $a \preceq b$, $b \succ a$ means $a \prec b$
- If $a \preceq b$ or $b \preceq a$ then a, b are comparable, otherwise they are not comparable.

A set S over which a relation of partial order is defined is called a *partially ordered set*. It is sometimes convenient to refer to a set S and a relation R defined over S together by the pair $< S, R >$. So, here are some examples of partially ordered sets $< S, \preceq >$:

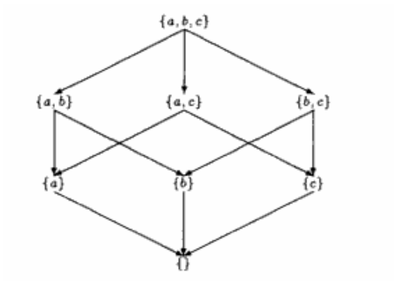


Figure 1.1: The lattice structure of $\langle S, \preceq \rangle$, where S is the power set of $\{a, b, c\}$.

- S is a set of sets, $S_1 \preceq S_2$ means $S_1 \subseteq S_2$
- $S = \mathcal{N}$, $n_1 \preceq n_2$ means $n_1 = n_2$ or there is a $n_3 \in \mathcal{N}$ such that $n_1 + n_3 = n_2$
- S is the set of equivalence relations $E_1, \ldots$ over some set T, $E_L \preceq E_M$ means for $u, v \in T$, uE_Lv means uE_Mv (that is, $(u, v) \in E_L$ means $(u, v) \in E_M$).

Given a set $S = \{a, b, \ldots\}$ if $a \prec b$ and there is no $x \in S$ such that $a \prec x \prec b$ then we will say b *covers* a or that a is a *downward cover* of b. Now, suppose S_{down} be a set of downward covers of $b \in S$. If for all $x \in S$, $x \prec b$ implies there is an $a \in S_{down}$ s.t. $x \preceq a \prec b$, then S_{down} is said to be a *complete* set of downward covers of b. Partially ordered sets are usually shown as diagrams like in Figure 1.1.
The diagrams, as you can see, are graphs (sometimes called Hasse graphs or Hasse diagrams). In the graph, vertices represent elements of the partially ordered set. A vertex v_2 is at a higher level than vertex v_1 whenever $v_1 \prec v_2$, and there is an edge between the two vertices only if v_2 covers v_1 (that is, v_2 is an immediate predecessor). The graph is therefore really a directed one, in which there is a directed edge from a vertex v_2 to v_1 whenever v_2 covers v_1. Also, since the relation is anti-symmetric, there can be no cycles. So, the graph is a directed acyclic graph, or DAG.

In the diagram in Figure **??** on the left, S is the set of non-empty subsets of $\{a, b, c\}$ and $\preceq$ denotes the subset relationship (that is, $S_1 \preceq S_2$ if and only if $S_1 \subset S_2$). The diagram on the right is an example of a *chain*, or a *totally ordered* set.

You should be able to see that a finite chain of length n can be put in a one-to-one correspondence to a finite sequence of natural numbers $(1, \ldots, n)$ (the correct way to say this is that a finite chain is isomorphic with a finite sequence of natural numbers). In general, a partially ordered set S is a chain if for every pair $a, b \in S$, $a \prec b$ or $b \prec a$. There is a close relationship between a partially ordered set and a chain. Suppose S is a partially ordered set. We can always associate a function f from the elements of S to $\mathcal{N}$ (the set of natural numbers), so that if $a \prec b$ for $a, b \in S$, then $f(a) < f(b)$. f is called a *consistent enumeration* of S, and is not unique and we can use it to define a chain consistent with S. (We will leave the proof of the existence a consistent enumeration for you. One way would be to use the method of induction on the number of elements in S: clearly there is such an enumeration for $|S| = 1$. Assume that an enumeration exists for $|S| = n - 1$ and prove it for $|S| = n$.)

Some elements of a partially ordered set have special properties. Let $< S, \preceq >$ be a p.o. set and $T \subseteq S$. Then (in the following, you should read the symbol $\exists$ as being shorthand for "there exists", and $\forall$ as "for all"):

– Least element of T $a \in T\ s.t.\ \forall t \in T\ a \preceq t$	– Greatest element of T $a \in T s.t.\ \forall t \in T\ a \succeq t$
– Least element, if it exists, is unique. If $T = S$ this is the "zero" element	– Greatest element, if it exists is unique. If $T = S$ then this is the "unity" element
– Minimal element of T $a \in T\ \ \nexists t \in T\ s.t.\ t \prec a$	– Maximal element of T $a \in T\ \ \nexists t \in T\ s.t.\ t \succ a$
– Minimal element need not be unique	– Maximal element need not be unique
– Lower bound of T $b \in S\ s.t.\ b \preceq t\ \forall t \in T$	– Upper bound of T $b \in S\ s.t.\ b \succeq t\ \forall t \in T$
– Glb g of T $b \preceq g\ \forall b, g :\ lbs\ of\ T$	– Lub g of T $b \succeq g\ \forall b, g :\ ubs\ of\ T$
– If it exists, the glb is unique	– If it exists the lub is unique

As you would have observed, there is a difference between a least element and a minimal element (and correspondingly, between greatest and maximal elements). The requirement of a minimal (maximal) upper bound is, in some sense, a weakening of the requirement of a least (greatest) upper bound. If x and y are both lub's of some set $T \subseteq S$, then $y \preceq x$ and $z \preceq y$, so then $x \approx y$. This means that all lub's of T are equivalent. Dually, if x and y are glb's of some T, then also $x \approx y$. Thus, if a least element exists, then it is unique: this is not necessarily the case with a minimal element. Also, least and greatest elements must belong to the set T, but lower and upper bounds need not.
For this example, S has: (1) one upper bound b; (2) no lower bound; (3) a greatest element b; (4) no least element; (5) no greatest lower bound; (6) two minimal elements a and e; and (7) one maximal element b. Can you identify what the corresponding statements are for T?

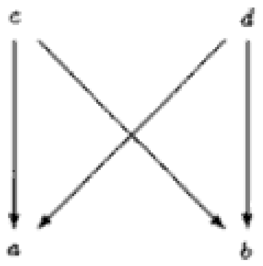


Figure 1.2: $\{a, b\}$ has no lub here.

The glb and lub are sometimes also taken to be binary operations on a partially ordered set S, that assigns to an ordered pair in S^2 the corresponding glb or lub. The first operation is called the *product* or *meet* and is denoted by $\cdot$ or $\sqcap$. The second operation is sometimes called the *sum* or *join* and is denoted by $+$ or $\sqcup$.

In a quasi-ordered set, a subset need not have a lub or glb. We will take an example to illustrate this. Let $S = \{a, b, c, d\}$, and let $\preceq$ be defined as $a \preceq c$, $b \preceq c$, $a \preceq d$ and $d \preceq b$. Then since c and d are incomparable, the set a, b has no lub in this quasi-order. See Figure 1.2.

Similarly, a set need not have a maximal or a minimal, nor upward or downward covers. For instance, let S be the infinite set $\{y, x_l, x_2, x_3, \ldots\}$, and let $\preceq$ be a quasi-order on S, defined as $y \prec \ldots x_{n+1} \prec x_n \prec \ldots \prec x_2 \prec x_1$. Then there is no upward cover of y: for every x_n, there always is an x_{n+l} such that $y \prec x_{n+1} \prec x_n$. In this case, y has no complete set of upward covers.

Note that a complete set of upward covers for y need not contain all upward covers of y. However, in order to be complete, it should contain at least one element from each equivalence class of upward covers. On the other hand, even the set of all upward covers of y need not be complete for y. For the example given above, the set of all upward covers of y is empty, but obviously not complete.

A notion of some relevance later is that of a function f defined on a partially ordered set $< S, \preceq >$. Specifically, we would like to know if the function is: (a) monotonic; and (b) continuous. Monotonicity first:

> A function f on $< S, \preceq >$ is monotonic if and only if for all $u, v \in S$, $u \preceq v$ means $f(u) \preceq f(v)$

Now, suppose a subset S_1 of S have a least upper bound $lub(S_1)$ (with some abuse of notation: here $lub(X)$ is taken to be the lub of the elements in set X). Such subsets are called "directed" subsets of S. Then:

> A function f on $< S, \preceq >$ is continuous if and only if for all directed subsets S_i of S, $f(lub(S_i)) = lub(\{f(x) : x \in S_i\})$.

That is, if a directed set S_i has a least upper bound $lub(S_i)$, then the set obtained by applying a continuous function f to the elements of S_i has least upper bound $f(lub(S_i))$. Functions that are both monotonic and continuous on some partially ordered set $< S, \preceq >$ are of interest to us because they can be used, for some kinds of orderings, to guarantee that for some $s \in S$, $f(s) = s$. That is, f is said to have a "fixpoint".

Lattices

A lattice is just a partially ordered set $< S, \preceq >$ in which every pair of elements $a, b \in S$ has a glb (represented by $\sqcap$) and a lub (represented by $\sqcup$). From the definitions of lower and upper bounds, we are able to show that in any such partially ordered set, the operations will have the following properties:

- $a \sqcap b = b \sqcap a$, and $a \sqcup b = b \sqcup a$ (that is, they are are commutative).
- $a \sqcap (b \sqcap c) = (a \sqcap b) \sqcap c$, and $a \sqcup (b \sqcup c) = (a \sqcup b) \sqcup c$ (that is, they are associative).
- $a \sqcap (a \sqcup b) = a$, and $a \sqcup (a \sqcap b) = a$ (that is, they are "absorptive").
- $a \sqcap b = a$ and $a \sqcup b = b$.

We will not go into all the proofs here, but show one for illustration. Since $a \sqcap b$ is the glb of a and b, $a \sqcap b \preceq a$. Clearly then $a \sqcup (a \sqcap b)$, which is the lub of a and $a \sqcap b$, is a. This is one of the absorptive properties above. You should also be able to see, from these properties, that a lattice can also be seen simply as an algebra with two binary operations $\sqcap$ and $\sqcup$ that are commutative, associative and absorptive.

Theorem 3 *A lattice is an algebra with the binary operations of* $\sqcup$ *and* $\sqcap$.

Here is an example of a lattice: let S be all the subsets of $\{a, b, c\}$, and for $X, Y \in S$, $X \preceq Y$ means $X \subseteq Y$, $X \sqcap Y = X \cap Y$ and $X \sqcup Y = X \cup Y$. Then $< S, \subseteq >$ is a lattice. The empty set $\emptyset$ is the zero element, and S is the unity element of the lattice. More generally, a lattice that has a zero or least element (which we will denote $\bot$), and a unity or greatest element (which we will denote $\top$) is called a *bounded* lattice. In such lattices, the following necessarily hold: $a \sqcup \top = \top$; $a \sqcap \top = a$; $a \sqcup \bot = a$; and $a \sqcap \bot = \bot$. A little thought should convince you that a finite lattice will always be bounded: if the lattice is the set $S = \{a_1, \ldots, a_n\}$ then $\top = a_1 \sqcup \cdots \sqcup a_n$ and $\bot = a_1 \sqcap \cdots \sqcap a_n$. (But, does the reverse hold: will a bounded lattice always be finite?)

Two properties of subsets of lattices are of interest to us. First, a subset M of a lattice L is called a *sublattice* of L if M is also closed under the same binary operations of $\sqcup$ and $\sqcap$ defined for L (that is, M is a lattice with the same operations as those of L). Second, if a lattice L has the property that every subset of L has a lub and a glb, then the L is said to be a *complete* lattice. Clearly, every finite lattice is complete. Further, since every subset of

L has a lub and a glb, this must certainly be true of L itself. So, L has a lub, which must necessarily be the greatest element of L. Similarly, L has a glb, which must necessarily be the least element of L. In fact, the elements of L are ordered in such a way that each element is on some path from $\top$ to $\bot$ in the Hasse diagram. An example of an ordered set that is always a complete lattice is the set of all subsets of a set S, ordered by $\subseteq$, with binary operations $\cap$ and $\cup$ for the glb and lub. This set, the "powerset" of S, is often denoted by 2^S. So, if $S = \{a,b,c\}$, 2^S is the set $\{\emptyset, \{a\}, \{b\}, \{c\}, \{a,b\}, \{a,c\}, \{b,c\}, \{a,b,c\}\}$. Clearly, every subset of s of 2^S has both a glb and a lub in S.

There are two important results concerning complete lattices and functions defined on them. The Knaster-Tarski Theorem tells us that every monotonic function on a complete lattice $< S, \preceq >$ has a least fixpoint.

Theorem 4 *Let $< S, \preceq >$ be a complete lattice and let $f : S \rightarrow S$ be a monotonic function. Then the set of fixed points of f in L is also a complete lattice $< P, \preceq >$ (which obviously means that f has a greatest as well as a least fixpoint).*

Proof Sketch:[2] Let $D = \{x | x \preceq f(x)\}$. From the very definition of D, it follows that every fixpoint is in D. Consider some $x \in D$. Then because f is monotone we have $f(x) \preceq f(f(x))$. Thus,

$$\forall\ x \in D,\ f(x) \in D \tag{1.1}$$

Let $u = lub(D)$ (which should exist according to our assumption that $< S, \preceq >$ is a complete lattice. Then $x \preceq u$ and $f(x) \preceq f(u)$, so $x \preceq f(x) \preceq f(u)$. Therefore $f(u)$ is an upper bound of D. However, u is the least upper bound, hence $u \preceq f(u)$, which in turn implies that, $u \in D$. From (1.1), it follows that $f(u) \in D$. From $u = lub(D)$, $f(u) \in D$ and $u \preceq f(u)$, it follows that $f(u) = u$. Because every fixpoint is in D we have that u is the greatest fixpoint of f. Similarly, it can be proved that if $E = \{x | f(x) \preceq x\}$, then $v = glb(E)$ is a fixed point and therefore the smallest fixpoint of f. $\Box$

Kleene's First Recursion Theorem tells us how to find the element $s \in S$ that is the least fixpoint, by incrementally constructing lubs starting from applying a continuous function to the least element of the lattice ($\bot$).

Theorem 5 *Let S be a complete partial order and let $f : S \rightarrow S$ be a continuous (and therefore monotone) function. Then the least fixed point of f is the supremum of the ascending Kleene chain of f:*

$$\bot \preceq f(\bot) \preceq f(f(\bot)) \preceq \ldots \preceq f^n(\bot) \preceq \ldots$$

In the special case that $\preceq$ is $\subseteq$, the incremental procedure starts with the empty set $\emptyset$, and progressive lub's are obtained by application of the set-union operation $\cup$. We will not give the proofs of this result here.

[2]Can you complete the proof?

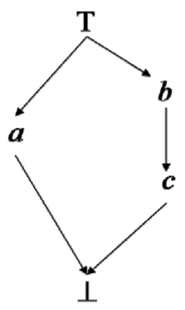


Figure 1.3: Example lattice for illustrating the concept of lattice length.

A final concept we will need is the concept of the length of a lattice. For a pair of elements a, b in a lattice L such that $a \preceq b$, the interval $[a, b]$ is the set $\{x : x \in L, \quad a \preceq x \preceq b\}$. Now, consider a subset of $[a, b]$ that contains both a and b, and is such that any pair of elements in the subset are comparable. Then that subset is a chain from a to b: if the number of elements in the subset is n, then the length of the chain is $n - 1$. Maximal chains from a to b are those of the form $a = x_1 \prec x_2 \prec \cdots \prec x_n = b$ such that each x_i is covered by x_{i+1}. If all maximal chains from a to b are finite, then the longest of these defines the length of the interval $[a, b]$. For a bounded lattice, the length of the interval $[\perp, \top]$ defines the length of the lattice. So, in the lattice in Figure 1.3, there are two maximal chains between $\perp$ and $\top$, of lengths 2 and 3 (what are these?). The length of lattice is thus equal to 3. Now, it should be evident that finite lattices will always have a finite length, but it is possible for lattices to have a finite length, but have infinitely many elements. For example, the lattice $L = \{\perp, \top, x_1, x_2, \ldots\}$ such that $\perp \prec x_i \prec \top$ has a finite length (all maximal chains are of length 2). (Indeed, it is even possible to have an infinite set in which maximal chains are of finite, but increasing in lengths of $1, 2 \ldots$.)

Quasi-Orders

A quasi-order Q in a set S is a binary relation over S that satisfies the following properties:

Reflexive. For every $a \in S$, aQa

Transitive. If aQb and bQc then aQc

You can see that a quasi-order differs from an equivalence relation in that symmetry is not required. Further, it differs from a partial order because no equality is defined, and therefore the property of anti-symmetry property cannot be defined either. There are two important properties of quasi-orders, which we will present here without proof:

- If a quasi-order Q is defined on a set $S = \{a, b, \ldots\}$, and we define a binary relation E as follows: aEb iff aQb and bQa. Then E is an equivalence relation.
- Let E partition S into subsets $X, Y, \ldots$ of equivalent elements. Let $T = \{X, Y, \ldots\}$ and $\preceq$ be a binary relation in T meaning $X \preceq Y$ in T if and only if xQy in S for some $x \in X, y \in Y$. Then T is partially ordered by $\preceq$.

What these two properties say is simply this:

> A quasi-order order Q over a set S results in a partial ordering over a set of equivalence classes of elements in S.

1.2 Logic

Logic, the study of arguments and 'correct reasoning', has been with us for at least the better part of two thousand years. In Greece, we associate its origins with Aristotle (384 B.C.–322 B.C.); in India with Gautama and the Nyaya school of philosophy (3$^{\text{rd}}$ Century B.C.?); and in China with Mo Ti (479 B.C.–381 B.C.) who started the Mohist school. Most of this dealt with the use and manipulation of *syllogisms*. It would only be a small injustice to say that little progress was made until Gottfried Wilhelm von Leibniz (1646–1716). He made a significant advance in the use of logic in mathematics by introducing symbols to represent statements and relations. Leibniz hoped to reduce all errors in human reasoning to minor calculational mistakes. Later, George Boole (1815–1864) established the connection between logic and sets, forming the basis of *Boolean algebra*. This link was developed further by John Venn (1834–1923) and Augustus de Morgan (1806–1872). It was around this time that Charles Dodgson (1832–1898), writing under the pseudonym Lewis Carroll, wrote a number of popular logic textbooks. Fundamental changes in logic were brought about by Friedrich Ludwig Gottlob Frege (1848–1925), who strongly rejected the idea that the laws of logic are synonymous with the laws of thought. For Frege, the former were laws of *truth*, having little to say on the processes by which human beings represent and reason with reality. Frege developed a logical framework that incorporated propositions with relations and the validity of arguments depended on the relations involved. Frege also introduced the device of *quantifiers* and *bound variables*, thus laying the basis for *predicate logic*, which forms the basis of all modern logical systems. All this and more is described by Bertrand Russell (1872–1970) and Alfred North Whitehead (1861–1947) in their monumental work, *Principia Mathematica*. And then in 1931, Kurt Gödel (1906–1978) showed much to the dismay of mathematicians everywhere that formal systems of arithmetic would remain incomplete.

Rational agents require knowledge of their world in order to make rational decisions. With the help of a declarative (knowledge representation) language, this knowledge (or a portion of it) is represented and stored in a knowledge base. A knowledge-base is composed of sentences in a language with a truth theory (logic), so that someone external to the system can interpret sentences as statements about the world (semantics). Thus, to express knowledge, we need a precise, *declarative* language. By a *declarative language*, we mean that

1. The system believes a statement S *iff* it considers S to be true, since one cannot believe S without an idea of what it means for the world to fulfill S.
2. The knowledge-based must be precise enough so that we must know, (1) which symbols represent sentences, (2) what it means for a sentence to be true, and (3) when a sentence follows from other sentences.

Two declarative languages will be discussed in this chapter: (0 order or) propositional logic and first order logic.

1.3 Propositional Logic

Formal logic is concerned with statements of fact, as opposed to opinions, commands, questions, exclamations *etc.* Statements of fact are assertions that are either true or false, the simplest form of which are called *propositions*. Here are some examples of propositions:

The earth is flat.
Humans are monkeys.
1 + 1 = 2

At this stage, we are not saying anything about whether these are true or false: just that they are sentences that are one or the other. Here are some examples of sentences that are not propositions:

Who goes there?
Eat your broccoli.
This statement is false.

It is normal to represent propositions by letters like $P, Q, \ldots$. For example, P could represent the proposition 'Humans are monkeys.' Often, simple statements of fact are insufficient to express complex ideas. *Compound statements* can be combining two or more propositions with *logical connectives* (or simply, connectives). The connectives we will look at here will allow us to form sentences like the following:

It is $\underline{\text{not}}$ the case that P
P $\underline{\text{and}}$ Q
P $\underline{\text{or}}$ Q
P $\underline{\text{if}}$ Q

The P's and Q's above are propositions, and the words underlined are the connectives. They have special symbols and names when written formally:

Statement	Formally	Name
It is not the case that P	$\neg P$	Negation
P and Q	$P \wedge Q$	Conjunction
P or Q	$P \vee Q$	Disjunction
P if Q	$P \leftarrow Q$	Conditional

There is, for example, a form of argument known to logicians as the *disjunctive syllogism*. Here is one due to the Stoic philosopher Chrysippus, about a dog chasing a rabbit. The dog arrives at a fork in the road, sniffs at one path and then dashes down the other. Chrysippus used formal logic to describe this:[3]

Statement	Formally
The rabbit either went down Path A or Path B.	$P \vee Q$
It did not go down Path A.	$\neg P$
Therefore it went down Path B.	$\therefore Q$

Here P represents the proposition 'The rabbit went down Path A' and Q the proposition 'The rabbit went down Path B.' To argue like Chrysippus requires us to know how to write correct logical sentences, ascribe truth or falsity to propositions, and use these to derive valid consequences. We will look at all these aspects in the sections that follow.

1.3.1 Syntax

Every language needs a *vocabulary*. For the language of propositional logic, we will restrict the vocabulary to the following:

Propositional symbols:	$P, Q, \ldots$
Logical connectives[4] **:**	$\neg, \wedge, \vee, \leftarrow$
Brackets:	$(,)$

The next step is to specify the rules that decide how legal sentences are to be formed within the language. For propositional logic, legal sentences or *well-formed formulæ* (wffs for short) are formed using the following rules:

1. Any propositional symbol is a wff;
2. If α is a wff then $\neg\alpha$ is a wff; and

[3]There is no suggestion that the principal agent in the anecdote employed similar means of reasoning.

3. If α and β are wffs then $(\alpha \wedge \beta), (\alpha \vee \beta)$, and $(\alpha \leftarrow \beta)$ are wffs.

Wffs consisting simply of propositional symbols (Rule 1) are sometimes called *atomic* wffs and others *compound* wffs Informally, it is acceptable to drop outermost brackets. Here are some examples of wffs and 'non-wffs':

Formula	Comment
$(\neg P)$	Not a wff. Parentheses are only allowed with the connectives in Rule 3
$\neg\neg P$	P is wff (Rule 1), $\neg P$ is wff (Rule 2), $\therefore\ \neg\neg P$ is wff (Rule 2)
$(P \leftarrow (Q \wedge R))$	P, Q, R are wffs (Rule 1), $\therefore\ (Q \wedge R)$ is a wff (Rule 3), $\therefore\ (P \leftarrow (Q \wedge R))$ is a wff (Rule 3)
$P \leftarrow (Q \wedge R)$	Not a wff, but acceptable informally
$((P) \wedge (Q))$	Not a wff. Parentheses are only allowed with the connectives in Rule 3
$(P \wedge Q \wedge R)$	Not a wff. Rule 3 only allows two symbols within a pair of brackets

One further kind of informal notation is widespread and quite readable. The conditional $(P \leftarrow ((Q_1 \wedge Q_2) \ldots Q_n)))$ is often written as $(P \leftarrow Q_1, Q_2, \ldots Q_n)$ or even $P \leftarrow Q_1, Q_2, \ldots Q_n$.

It is one thing to be able to write legal sentences, and quite another matter to be able to assess their truth or falsity. This latter requires a knowledge of semantics, which we shall look at shortly.

Normal Forms

Every formulae in propositional logic is equivalent to a formula that can be written as a conjunction of disjunctions. That is, something like $(A \vee B) \wedge (C \vee D) \wedge \cdots$. When written in this way the formula is said to be in *conjunctive normal form* or CNF. There is another form, which consists of a disjunction of conjunctions, like $(A \wedge B) \vee (C \wedge D) \vee \cdots$, called the *disjunctive normal form* or DNF. In general, a formula F in CNF can be written somewhat more cryptically as:

$$F = \bigwedge_{i=1}^{n} \left(\bigvee_{j=1}^{m} L_{i,j} \right)$$

and a formula G in DNF as:

$$G = \bigvee_{i=1}^{n} \left(\bigwedge_{j=1}^{m} L_{i,j} \right)$$

Here, $\bigvee F_i$ is short for $F_1 \vee F_2 \vee \cdots$ and $\bigwedge F_i$ is short for $F_i \wedge F_2 \wedge \cdots$. In both CNF and DNF forms above, the $L_{i,j}$ are either propositions or negations of propositions (we shall shortly call these "literals").

1.3.2 Semantics

There are three important concepts to be understood in the study of semantics of well-formed formulæ: *interpretations*, *models*, and *logical consequence*.

Interpretations

For propositional logic, an *interpretation* is simply an assignment of either *true* or *false* to all propositional symbols in the formula. For example, given the wff $(P \leftarrow (Q \wedge R))$ here are two different interpretations:

	P	Q	R
I_1 :	true	false	true
I_2 :	false	true	true

You can think of I_1 and I_2 as representing two different 'worlds' or 'contexts'. After a moment's thought, it should be evident that for a formula with N propositional symbols, there can never be more than 2^N possible interpretations.

Truth or falsity of a wff only makes sense given an interpretation (by the principle of bivalence, any interpretation can only result in a wff being either *true* or *false*). Clearly, if the wff simply consists of a single propositional symbol (recall that this was called an atomic wff), then the truth-value is simply that given by the interpretation. Thus, the wff P is *true* in interpretation I_1 and *false* in interpretation I_2. To obtain the truth-value of compound wffs like $(P \leftarrow (Q \wedge R))$ requires a knowledge the semantics of the connectives. These are usually summarised in a tabular form known as *truth tables*. The truth tables for the connectives of interest to us are given below.

Negation. Let α be a wff[5]. Then the truth table for $\neg\alpha$ is as follows:

α	$\neg\alpha$
false	true
true	false

Conjunction. Let α and β be wffs. The truth table for $(\alpha \wedge \beta)$ is as follows:

[5] We will use Greek characters like α, β to stand generically for any wff.

α	β	$(\alpha \wedge \beta)$
false	false	false
false	true	false
true	false	false
true	true	true

Disjunction. Let α and β be wffs. The truth table for $(\alpha \vee \beta)$ is as follows:

α	β	$(\alpha \vee \beta)$
false	false	false
false	true	true
true	false	true
true	true	true

Conditional. Let α and β be wffs. The truth table for $(\alpha \leftarrow \beta)$ is as follows:

α	β	$(\alpha \leftarrow \beta)$
false	false	true
false	true	false
true	false	true
true	true	true

We are now in a position to obtain the truth-value of a compound wff. The procedure is straightforward: given an interpretation, we find the truth-values of the smallest 'sub-wffs' and then use the truth tables for the connectives to obtain truth-values for increasingly complex sub-wffs. For $(P \leftarrow (Q \wedge R))$ this means:

1. First, obtain the truth-values of P, Q, R using the interpretation;
2. Next, obtain the truth-value of $(Q \wedge R)$ using the truth table for 'Conjunction' and the truth-values of Q and R (Step 1); and
3. Finally, obtain the truth-value $(P \leftarrow (Q \wedge R))$ using the truth table for 'Conditional' and the truth-values of P (Step 1) and $(Q \wedge R)$ (Step 2).

For the interpretations I_1 and I_2 earlier these truth-values are as follows:

	P	Q	R	$(Q \wedge R)$	$(P \leftarrow (Q \wedge R))$
I_1 :	true	false	true	false	true
I_2 :	false	true	true	true	false

Thus, $(P \leftarrow (Q \wedge R))$ is *true* in interpretation I_1 and *false* in I_2.

Models

Every interpretation (that is, an assignment of truth-values to propositional symbols) that makes a well-formed formula *true* is said to be a *model* for that formula. Take for example, the two interpretations I_1 and I_2 above. We have already seen that I_1 is a model for $(P \leftarrow (Q \wedge R))$; and that I_2 is not a model for the same formula. In fact, I_1 is also a model for several other wffs like: P, $(P \wedge R)$, $(Q \vee R)$, $(P \leftarrow Q)$, *etc.* Similarly, I_2 is a model for Q, $(Q \wedge R)$, $(P \vee Q)$, $(Q \leftarrow P)$, *etc.*

As another example, let $\{P, Q, R\}$ be the set of all atoms in the language, and α be the formula $((P \wedge Q) \leftrightarrow (R \rightarrow Q))$. Let I be the interpretation that makes P and R true, and Q false (so $I = \{P, R\}$). We determine whether α is true or false under I as follows:

1. P is true under I, and Q is false under I, so $(P \wedge Q)$ is false under I.
2. R is true under I, Q is false under I, so $(R \rightarrow Q)$ is false under I.
3. $(P \wedge Q)$ and $(R \rightarrow Q)$ are both false under I, so α is true under I.

Since α is true under I, I is a model of α. Let $I' = \{P\}$. Then $(P \wedge Q)$ is false, and $(R \rightarrow Q)$ is true under I'. Thus α is false under I', and I' is not a model of α.

The definition of model can be extended to a set of formulæ; an interpretation I is said to be a model of a set of formulæ Σ if I is a model of all formulæ $\alpha \in \Sigma$. Σ is then said to have I as a model. We will offer an example to illustrate this extended definition. Let $\Sigma = \{P, (Q \vee I), (Q \rightarrow R)\}$, and let $I = \{P, R\}$, $I' = \{P, Q, R\}$, and $I" = \{P, Q\}$ be interpretations. I and I' satisfy all formulas in Σ, so I and I' are models of Σ. On the other hand, $I"$ falsifies $(Q \rightarrow R)$, so I" is not a model of Σ.

At this point, we can distinguish amongst two kinds of formulæ:

1. A wff may be such that *every* interpretation is a model. An example is $(P \vee \neg P)$. Since there is only one propostional symbol involved (P), there are at most $2^1 = 2$ interpretations possible. The truth table summarising the truth-values for this formula is:

	P	$\neg P$	$(P \vee \neg P)$
I_1 :	false	true	true
I_2 :	true	false	true

   $(P \vee \neg P)$ is thus *true* in every possible 'context'. Formulæ like these, for which every interpretation is a model are called *valid* or *tautologies*

2. A wff may be such that *none* of the interpretations is a model. An example is $(P \wedge \neg P)$. Again there is only one propostional symbol involved (P), and thus only two interpretations possible. The truth table summarising the truth-values for this formula is:

	P	$\neg P$	$(P \wedge \neg P)$
I_1 :	false	true	false
I_2 :	true	false	false

$(P \wedge \neg P)$ is thus *false* in every possible 'context'. Formulæ like these, for which none of the interpretations is a model are called *unsatisfiable* or *inconsistent*

Finally, any wff that has at least *one* interpretation as a model is said to be *satisfiable*.

Logical Consequence

We are often interested in establishing the truth-value of a formula given that of some others. Recall the Chrysippus argument:

Statement	Formally
The rabbit either went down Path A or Path B.	$P \vee Q$
It did not go down Path A.	$\neg P$
Therefore it went down Path B.	$\therefore Q$

Here, we want to establish that if the first two statements are true, then the third follows. The formal notion underlying all this is that of *logical consequence*. In particular, what we are trying to establish is that some well-formed formula α is the *logical consequence* of a conjunction of other well-formed formulæ Σ (or, that Σ *logically implies* α). This relationship is usually written thus:

$$\Sigma \models \alpha$$

Σ being the conjunction of several wffs, it is itself a well-formed formula[6]. Logical consequence can therefore also be written as the following relationship between a pair of wffs:

$$((\beta_1 \wedge \beta_2) \dots \beta_n) \models \alpha$$

It is sometimes convenient to write Σ as the set $\{\beta_1, \beta_2, \dots, \beta_n\}$ which is understood to stand for the conjunctive formula above. But how do we determine if this relationship between Σ and α does indeed hold? What we want is the following: whenever the statements in Σ are true, α must also be true. In formal terms, this means: $\Sigma \models \alpha$ *if every model of* Σ *is also model of* α. Decoded:

[6]There is therefore nothing special needed to extend the concepts of validity and unsatisfiability to conjunctions of formulæ like Σ. Thus, Σ is valid if and only if every interpretation is a model of the conjunctive wff (in other words, a model for each wff in the conjunction); and it is unsatisfiable if and only if none of the interpretations is a model of the conjunctive wff. It should be apparent after some reflection that if Σ is valid, then all logical consequences of it are also valid. On the other hand, if Σ is unsatisfiable, then any well-formed formula is a logical consequence.

- Recall that a model for a formula is an interpretation (assignment of truth-values to propositions) that makes that formula *true*;
- Therefore, a model for Σ is an interpretation that makes $((\beta_1 \wedge \beta_2) \ldots \beta_n)$ *true*. Clearly, such an interpretation will make each of $\beta_1, \beta_2, \ldots, \beta_n$ *true*;
- Let $I_1, I_2, \ldots, I_k$ be all the interpretations that satisfy the requirement above: that is, each is a model for Σ and there are no other models for Σ (recall that if there are N propositional symbols in Σ and α together, then there can be no more than 2^N such interpretations);
- Then to establish $\Sigma \models \alpha$, we have to check that each of $I_1, I_2, \ldots, I_k$ is also a model for α (that is, each of them make α *true*).

The definition of logical entailment can be extended to the entailment of sets of formulæ. Let Σ and Γ be sets of formulas. Then Γ is said to be a logical consequence of Σ (written as $\Sigma \models \Gamma$), if $\Sigma \models \alpha$, for every formula $\alpha \in \Gamma$. We also say Σ (logically) implies Γ.

We are now in a position to see if Chrysippus was correct. We wish to see if $((P \vee Q) \wedge \neg P) \models Q$. From the truth tables on page 17, we can construct a truth table for $((P \vee Q) \wedge \neg P)$:

	P	Q	$(P \vee Q)$	$\neg P$	$((P \vee Q) \wedge \neg P)$
I_1 :	false	false	false	true	false
I_2 :	false	true	true	true	true
I_3 :	true	false	true	false	false
I_4 :	true	true	true	false	false

It is evident that of the four interpretations possible only one is a model for $((P \vee Q) \wedge \neg P)$, namely: I_2. Clearly I_2 is also a model for Q. Therefore, every model for $((P \vee Q) \wedge \neg P)$ is also a model for Q[7]. It is therefore indeed true that $((P \vee Q) \wedge \neg P) \models Q$. In fact, you will find you can 'move' formulæ from left to right in a particular manner. Thus if:

$$((P \vee Q) \wedge \neg P) \models Q$$

then the following also hold:

$$(P \vee Q) \models (Q \leftarrow \neg P) \quad \text{and} \quad \neg P \models (Q \leftarrow (P \vee Q))$$

These are consequences of a a more general result known as the *deduction theorem*, which we look at now. Using a set-based notation, let $\Sigma = \{\beta_1, \beta_2, \ldots, \beta_i, \ldots, \beta_n\}$. Then, the deduction theorem states:

[7]Although I_4 is also a model for Q, the test for logical consequence only requires us to examine those interpretations that are models of $((P \vee Q) \wedge \neg P)$. This precludes I_4.

Theorem 6

$$\Sigma \models \alpha \text{ if and only if } \Sigma - \{\beta_i\} \models (\alpha \leftarrow \beta_i)$$

.

Proof: Consider first the case that $\Sigma \models \alpha$. That is, every model of Σ is a model of α. Now assume $\Sigma - \{\beta_i\} \not\models (\alpha \leftarrow \beta_i)$. That is, there is some model, say M, of $\Sigma - \{\beta_i\}$ that is not a model of $(\alpha \leftarrow \beta_i)$. That is, β_i is true and α is false in M. That is M is a model for $\Sigma - \{\beta_i\}$ and for β_i, but not a model for α. In other words, M is a model for Σ but not a model for α which is not possible. Therefore, if $\Sigma \models \alpha$ then $\Sigma - \{\beta_i\} \not\models (\alpha \leftarrow \beta_i)$. Now for the "only if" part. That is, let $\Sigma - \{\beta_i\} \models (\alpha \leftarrow \beta_i)$. We want to show that $\Sigma \models \alpha$. Once again, let us assume the contrary (that is, $\Sigma \not\models \alpha$. This means there must be a model M for Σ that is not a model for α. However, since $\Sigma = \{\beta_1, \beta_2, \ldots, \beta_i, \ldots, \beta_n\}$, M is both a model for $\Sigma - \{\beta_i\}$ and a model for each of the β_i. So, M cannot be a model for $(\alpha \leftarrow \beta_i)$. We are therefore in a position that there is a model M for $\Sigma - \{\beta_i\}$ that is not a model of $(\alpha \leftarrow \beta_i)$, which contradicts what was given. □

The deduction theorem isn't restricted to propositional logic, and holds for first-order logic as well. It can be invoked repeatedly. Here is an example of using it twice:

$$\Sigma \models \alpha \text{ if and only if } \Sigma - \{\beta_i, \beta_j\} \models (\alpha \leftarrow (\beta_i \wedge \beta_j))$$

With Chrysippus, applying the deduction theorem twice results in:

$$\{(P \vee Q), \neg P\} \models Q \text{ if and only if } \emptyset \models (Q \leftarrow ((P \vee Q) \wedge \neg P))$$

If $\emptyset \models (Q \leftarrow ((P \vee Q) \wedge \neg P))$ then every model for $\emptyset$ must be a model for $(Q \leftarrow ((P \vee Q) \wedge \neg P))$. By convention, every interpretation is a model for $\emptyset$[8]. It follows that every interpretation must be a model for $(Q \leftarrow ((P \vee Q) \wedge \neg P))$. Recall that this is just another way of stating that $(Q \leftarrow ((P \vee Q) \wedge \neg P))$ is valid (page 19)[9].

What is the difference between the concepts of logical consequence denoted by $\models$ and the connective $\rightarrow$ in a statement such as $\Sigma \models \Gamma$? where, $\Sigma = \{(P \wedge Q), (P \rightarrow R)\}$ and $\Gamma = \{P, Q, R\}$? And how do these two notions of implication relate to the phrase 'if....then', often used in propositions or theorems? We delineate the differences below:

[8]That is, we take the empty set to denote a distinguished proposition $True$ that is *true* in every interpretation. Correctly then, the formula considered is not $((\beta_1 \wedge \beta_2) \ldots \beta_n)))$ but $(True \wedge ((\beta_1 \wedge \beta_2) \ldots \beta_n))))$.

[9]To translate declarative knowledge into action (as in the case of the dog from Chrysippus's anecdote), one of two possible strategies can be adopted. The first is called 'Negative selection' which involves *excluding any provably futile* actions. The second is called 'Positive selection' which involves *suggesting only actions that are provably safe.* There can be some actions that are neither provably safe nor provably futile.

1. The connective $\rightarrow$ is a *syntactical symbol* called ‘if ... then’ or ‘implication’, which appears within formulæ. The truth value of the formula $(\alpha \rightarrow \xi)$ depends on the *particular* interpretation I we happen to be considering: according to the truth table, $(\alpha \rightarrow \xi)$ is true under I if α is false under I and/or ξ is true under I; $(\alpha \rightarrow \xi)$ is false otherwise.

2. The concept of ‘logical consequence’ or ‘(logical) implication’, denoted by ’$\models$’ describes a *semantical relation* between formulæ. It is defined in terms of *all* interpretations: ’$\alpha \models \xi$’ is true if every interpretation that is a model of α, is also a model of ξ.

3. The phrase ’if. .. then’, which is used when stating, for example, propositions or theorems is also sometimes called ’implication’. This describes a relation between assertions which are phrased in (more or less) natural language. It is used for instance in proofs of theorems, when we state that some assertion implies another assertion. Sometimes we use the symbols ’ ’ or . ¡=’ for this. If assertion A implies assertion B, we say that B is a necessary condition for A (i.e., if A is true, B must necessarily be true), and A is a. Ilufficient condition for B (i.e., the truth of B is sufficient to make A true). In (’tum A implies B, and B implies A, we write ”A iff B”, where ’iff’ abbreviates ’if, and only if’.

Closely related to logical consequence is the notion of *logical equivalence*. A pair of wffs α and β are logically equivalent means:

$$\alpha \models \beta \quad \textit{and} \quad \beta \models \alpha$$

This means the truth values for α and β are the same in all cases, and is usually written more concisely as:

$$\alpha \equiv \beta$$

Examples of logically equivalent formulæ are provided by De Morgan’s laws:

$$\neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$$

$$\neg(\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta$$

Also, if $True$ denotes the formula that is *true* in every interpretation and $False$ the formula that is *false* in every interpretation, then the following equivalences should be self-evident:

$$\alpha \equiv (\alpha \wedge True)$$

$$\alpha \equiv (\alpha \vee False)$$

More on the Conditional

We are mostly concerned with rules that utilise the logical connective $\leftarrow$. This makes this particular connective more interesting than the others, and it is worth noting some further details about it. Although we will present these here using examples from the propositional logic, the main points are just as applicable to formulæ in the predicate logic.

Recall the truth table for the conditional from page 18:

α	β	$(\alpha \leftarrow \beta)$
false	false	true
false	true	false
true	false	true
true	true	true

There is, therefore, only one interpretation that makes $(\alpha \leftarrow \beta)$ *false*. This may come as a surprise. Consider for example the statement:

$$\textsf{The earth is flat} \leftarrow \textsf{Humans are monkeys}$$

An interpretation that assigns *false* to both 'The earth is flat' and 'Humans are monkeys' makes this statement *true* (line 1 of the truth table). In fact, the only world in which the statement is false is one in which the earth is not flat, and humans are monkeys[10]. Consider now the truth table for $(\alpha \vee \neg\beta)$:

α	β	$\neg\beta$	$(\alpha \vee \neg\beta)$
false	false	true	true
false	true	false	false
true	false	true	true
true	true	false	true

It is evident from these truth tables that every model for $(\alpha \leftarrow \beta)$ is a model for $(\alpha \vee \neg\beta)$ and vice-versa. Thus:

$$(\alpha \leftarrow \beta) \equiv (\alpha \vee \neg\beta)$$

Thus, the conditional:

$$(\textsf{Fred is human} \leftarrow (\textsf{Fred walks upright} \wedge \textsf{Fred has a large brain}))$$

is equivalent to:

[10]The unusual nature of the conditional is due to the fact that it allows premises and conclusions to be completely unrelated. This is not what we would expect from conditional statements in normal day-to-day discourse. For this reason, the $\leftarrow$ connective is sometimes referred to as the *material conditional* to distinguish it from a more intuitive notion.

$$(\textsf{Fred is human} \vee \neg\, (\textsf{Fred walks upright} \wedge \textsf{Fred has a large brain}))$$

Or, using De Morgan's Law (page 23) and dropping some brackets for clarity:

$$\textsf{Fred is human} \vee \neg\, \textsf{Fred walks upright} \vee \neg\, \textsf{Fred has a large brain}$$

In this form, each of the premises on the right-hand side of the the original conditional (Fred walks upright, Fred has a large brain) appear negated in the final disjunction; and the conclusion (Fred is human) is unchanged. For a reason that will become apparent later we will use the term *clauses* to denote formulæ that contain propositions or negated propositions joined together by disjunction ($\vee$). We will also use the term *literals* to denote propositions or negated propositions. Clauses are thus disjunctions of literals.

It is common practice to represent a clause as a set of literals, with the disjunctions understood. Thus, the clause above can be written as:

$$\{\ \textsf{Fred is human},\ \neg\, \textsf{Fred walks upright},\ \neg\, \textsf{Fred has a large brain}\ \}$$

The equivalence $\alpha \leftarrow \beta \equiv \alpha \vee \neg\beta$ also provides an alternative way of presenting the deduction theorem.

On page 22 the statement of this theorem was:

$$\Sigma \models \alpha \ \text{ if and only if } \ \Sigma - \{\beta_i\} \models (\alpha \leftarrow \beta_i)$$

This can now be restated as:

$$\Sigma \models \alpha \ \text{ if and only if } \ \Sigma - \{\beta_i\} \models (\alpha \vee \neg\beta_i)$$

The theorem thus operates as follows: when a formula moves from the left of $\models$ to the right, it is negated and disjoined (using $\vee$) with whatever exists on the right. The theorem can also be used in the "other direction": when a formula moves from the right of $\models$ to the left, it is negated and conjoined (using $\wedge$ or $\cup$ in the set notation) to whatever exists on the left. Thus:

$$\Sigma \models (\alpha \vee \neg\beta) \ \text{ if and only if } \ \Sigma \cup \{\neg\alpha\} \models \neg\beta$$

A special case of this arises from the use of the equivalence $\alpha \equiv (\alpha \vee False)$ (page 23):

$$\Sigma \models \alpha \ \text{ if and only if } \ \Sigma \models (\alpha \vee False) \ \text{ if and only if } \ \Sigma \cup \{\neg\alpha\} \models False$$

The formula $False$ is commonly written as $\Box$ and the result above as:

$$\Sigma \models \alpha \ \text{ if and only if } \ \Sigma \cup \{\neg\alpha\} \models \Box$$

The conditional $(\alpha \leftarrow \beta)$ is sometimes mistaken to mean the same as $(\alpha \wedge \beta)$. Comparison against the truth table for $(\alpha \wedge \beta)$ shows that these two formulæ are not equivalent:

α	β	$(\alpha \wedge \beta)$
false	false	false
false	true	false
true	false	false
true	true	true

There are a number of ways in which $(\alpha \leftarrow \beta)$ can be translated in English. Some of the more popular ones are:

If β, then α	α, if β	β implies α
β only if α	β is sufficient for α	α is necessary for β
All β's are α's		

Note the following related statements:

Conditional	$(\alpha \leftarrow \beta)$
Contrapositive	$(\neg\beta \leftarrow \neg\alpha)$

It should be easy to verify the following equivalence:

Conditional ≡ Contrapositive $\quad (\alpha \leftarrow \beta) \equiv (\neg\beta \leftarrow \neg\alpha)$

Errors of reasoning arise by assuming other equivalences. Consider for example the pair of statements:

S_1 : Fred is an ape ← Fred is human
S_2 : Fred is human ← Fred is an ape

S_2 is the sometimes called the *converse* of S_1. An interpretation that assigns *true* to 'Fred is an ape' and *false* to 'Fred is human' is a model for S_1 but not a model for S_2. The two statements are thus not equivalent.

More on Normal Forms

We are now able to state two properties concerning normal forms:

1. If F is a formula in CNF and G is a formula in DNF, then $\neg F$ is a formula in DNF and $\neg G$ is a formula in CNF. This is a generalisation of De Morgan's laws and can be proved using the technique of mathematical induction (that is, show truth for a formula with a single literal; assume truth for a formula with n literals; and then show that it holds for a formula with $n + 1$ literals.)

2. Every formula F can be written as a formula F_1 in CNF and a formula F_2 in DNF. It is straightforward to see that any formula F can be written as a DNF formula by examining the rows of the truth table for F for which F is true. Suppose F consists of the propositions $A_1, A_2, \ldots, A_n$. Then each such row is equivalent to some conjunction of literals $L_1, L_2, \ldots, L_n$, where L_i is equal to A_i if A_i is true in that row and equal to $\neg A_i$ otherwise. Clearly, the disjunction of each row for which F is true gives the DNF formula for F. We can get the corresponding CNF formula G by negating the DNF formula (using the property above), or by examining the rows for which F is false in the truth table.

It should now be clear that a CNF expression is nothing more than a conjunction of a set of clauses (recall a clause is simply a disjunction of literals). It is therefore possible to convert any propositional formula F into CNF—either using the truth table as described, or using the following procedure:

1. Replace all conditionals statements of the form $A \leftarrow B$ by the equivalent form using disjunction (that is, $A \vee \neg B$). Similarly replace all $A \leftrightarrow B$ with $(A \vee \neg B) \wedge (\neg A \vee B)$.

2. Eliminate double negations ($\neg\neg A$ replaced by A) and use De Morgan's laws wherever possible (that is, $\neg(A \wedge B)$ replaced by $(\neg A \vee \neg B)$ and $\neg(A \vee B)$ replaced by $(\neg A \wedge \neg B)$).

3. Distribute the disjunct $\vee$. For example, $(A \vee (B \wedge C))$ is replaced by $(A \vee B) \wedge (A \vee C)$.

An analogous process converts any formula to an equivalent formula in DNF. We should note that during conversion, formulæ can expand exponentially. However, if only satisfiability should be preserved, conversion to CNF formula can be done polynomially.

1.3.3 Inference

Enumerating and comparing models is one way of determining whether one formula is a logical consequence of another. While the procedure is straightforward, it can be tedious, often requiring the construction of entire truth tables. A different approach makes no explicit reference to truth values at all. Instead, if α is a logical consequence of Σ, then we try to show that we can *infer* or *derive* α from Σ using a set of well-understood rules. Step-by-step application of these rules results in a *proof* that deduces that α follows from Σ. The rules, called *rules of inference*, thus form a system of performing calculations with propositions, called the *propositional calculus*[11]. Logical implication can be mechanized by using a propositional calculus. We will first concentrate on a particular inference rule called *resolution*.

[11] In general, a set of inference rules (potentially including so called logical axioms) is called a *calculus*.

1.3.4 Resolution

Before proceding further, some basic terminology from proof theory may be helpful (this is not specially confined to the propositional calculus). Proof theory considers the *derivability* of formulæ, given a set of inference rules $\mathcal{R}$. Formulæ given initially are called *axioms* and those derived are *theorems*. That formula α is a theorem of a set of axioms Σ using inference rules $\mathcal{R}$ is denoted by:

$$\Sigma \vdash_{\mathcal{R}} \alpha$$

When $\mathcal{R}$ is obvious, this is simply written as $\Sigma \vdash \alpha$. The axioms can be valid (that is, all interpretations are models), or problem-specific (that is, only some interpretations may be models). The axioms together with the inference rules constitute what is called an *inference system*. The axioms together with all the theorems that are derivable from it is called a *theory*. A theory is said to be *inconsistent* if there is a formula α such that the theory contains both α and $\neg\alpha$.

We would like the theorems derived to be logical consequences of the axioms provided. For, if this were the case then by definition, the theorems will be *true* in all models of the axioms (recall that this is what logical consequence means). They will certainly be true, therefore, in any particular 'intended' interpretation of the axioms. Ensuring this property of the theorems depends entirely on the inference rules chosen: those that have this property are called *sound*. That is:

$$\text{if } \Sigma \vdash_{\mathcal{R}} \alpha \text{ then } \Sigma \models \alpha$$

Some well-known sound inference rules are:

$$\begin{array}{ll} \text{Modus ponens:} & \{\beta, \alpha \leftarrow \beta\} \;\vdash\; \alpha \\ \text{Modus tollens:} & \{\neg\alpha, \alpha \leftarrow \beta\} \;\vdash\; \neg\beta \end{array}$$

Theorems derived by the use of sound inference rules can be added to the original set of axioms. That is, given a set of axioms Σ and a theorem α derived using a sound inference rule, $\Sigma \equiv \Sigma \cup \{\alpha\}$.

We would also like to derive *all* logical consequences of a set of axioms and rules with this property at said to be *complete*:

$$\text{if } \Sigma \models \alpha \text{ then } \Sigma \vdash_{\mathcal{R}} \alpha$$

Axioms and inference rules are not enough: we also need a strategy to select and apply the rules. An inference system (that is, axioms and inference rules) along with a strategy is called a *proof procedure*. We are especially interested here in a special inference rule called *resolution* and a strategy called SLD (the meaning of this is not important at this point: we will come to it later). The result is a proof procedure called *SLD-resolution*. Here we will simply illustrate the rule of resolution for manipulating propositional formulæ, and use an unconstrained proof strategy. A description of SLD will be left for a later section.

Suppose we are given as axioms the conditional formulæ (using the informal notation that replaces $\wedge$ with commas):

β_1 : Fred is an ape $\leftarrow$ Fred is human

β_2 : Fred is human $\leftarrow$ Fred walks upright, Fred has a large brain

Then the following is a theorem resulting from the use of resolution:

α : Fred is an ape $\leftarrow$ Fred walks upright, Fred has a large brain

That α is indeed a logical consequence of $\beta_1 \wedge \beta_2$ can be checked by constructing truth tables for the formulæ: you will find that every interpretation that makes $\beta_1 \wedge \beta_2$ *true* will also make α *true*. More generally, here is the rule of resolution when applied to a pair of conditional statements:

$$\{(P \leftarrow Q_1, \ldots, Q_i, \ldots Q_n),\ (Q_i \leftarrow R_1, \ldots R_m)\} \ \vdash \\ P \leftarrow Q_1, \ldots, Q_{i-1}, R_1, \ldots R_m, Q_{i+1}, \ldots, Q_n$$

The equivalence from page 24 ($\alpha \leftarrow \beta \equiv \alpha \vee \neg\beta$) allows resolution to be presented in a different manner (we have taken the liberty of dropping some brackets here):

$$\{(P \vee \neg Q_1 \vee \ldots \neg Q_i \vee \ldots \vee \neg Q_n),\ (Q_i \vee \neg R_1 \vee \ldots \vee \neg R_m)\} \ \vdash \\ P \vee \neg Q_1 \vee \ldots \vee \neg Q_{i-1} \vee \neg R_1 \vee \ldots \vee \neg R_m \vee \neg Q_{i+1} \vee \ldots \vee \neg Q_n$$

On page 25, we introduced the terms clauses and literals. Thus, resolution applies to a pair of 'parent' clauses that contain *complementary* literals $\neg L$ and L. The result (the 'resolvent') is a clause containing all literals from each clause, except the complementary pair. Or, more abstractly, let C_1 and C_2 be a pair of clauses, and let $L \in C_1$ and $\neg L \in C_2$. Then, the resolvent of C_1 and C_2 is the clause:

$$R = (C_1 - \{L\}) \cup (C_2 - \{\neg L\})$$

Resolution of a pair of *unit clauses*—those that contain just single literals L and $\neg L$—results in the the *empty clause*[12], or $\Box$, which means that the parent clauses were inconsistent.

We can show that resolution is a sound inference rule.

Theorem 7 *Suppose R is the resolvent of clauses C_1 and C_2. That is, $\{C_1, C_2\} \vdash R$. The resolution is sound, that is, $\{C_1, C_2\} \models R$.*

Proof:: We want to show that if C_1 and C_2 are true and R is a resolvent of C_1 and C_2 then R is true. Let us assume C_1 and C_2 are true, and that R was obtained by resolving on some literal L in C_1 and C_2. Further, let $C_1 = C \vee L$

[12]Note the difference between an empty clause $\Box$ and empty set of clauses $\{\}$. An interpretation I logically entails C *iff* there exists an $l \in C$ such that $I \models l$. I logically entails Σ if for all $C \in \Sigma$, $I \models C$. Thus, by definition, for all interpretations I, $I \not\models \Box$ and $I \not\models \{\Box\}$, whereas $I \models \{\}$.

and $C_2 = D \vee \neg L$, giving $R = C \vee D$. Now, L is either true or false. Suppose L is true. Then clearly C_1 is true, but since $\neg L$ is false and C_2 is true (by assumption), it must be that D, and hence R must be true. It is easy to see that we similarly arrive to the same conclusion about R even if L was false. $\Box$

So, the theorems obtained by applying resolution to a set of axioms are all logical consequences of the axioms. In general, we will denote a clause C derived from a set of clauses Σ using resolution by $\Sigma \vdash_R C$. This means that there is a finite sequences of clauses $R_1, \ldots, R_k = C$ such that each C_i (where C_i is a clause being resolved upon in the i^{th} resolution step) is either in Σ or is a resolvent of a pair of clauses already derived (that is, from $\{R_1, \ldots, R_{i-1}\}$) . Now, although it is the case that if $\Sigma \vdash_R \alpha$ then $\Sigma \models \alpha$, the reverse does not hold. For example, a moment's thought should convince you that:

$$\{\textsf{Fred is an ape, Fred is human }\} \models \textsf{Fred is an ape} \leftarrow \textsf{Fred is human}$$

However, using resolution, there is no way of deriving Fred is an ape ← Fred is human from Fred is an ape and Fred is human. As an inference rule, resolution is thus incomplete[13]. However, it does have an extremely useful property known as *refutation completeness*. This is that if a formula Σ is inconsistent, then the empty clause $\Box$ will be eventually derivable by resolution. We will distinguish between the two completeness by referring to general completeness as affirmation completeness.

Thus, since Fred is an ape ← Fred is human is a logical consequence of Fred is an ape and Fred is human, then the formula:

$$\Sigma : \{\textsf{Fred is an ape, Fred is human, } \neg(\textsf{Fred is an ape} \leftarrow \textsf{Fred is human})\}$$

must be inconsistent. This can be verified using resolution. First, the clausal form of (Fred is an ape ← Fred is human) is (Fred is an ape ∨ ¬ Fred is human). Using De Morgan's Law on this clausal form, we can see that ¬(Fred is an ape ← Fred is human) is equivalent to ¬ Fred is an ape ∧ Fred is human. We can now rewrite Σ :

$$\Sigma' : \{\textsf{Fred is an ape, Fred is human, } \neg\textsf{Fred is an ape}\}$$

Resolution of the pair Fred is an ape, ¬Fred is an ape would immediately result in the empty clause $\Box$. The general steps in a refutation proof procedure using resolution are therefore:

- Let S be a set of clauses and α be a propositional formula. Let $C = S \cup \{\neg\alpha\}$.
- Repeatedly do the following:
    1. Select a pair of clauses C_1 and C_2 from C that can be resolved on some proposition P.

[13]It can be however proved that resolution is affirmation complete with respect to atomic conclusions.

2. Resolve C_1 and C_2 to give R.
3. If $R = \Box$ then stop. Otherwise, if R contains both a proposition Q and its negation $\neg Q$ then discard R. Otherwise add R to C.

In general, we know that any formula F can be converted to a conjunction of clauses. We can distinguish between the following sets. $Res^0(F)$, which is simply the set of clauses in F. $Res^n(F)$, for $n > 0$, which is the clauses containing all clauses in $Res^{n-1}(F)$ and all clauses obtained by resolving a pair of clauses from $Res^{n-1}(F)$. Since there are only a finite number of propositional symbols in F and a finite number of clauses in its CNF, we can see that there will only be a finite number of clauses that can be obtained using resolution. That is, there is some m such that $Res^m(F) = Res^{m-1}(F)$. Let us call this final set consisting of all the original clauses and all possible resolvents $Res^*(F)$. Then, the property of refutation-completeness for resolution can be stated more formally as follows:

Theorem 8 *For some formula F, $\Box \in Res^*(F)$ if and only if F is unsatisfiable.*

Though the resolution rule by itself is not (affirmation) complete for clauses in general, this property states that it is complete with respect to unsatisfiable sets of clauses. The complete proof of this will be provided on page 33. To get you started however, we show that if $\Box \in Res^*(F)$ then F is unsatisfiable. We can assume that $\Box \notin Res^0(F)$, since $\Box$ is not a disjunction of literals. Therefore there must be some k for which $\Box \notin Res^k(F)$ and $\Box \in Res^{k+1}(F)$. This can only mean that both L and $\neg L$ are in $Res^k(F)$. That is L and $\neg L$ are obtained from F by resolution. By the property of soundness of resolution, this means that $F \models (L \wedge \neg L)$. That is, F is unsatisfiable.

There are also other proof processes that are refutation-complete. Examples of such processes are the Davis-Putnam Procedure[14], Tableaux Procedure, *etc.* In the worst case, the resolution search procedure can take exponential time. This, however, very probably holds for all other proof procedures. For CNF formulae in propositional logic, a type of resolution process called the Davis-Putnam Procedure (backtracking over all truth values) is probably (in practice) the fastest refutation-complete process.

The Subsumption Theorem

A property related to logical implication is that of subsumption. A propositional clause C *subsumes* a propositional clause D if $C \subseteq D$. What does this mean? It just means that every literal in C appears in D. Here are a pair of clauses C and D such that C subsumes D:

$$C : \textsf{Fred is an ape}$$

$$D : \textsf{Fred is an ape} \leftarrow \textsf{Fred is human}$$

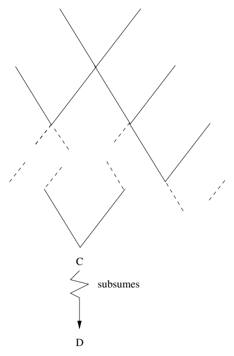
[14]It can be proved that the Davis-Putnam procedure is sound as well as complete.

In general, it should be easy to see that if C and D are clauses such that $C \subseteq D$, then $C \models D$. In fact, for propositional logic, it is also the case that if $C \models D$ then C subsumes D (we see why this is so shortly).

The notion of subsumption acts as the basis for an important result linking resolution and logical implication, called the *subsumption theorem*:

Theorem 9 *If Σ is a set of clauses and D is a clause. Then $\Sigma \models D$ if and only if D is a tautology or there is a clause C such that there is a derivation of C from Σ using resolution ($\Sigma \vdash_R C$) and C subsumes D.*

By "derivation of a clause C" here, we mean the same as on page 30, that is, there is a sequence of clauses $R_1, \ldots, R_k = C$ such that each R_i is either in Σ or is a resolvent of a pair of clauses in $\{R_1, \ldots, R_{i-1}\}$. In effect, the Subsumption Theorem tells us that logical implication can be decomposed into a sequence of resolution steps, followed by a subsumption step:



Proof of Subsumption Theorem:

We will show that "only if" part of the theorem holds using the method of induction on the size of Σ:

1. Let $|\Sigma| = 1$. That is $\Sigma = \{\alpha_1\}$. Let $\Sigma \models D$. Since the only result of applying resolution to Σ is α_1, we need to show that $\alpha_1 \subseteq D$. Suppose $\alpha_1 \not\subseteq D$. Let L be a literal in α_1 that is not in D. Let I be an interpretation that assigns L to true and all literals in D to false. Clearly I is a model for α_1 but not a model for D, which is not possible since $\Sigma \models D$. Therefore, $\alpha_1 \subseteq D$.

2. Let the theorem hold for $|\Sigma| = n$. We will see that it follows that it holds for $|\Sigma| = n + 1$. Let $\Sigma = \{\alpha_1, \ldots, \alpha_{n+1}\}$ and $\Sigma \models D$. By the Deduction Theorem, we know that this means $\Sigma - \{\alpha_{n+1}\} \models (D \leftarrow \alpha_{n+1})$, or $\Sigma - \{\alpha_{n+1}\} \models (D \vee \neg\alpha_{n+1})$. Let us set $\Sigma' = \Sigma - \{\alpha_{n+1}\}$, and let $L_1, \ldots, L_k$ be the literals in α_{n+1} that do not appear in D. That is $\alpha_{n+1} = L_1 \vee \cdots \vee L_k \vee D'$, where $D' \subseteq D$. $\Sigma' \models (D \vee \neg\alpha_{n+1})$, you should be able to see that $\Sigma' \models (D \vee \neg L_i)$ for $1 \leq i \leq k$. Since $|\Sigma'| = n$ and we

believe, by the induction hypothesis, that the subsumption theorem holds for $|\Sigma'| = n$, there must be some β_i for each L_i, such that $\Sigma' \vdash_R \beta_i$ and $\beta_i \subseteq (D \vee \neg L_i)$. Suppose $\neg L_i \notin \beta_i$. Then $\beta_i \subseteq D$, which means that β_i subsumes D. Since $\Sigma' \models \beta_i$ and $\Sigma \models \Sigma'$, the result follows. Now suppose $\neg L_i \in \beta_i$. That is, $\beta_i = \neg L_i \vee \beta_i'$, where $\beta_i' \subseteq D$. Clearly, we can resolve this with $\alpha_{n+1} = L_1 \vee \cdots \vee L_i \vee \cdots \vee L_k \vee D'$ to give $L_1 \vee \cdots \vee L_{i-1} \vee \beta_i' \vee \cdots \vee L_k \vee D'$. Progressively resolving against each of the β_i, we will be left with the clause $C = \beta_1' \vee \beta_2' \vee \cdots \vee \beta_k' \vee D'$. Since C is the result of resolutions using a clause from Σ (that is α_{n+1}) and clauses derivable from $\Sigma' \subset \Sigma$, it is evident that $\Sigma \vdash_R C$. Also, since $\beta_i' \subseteq D$ and $D' \subseteq D$, $C \subseteq D$ and the result follows.

You should be able to see that the proof in the other direction (the "if" part) follows easily enough from the soundness of resolution. $\square$

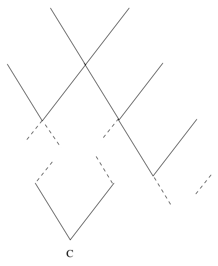
An immediate consequence of the Subsumption Theorem is that refutation-completeness of resolution follows.

Proof of Theorem 8

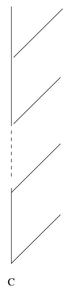
Recall what refutation completeness of resolution means: if Σ is a set of clauses that is unsatisfiable, then the empty clause $\square$ is derivable using resolution. If Σ is unsatisfiable, then $\Sigma \models \square$. From the Subsumption Theorem, we know that if $\Sigma \models \square$, there must be a clause C such that $\Sigma \vdash_R C$ and C subsumes $\square$. But the only clause subsuming $\square$ is $\square$ itself. Hence $C = \square$, which means that if $\Sigma \models \square$ then $\Sigma \vdash_R \square$.

Proofs Using Resolution

So far, we have no strategy for directing the clauses obtained using resolution. Clauses are derived using any pair of clauses with complementary literals, and the process simply continues until we find the clause we want (for example, $\square$ if we are interested in a proof of unsatisfiability). This procedure is clearly quite inefficient, since there is almost nothing constraining a proof, other than the presence of complementary literals. A "proof" for a clause C then ends up looking something like this:



Being creatures of limited patience and resources, we would like more directed approach. We can formalise this by changing our notion of a derivation. Recall what we have been using so far: the derivation of a clause C from a set of clauses Σ means there is a sequence of clauses $R_1, \ldots, R_k = C$ such that each C_i is either in Σ or is a resolvent of a pair of clauses in $\{R_1, \ldots, R_{i-1}\}$. This results in the unconstrained form of a proof for C. We will say that there is a *linear* derivation for C from Σ if there is a sequence $R_0, \ldots, R_k = C$ such that $R_0 \in \Sigma$ and each R_i $(1 \leq i \leq k)$ is a resolvent of R_{i-1} and a clause $C_i \in \Sigma \cup \{R_0, \ldots, R_{i-2}\}$. With a little thought, you should be able to convince yourself that this will result in a derivation with a “linear” look:



Here, you can see that each new resolvent forms one of the clauses for the next resolution step. The other clause—sometimes called the “side clause”—can be any one of the clauses in Σ or a previous resolvent. For reasons evident from the diagram above, the proof strategy is called linear resolution, and we will extend our notation to indicate both the inference rule and the proof strategy. Thus $\Sigma \vdash_{LR} C$ will mean that C is derived from Σ using linear resolution. We can restrict things even further, by requiring side clauses to be only from Σ. The resulting proof strategy, called *input resolution* is important as it is a generalised form of SLD resolution, first mentioned on page 28.

While the restrictions imposed by the proof strategies ensure that proofs are more directed (and hence efficient), it is important at this point to ask: at what cost? Of course, since we are still using resolution as an inference rule, the individual (and overall) inference steps remain sound. But what about completeness? By this we mean refutation-completeness, since this is the only kind of completeness we were able to show with unconstrained resolution. In fact, it is the case that linear resolution retains the property of refutation-completeness, but input resolution for arbitrary clauses does not. That input resolution is not refutation complete can be proved using a simpe counter-example:

$$
\begin{aligned}
C_0 &: \textsf{Fred is an ape} \leftarrow \textsf{Fred is human}\\
C_1 &: \textsf{Fred is an ape} \leftarrow \neg\ \textsf{Fred is human}\\
C_2 &: \neg\ \textsf{Fred is an ape} \leftarrow \textsf{Fred is human}\\
C_3 &: \neg\ \textsf{Fred is an ape} \leftarrow \neg\ \textsf{Fred is human}
\end{aligned}
$$

Now, a little effort should convince you that this set of clauses is unsatisfiable. But input resolution will simply yield a sequence of resolvents: Fred is human, Fred is an ape, Fred is human, ...

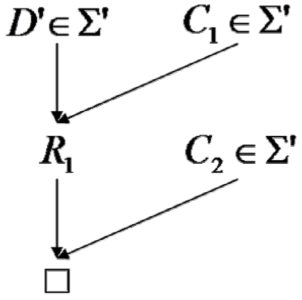


Figure 1.4: Part of case 1 of the proof for theorem 10.

Theorem 10 *If Σ is an unsatisfiable set of clauses, and $C \in \Sigma$ such that $\Sigma \backslash \{C\}$ is satisfiable, then there is a linear refutation of Σ with C as top clause.*

Proof:

We can assume Σ is finite. Let n be the number of distinct atoms occurring in literals in literals in Σ. We prove the lemma by induction on n

1. If $n = 0$, then $\Sigma = \{\square\}$. Since $\Sigma \setminus \{C\}$ is satisfiable, $C = \square$.

2. Suppose the lemma holds for $n \leq m$, and suppose $m + 1$ distinct atoms appear in Σ. We distinguish two cases.

   - *Case 1*: Suppose $C = L$, where L is a literal. We first delete all clauses from Σ which contain the literal L (so we also delete C itself from Σ). Then we replace clauses which contain the literal $\neg L$ by clauses constructed by deleting these $\neg L$ (so for example, $L_l \vee \neg L \vee L_2$ will be replaced by $L_1 \vee L_2$). Call the finite set obtained in this way Γ.

     Note that neither the literal L, nor its negation, appears in clauses in Γ. If M were a Herbrand model of Γ, then $M \cup \{L\}$ (*i.e.*, the Herbrand interpretation which makes L true, and is the same as M for other literals) would be a Herbrand model of Σ. Thus since Σ is unsatisfiable, Γ must be unsatisfiable.

     Now let Σ' be an unsatisfiable subset of Γ, such that every proper subset of Σ' is satisfiable. Σ' must contain a clause D' obtained from a member of Σ which contained $\neg L$, for otherwise the unsatisfiable set Σ' would be a subset of $\Sigma \backslash \{C\}$, contradicting the assumption that $\Sigma \setminus \{C\}$ is satisfiable. By construction of Σ', we have that $\Sigma' \setminus \{D'\}$ is satisfiable. Furthermore, Σ' contains at most m distinct atoms, so by the induction hypothesis there exists a linear refutation of Σ' with top clause D'. See the Figure 1.4 for illustration.

     Each side clause in this refutation that is not equal to a previous center clause, is either a member of Σ or is obtained from a member

$$\begin{tikzcd}
C = L \in \Sigma \arrow[d] & D = \neg L \vee D' \in \Sigma \arrow[dl] \\
D' \arrow[d] & C_1 \vee \neg L \in \Sigma \arrow[dl] \\
R_1 \vee \neg L \arrow[d] & C_2 \in \Sigma \arrow[dl] \\
\neg L \arrow[d] & L \arrow[dl] \\
\Box &
\end{tikzcd}$$

Figure 1.5: The complete picture of case 1 of the proof for theorem 10.

of Σ by means of the deletion of $\neg L$. In the latter kind of side clauses, put back the deleted $\neg L$ literals, and add these $\neg L$ to all later center clauses. Note that afterwards, these center clauses may contain multiple copies of $\neg L$. In particular, the last center clause changes from $\Box$ to $\neg L \vee \ldots \vee \neg L$. Since D' is a resolvent of C and $D = \neg L \vee D' \in \Sigma$, we can add C and D as parent clauses on top of the previous top clause D'. That way, we get a linear derivation of $\neg L \vee \ldots \vee \neg L$ from Σ, with top clause C. Finally, the literals in $\neg L \vee \ldots \vee \neg L$ can be resolved away using the top clause $C = L$ as side clause. This yields a linear refutation of Σ with top clause C (see Figure 1.5).

- *Case 2:*

  Suppose $C = L \vee C'$, where C' is a non-empty clause. C' cannot contain $\neg L$, for otherwise C would be a tautology, contradicting the assumption that Σ is unsatisfiable while $\Sigma \setminus \{C\}$ is satisfiable. Obtain Σ' from Σ by deleting clauses containing $\neg L$, and by removing the literal L from the remaining clauses. Note that $C' \in \Sigma'$. If M were a Herbrand model of Σ', then $M \cup \{\neg L\}$ is a Herbrand model of Σ. Thus since Σ is unsatisfiable, Σ' is unsatisfiable.

  Furthermore, because $\Sigma \setminus \{C\}$ is satisfiable, there is a Herbrand model M' of $\Sigma \setminus \{C\}$. Since Σ is unsatisfiable, M' is not a model of C. L is a literal in C, hence L must be false under M'. Every clause in $\Sigma' \setminus \{C'\}$ is obtained from a clause in $\Sigma \setminus C$ by deleting L from it. Since M' is a model of every clause in $\Sigma \setminus \{C\}$ and L is false under M', every clause in $\Sigma' \setminus \{C'\}$ is true under M'. Therefore M' is a model of $\Sigma' \setminus \{C'\}$, which shows that $\Sigma' \setminus \{C'\}$ is satisfiable.

  Then by the induction hypothesis, there exists a linear refutation of Σ' with top clause C'. Now similar to case 1, put back the previously deleted L literals to the top and side clauses, and to the appropriate center clauses. This gives a linear derivation of $L \vee \ldots \vee L$ from Σ with top clause C.

Note that $\{L\} \cup (\Sigma \setminus \{C\})$ is unsatisfiable, because L is false in any Herbrand model of $\Sigma \setminus \{C\}$, as shown above. On the other hand, $\Sigma \setminus \{C\}$ is satisfiable. Thus by case 1 of this proof, there exists a linear refutation of $\{L\} \cup (\Sigma \setminus \{C\})$ with top clause L. Since L is a factor of $L \vee \ldots \vee L$, we can put out linear derivation of $L \vee \ldots \vee L$ "on top" of this linear refutation of $\{L\} \cup (\Sigma \setminus \{C\})$ with top clause L, thus obtaining a linear refutation of Σ with top clause C.

$\Box$

The incompleteness of input also means that the Subsumption Theorem will not hold for input resolution in general. What, then, can we say about SLD resolution? The short answer is that it too is incomplete. But, for a restricted form of clauses, input and SLD resolution *are* complete. The restriction is to Horn clauses: recall that these are clauses that have at most 1 positive literal. Indeed, it is this restriction that forms the basis of theorem-proving in the PROLOG language, which is restricted (at least in its pure form) to Horn clauses, albeit in first-order logic (but the result still holds in that case as well).

Proofs Using SLD Resolution

Before we get to SLD, we first make our description of input resolution a little more precise: the derivation of a clause C from a set of clauses Σ using input resolution means there is a sequence of clauses $R_0, \ldots, R_k = C$ such that $R_0 \in \Sigma$ and each R_i $(1 \leq i \leq k)$ is a resolvent of R_{i-1} and a clause $C_i \in \Sigma$. Now, we add further restrictions. Let Σ be a set of Horn clauses. Further, let R_i be a resolvent of a selected negative literal in R_{i-1} and the positive literal of a definite clause $C_i \in \Sigma$. The selection rule is called the "computation rule" and the resulting proof strategy is called SLD resolution ("$\underline{\text{S}}$elected $\underline{\text{L}}$inear $\underline{\text{D}}$efinite" resolution). We illustrate this with an example. Let Σ be the set of clauses:

C_0 : $\neg$ Fred is an ape
C_1 : Fred is an ape $\leftarrow$ Fred is human, Fred has hair
C_2 : Fred is human
C_3 : Fred has hair

A little thought should convince you that $\Sigma \models \Box$. We want to see if $\Sigma \vdash_{SLD} \Box$. It is evident that C_0 and C_1 resolve. The resolvent is R_1:

C_0 : $\neg$ Fred is an ape
C_1 : Fred is an ape $\leftarrow$ Fred is human, Fred has hair
R_1 : $\leftarrow$ Fred is human, Fred has hair
C_2 : Fred is human
C_3 : Fred has hair

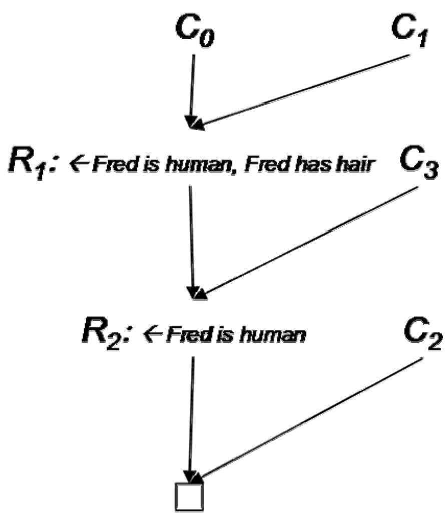


Figure 1.6: Example of SLD-deduction of □ from Σ.

Since we are using SLD, one of the resolvents for the next step has to be R_1. The other resolvent has to be one of the C_i's. Suppose our selection rule selects the "rightmost" literal first for resolution (that is, $\neg$ Fred has hair in R_1) This resolves with C_3, giving $R_2 : \neg$ Fred is human, which in turn resolves with C_2 to give □. The SLD (input) resolution diagram for this is presented in Figure 1.6.

It is more common, especially in the logic-programming literature, to present instead the search process confronting a SLD-resolution theorem prover in the form of a tree-diagram, called an *SLD-tree*. Such a tree effectively contains all possible derivations that can be obtained using a particular literal selection rule. Each node in the tree is a "goal" of the form $\leftarrow L_1, L_2, \ldots, L_k$. That is, it is a clause of the form $(\neg L_1 \vee \neg L_2 \vee \cdots \vee \neg L_k)$. Given a set of clauses Σ, the children of a node in the SLD-tree are the result of resolving with clauses in Σ (nodes representing the empty clause □ have no children). The SLD-tree for the example we just looked at is shown in Figure 1.7

We can now see what refutation-completeness for Horn clauses for SLD-resolution means in terms of SLD-trees. In effect, this means that if a set of clauses is unsatisfiable then there will be a leaf in the SLD-tree with the empty clause □. Further, the computation rule will not alter this (informally, you can see that different computation rules will simply move the location of the □ around). We will have more to say on SLD-resolution with first-order logic in a later section. There, we will see that in addition to the computation rule, we will also need a "search" rule that determines how the SLD-tree is searched. Search

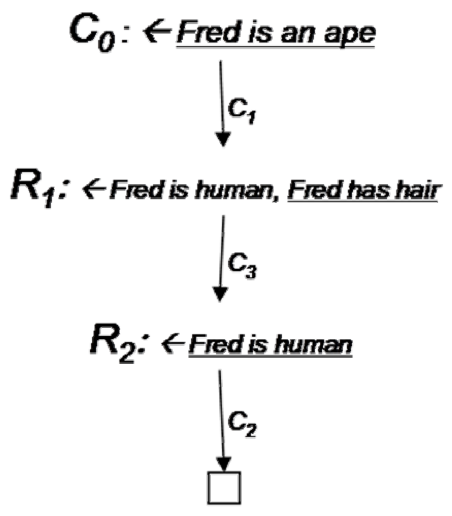


Figure 1.7: Example SLD-tree with C_0 at root.

trees there can have infinite branches, and although completeness is unaffected by the choice of the computation rule (that is, there will be a □ in the tree if the set of first-order clauses is unsatisfiable), we may not be able to reach it with a fixed search rule.

Theorem 11 *If Σ is an unsatisfiable set of horn clauses, then there is an SLD refutation of Σ.*

Proof:
We can assume Σ is finite. Let n be the number of facts (clauses consisting of a single positive literal) in Σ. We prove the lemma by induction on n

1. If $n = 0$, then $\Box \in \Sigma$ for otherwise the empty set would be a Herbrand model of I.

2. Suppose the lemma holds for $0 \leq n \leq m$, and suppose $m+1$ distinct facts appear in Σ. If $\Box \in \Sigma$ the lemma is obvious, so suppose $\Box \notin \Sigma$.

   Let, A be a fact in Σ. We first delete all clauses from Σ; which have A as head (so we also delete the fact A from Σ). Then we replace clauses which have A in their body by clauses constructed by deleting these atoms A from the body (so for example, $B \leftarrow A, B_1, \ldots, B_k$ will be replaced by $B \leftarrow B_1, \ldots, B_k$). Call the set obtained in this way Σ'. If M is a model of Σ', then $M \cup \{A\}$ is a Herbrand model of Σ. Thus since Σ is unsatisfiable,

$$\begin{tikzcd}
R_0 \notin \Sigma \arrow[d] & C_1 \in \Sigma \arrow[dl] & (R_0 \vee \neg A) \in \Sigma \arrow[d] & C_1 \in \Sigma \arrow[dl] \\
R_1 \arrow[d] & C_2 \notin \Sigma \arrow[dl] & R_1 \vee \neg A \arrow[d] & C_2 \vee \neg A \in \Sigma \arrow[dl] \\
R_2 = \Box & & \leftarrow A, A \arrow[d] & A \in \Sigma \arrow[dl] \\
& & \leftarrow A \arrow[d] & A \in \Sigma \arrow[dl] \\
& & \Box &
\end{tikzcd}$$

Figure 1.8: Illustration of the proof for theorem 11. The SLD refutation of Σ' is on the left and that for Σ is on the right.

Σ' must be unsatisfiable. Σ' only contains m facts, so by the induction hypothesis, there is an SLD-refutation of Σ'. If this refutation only uses clauses from Σ' which were also in Σ then this is also an SLD-refutation of Σ, so then we are done. Otherwise, if C is the top clause or an input clause in this refutation and $C \notin \Sigma$, then C was obtained from some $C' \in \Sigma$ by deleting all atoms A from the body of Σ. For all such C, do the following: restore the previously deleted copies of A to the body of C (which turns C into C' again), and add these atoms A to all later resolvents. This way, we can turn the SLD-refutation of Σ' into an SLD-derivation of $\leftarrow A, \ldots, A$ from Σ. See Figure 1.8 for illustration, where we add previously deleted atoms A to the bodies of R_0 and C_2. Since also $A \in \Sigma$, we can construct an SLD-refutation of Σ, using A a number of times as input clause to resolve away all members of the goal $\leftarrow A, \ldots, A$.

$\Box$

1.3.5 Davis-Putnam Procedure

The inference problem addressed so far (particularly through the resolution procedure) is to determine if a proposition α logically follows from a given logical theory Σ. As we saw, this is achieved by reducing the problem to a coNP-complete[15] unsatisfiability problem; based on the contradiction theorem, it amounts to negating the goal formula α, add it to the theory and test the conjunction for unsatisfiability.

However, often, one is faced with the requirement for a model M for a logical theory Σ. This can turn out to be easier problem than the usual problem of

[15]A decision problem C is Co-NP-complete if it is in Co-NP and if every problem in Co-NP is polynomial-time many-one reducible to it. The problem of determining whether a given boolean formula is tautology is a coNP-complete problem as well. A problem C is a member of co-NP if and only if its complement $\overline{C}$ is in complexity class NP. For example, the satisfiability problem is an NP-complete problem. Therefore the unsatisfiability problem is a coNP-complete problem.

inference, since it is enough to find one model for the theory, as against trying all possible truth assignments as in the case of solving an unsatisfiability problem. For example, theory might describe constraints on the different parts of a car. And you are interested in a model that satisifes all the constraints. In terms of search, you search the space of assignment and stop when you find an assignment that satisifies the theory.

While the resolution procedure can be modified so that it gives you a model, the Davis-Putnam-Logemann-Loveland (DPLL) procedure is a more efficient procedure for solving SAT problems. Given a set of clauses Σ defined over a set of variables $\mathcal{V}$, the Davis-Putnam procedure $DPLL(\Sigma)$ returns 'satisfiable' if is satisfiable. Otherwise return 'unsatisfiable'.

The $DPLL(\Sigma)$ procedure consists of the following steps. The first two steps specify termination conditions. The last two rules actually work on the clauses in Σ.

1. If $\Sigma = \emptyset$, return 'satisfiable'. This convention was introduced on pages 22 when we introduced logical entailment, as also in the footnote on page 29.

2. If $\Box \in \Sigma$ return 'unsatisfiable'. This convention was discussed in the footnote on page 29.

3. **Unit-propagation Rule:** If Σ contains any unit-clause $C = \{c\}$ (*c.f.* page 29 for definition), assign a truth-value to the variable in literal c that satisfies c, 'simplify' Σ to Σ' and (recursively) return $DPLL(\Sigma')$. The rationale here is that if Σ has any unit clase $C = \{c\}$, the only way to satisfy C is to make c true. The simplification of Σ to Σ' is achieved by:

   (a) removing all clauses from Σ that contain the literal c (since all such clauses will now be true)

   (b) removing the negation of literal c (*i.e.*, $\neg c$) from every clause in Σ that contains it

4. **Splitting Rule:** Select from $\mathcal{V}$, a variable v which has not been assigned a truth-value. Assign one truth value t to it, simplify Σ to Σ' and (recursively) call $DPLL(\Sigma')$.

   (a) If the call returns 'satisfiable' (*i.e.*, we made a right choice for truth value of v), then return 'satisfiable'.

   (b) Otherwise (that is, if we made a wrong choice for truth value of v), assign the other truth-value to v in Σ, simplify to Σ'' and return $DPLL(\Sigma'')$.

The DPLL procedure can construct a model (if there exists one) by doing a book-keeping over all the assignments. This procedure is complete (that is, it constructs a model if there exists one), correct (the procedure always finds a trutn assignment that is a model) and guaranteed to terminate (since the space of possible assignments is finite and since DPLL explores that space systematically). In the worst case, DPLL requires exponential time, owing to the splitting rule. This is not surprissing, given the NP-completeness of the SAT problem. Heuristics are needed to determine (i) which variable should be instantiated next and (ii) to what value the instantiated varaible should be set. In all SAT competitions[16] so far, Davis Putnam-based procedures have shown the best performance.

As an illustration, we will use the DPLL procedure to determine a model for $\Sigma = \{\{a, b, \neg c\}, \{\neg a, \neg b\}, \{c\}, \{a, \neg b\}\}$. Since Σ is neither an empty set nor contains the empty clause, we move on to step 3 of the procedure. Σ contains a single unit clause $\{c\}$, which we will set to true and simplify the theory to $\Sigma^1 = \{\{a, b\}, \{\neg a, \neg b\}, \{a, \neg b\}\}$. Next, we apply the splitting rule. Let us choose a and set it to 'false'. This yields $\Sigma^2 = \{\{b\}, \{\neg b\}\}$. It can be verified that $\Sigma^2 \equiv \{\Box\}$. We therefore backtrack and set a to 'true'. This yields $\Sigma^3 = \{\{\neg b\}\}$. Thereafter, application of unit propagation yields $\Sigma^4 = \{\}$, which is satisfiable according to step 1 of the procedure. Thus, using the DPLL procedure, we obtain a model for Σ as $M = \{a, c\}$.

As another example, consider $\Sigma = \{\{a, \neg b, \neg c, \neg d\}, \{b, \neg d\}, \{c, \neg d\}, \{d\}\}$. As we will see, application to DPLL to this problem does not involve any backtracking, making the task relatively easier. Σ is neither an empty set nor contains the empty clause. Hence we apply the unit propagation rule 3 on $\{d\}$ to obtain $\Sigma^1 = \{\{a, \neg b, \neg c\}, \{b\}, \{c\}\}$. We can again apply unit propagation to Σ^1 on $\{b\}$ to obtain $\Sigma^2 = \{\{a, \neg c\}, \{c\}\}$. Finally, we can apply unit propagation to Σ^2 on $\{c\}$ and then to $\{a\}$ obtain $\Sigma^4 = \{\}$, which is satisfiable. This yields a model $M = \{d, b, c, a\}$ of Σ.

The DPLL procedure is similar to the traditional constraint propagation procedure. The splitting rule in DPLL is similar to the backtracking step in constraint propagation, while the unit propagation rule is similar to the unavoidable steps of consistency checking and forward propagation in traditional CP.

Davis Putnam on Horn Clauses

The Davis Putnam procedure is polynomial on horn clauses. Why this is so will become apparent through the following sequence of arguments:

1. Since the DPLL steps (1 through 4) could atmost throw out a clause or throw out a literal within a clause, it is obvious that simplifications in DPLL on Horn clauses always generate Horn clauses.

2. A set of Horn clauses without unit clauses can be always satisfied by an assignment of 'false' to all variables (since all clauses will have atleast one negative literal).

3. The first sequence of applications of the unit propagation rule in DPLL should either lead to the empty clause (which is satisfiable) or should lead

[16] `http://www.satcompetition.org/`

to a set of Horn clauses that has no unit clause (which is also satisfiable according to statement (2) above).

4. Thus, for Horn clauses, it suffices to apply steps 1 through 3 of the DPLL procedure. Since no backtracking is involved and since steps 1 through 3 can apply atmost n times (n being the number of variables in the theory), for Horn clauses, DPLL is polynomial in the number of variables.

5. Even if step 4 of DPLL were applied in the case of Horn clauses, the time taken would still be polynomial in the number of variables. This is because, assigning a value 'false' to the variable chosen in step 4 cannot change the satisfiability, whereas, assigning a value 'true' can either lead to an immediate contradiction (after unit propagation) or will not affect satisfiability at all.

Phase Transitions

We saw that in the worst case, DPLL requires exponential time. Couldn't we do better in the average case? For CNF-formulæ in which the probability for a positive appearance, negative appearance and non-appearance in a clause is 1/3, DP needs on average quadratic time [Gol79]. In retrospect, it was discovered that the formulæ in [Gol79] have a very high probability of being satisfiable. Thus, these formulæ are not representative of those encountered in practice. The idea of *phase transition* was conjectured by [CKT91] to identify hard to solve problem instances:

> *All NP-complete problems have at least one order parameter and the hard to solve problems are around a critical value of this order parameter. This critical value (a phase transition) separates one region from another, such as over-constrained and under-constrained regions of the problem space.*

This conjecture was initially confirmed for the graph coloring and Hamilton path problems and later for other NP-complete problems, including SAT. In the case of SAT problem, an example of the order parameter is ratio of the number of variables to the number of clauses.

1. For higher settings of the parameter, the problem is over-constrained (the formulæ are unsatisfiable). If the probability of a solution is close to 0, this fact can usually be determined early in the search.

2. For lower settings of the parameter, the problem is under-constrained (the formulæ are easily satisfiable). When the probability of a solution is close to 1, there are many solutions, and the first search path of a backtracking search is usually successful.

When this parameter is varied, the problem moves from the over-constrained to the under-constrained region (or vice versa). At phase the transition points, half of the problems are satisfiable and half are not. It is typically in this region that algorithms have difficulty in solving the problem[17]. Cook and Mitchell [CM97] empirically found that for a 3-SAT problem, the phase transition occurs at a clause:variables ratio of around 4.3 (in this experiment, clauses were generated by choosing variables for a clause and complementing each variable with probability 0.5). As an illustration, in the 2003 version of the SAT competition, the largest instances solved using greedy SAT solvers consisted of $100,000$ variables and $1,000,000$ clauses (clause:variable ratio of 10), whereas the smallest unsolved instances comprised 200 variables and $1,000$ clauses (clause:variable ratio of 5). It was also reported in [CM97] that the runtime for the DPLL procedure peaks at the phase transition. In the phase transition region, the DPLL algorithm often near successes. Many benchmark problems are located in the phase transition region, though they have a special structure in addition.

1.3.6 Local Search Methods

Local search methods are standard search procedures for optimization problems. A local search method explores the neighborhood of the current solution and tries to enhance the solution till it cannot do any better. The hope is to produce better configurations through local modifications. The value of a configuration in a logical problem could be measured using the number of satisfied constraints/clauses. However, for logical problems, local maxima are inappropriate; it is required to satisfy all clauses in the theory and not just some. But through random restarts or by noise injection, local maxima can be escaped. In practice, local search performs quite well for finding satisfying assignments of CNF formulæ, especially for under-constrained or over-constrained SAT problems.

GSAT and WalkSat [SKC93] are two local search algorithms to solve boolean satisfiability problems in CNF. They start by assigning a random value to each variable. If the assignment satisfies all clauses, the algorithm terminates, returning the assignment. Otherwise, an unsatified clause is selected and the value of exactly one variable changed. Due to the conjunctive normal form, flipping one variable will result in that clause becoming satisfied. The above is then repeated until all the clauses are satisfied. WalkSAT and GSAT differ in the methods used to select the variable to flip. While GSAT makes the change which minimizes the number of unsatisfied clauses in the new assignment, WalkSAT selects the variable that, when flipped, results in no previously satified clauses becoming unsatified (some sort of downward compatibility requirement). MaxWalkSat is a variant of WalkSat designed to solve the weighted satisfiability problem, in which each clause is associated with a weight. The goal in MaxWalkSat is to find an assignment (which may or may not satisfy the entire formula) that maximizes the total weight of the clauses satisfied by that assignment. These algorithms

[17]Note that hard instances can also exist in regions of the more easily satisfiable or unsatisfiable instances.

perform very well on the randomly generated formulæ in the phase transition region. Monitoring the search procedure of these greedy solvers reveals that in the begginning, each procedure is very good at reducing the number of unsatisfied clauses. However, it takes a long time to satisfy the few remaining clauses (called plateaus). The GSAT algorithm is outlined in Figure 1.9.

```
INPUT: A set of clauses $\Sigma$, MAX-FLIPS, and MAX-TRIES.
OUTPUT: A satisfying truth assignment of $\Sigma$, if found.
for $i = 1$ to MAX-TRIES do
  $T$ = a randomly-generated truth assignment.
  for j:=1 to MAX-FLIPS do
    if $T$ satisfies $\Sigma$ then
      return $T$
    end if
    $v$ = a propositional variable such that a change in its truth assignment
    gives the largest increase in the number of clauses of $\Sigma$ that are satisfied
    by $T$.
    $T = T$ with the truth assignment of $v$ reversed.
  end for
end for
return "Unsatisfiable".
```

Figure 1.9: Procedure GSAT.

1.3.7 Default inference under closed world assumption

Given any set of formulae Σ, the closed-world assumption is the assumption that Σ determines all the knowledge there is to be had about the formulae in the language. Thus, if we consider any proposition[18] A then A is taken to be true exactly when Σ (logically) implies A, but is otherwise taken to be false.

The closed-world assumption underlies the mode of reasoning known as *default inference*. There are many situations, both in ordinary daily life and in specific technical computing matters (such as explaining the theory of finite failure and the relationship it bears to reasoning with negative information), when default inference is a necessary supplement to deductive inference. Consider this simple example of a single clause axiom $\Sigma = \{A \leftarrow B\}$ for which $B(\Sigma)$ is just $\{A, B\}$. Under the closed-world assumption, we may infer $\neg A$ for any ground atom $A \in B(\Sigma)$ that us not implied by Σ. This is a constrained[19] application of the rule of default inference:

[18]In the first order logic programming context, the propositions in which we are primarily interested are the atoms of the Herbrand base.

[19]Constrained because (i) Σ is assumed to be inconsistent, else Σ would necessarily imply both A and $\neg A$ (ii) only the case where A is atomic is considered, otherwise if Σ implied, say, neither A nor $\neg A$, then the default rule would infer both $\neg A$ and $\neg A$, which would again be inconsistent.

Infer $\neg A$ in default of Σ implying A

Motivated by the desire to draw sound conclusions about negative information, we will consider two constructions that provide consequence-oriented meaning for default inference under the closed-world assumption.

1. CWA(Σ) : The combination of Σ with the default conclusions inferred from it is denoted by CWA(Σ) and is defined by

$$\text{CWA}(\Sigma) = \Sigma \cup \{\neg A \mid A \in B(\Sigma) \textit{ and not } \Sigma \models A\}$$

   For the above example, CWA(Σ) $= \{A \leftarrow B, \neg A, \neg B\}$.

   Soundness for the default inference of negative conclusions under the closed-world assumption can be referred to the the logical construction CWA(Σ) as follows:

$$for\ all\ A \in B(\Sigma),\ \text{CWA}(\Sigma) \models \neg A\ if\ \Sigma \vdash_{CWA} \neg A$$

   There some practical problems with CWA:

   (a) One problem is that, we have no immediate way of writing down CWA(Σ), for we cannot directly discern which particular negative facts $\neg A$ to include in it; we would have to infer them all first.

   (b) A more serious defect is that, whilst CWA(Σ) is always consistent when Σ is definite[20], it is likely to be inconsistent otherwise. For example, if Σ comprises just the clause $\{A \vee B\}$, then CWA(Σ) $=$ $\{A \vee B, \neg A, \neg B\}$ which is inconsistent.

2. CWA(Σ) : For default inference applied to indefinite programs, we refer the soundness criterion to *completion* COMP(Σ). It is also known as the completed database of Σ. Unlike CWA(Σ), it can be written down more-or-less directly and is consistent for all well-structured programs. In the case of propositional logic, the construction is particularly simple:

   (a) Initialize COMP(Σ) $= \emptyset$.

   (b) Assume that Σ is any set of clauses of the form $(A \leftarrow\ body)$.

   (c) For each A mentioned in Σ but not defined in Σ, construct $\neg A$ and add it to COMP(Σ).

   (d) For each A having a definition in Σ of the form

$$\begin{array}{l} A \leftarrow\ body - 1 \\ \ldots \\ \ldots \\ A \leftarrow\ body - n \end{array}$$

[20]Exercise: Prove.

construct the clause A iff $(body - 1 \vee \ldots \vee body - n)$ and add it to COMP(Σ),

$Comp(\Sigma)$ can also be viewed as the result of simplifying $\Sigma \cup$ only-if(Σ), where only-if(Σ) comprises every completed definition of the form $\neg A$ for $A \notin \Sigma$ as well as $(q \rightarrow \alpha)$ for every completed definition $(q \text{ iff } \alpha) \in$ COMP(Σ).

Following are some characteristics of completions

(a) In general it is also a more economical construction in the sense that it implies, but does not necessarily declare, various negative propositions which CWA(Σ) would have to declare explicitly. As an example, if $\Sigma = \{A \leftarrow B\}$, COMP($\Sigma$) $= \{\neg B, A \text{ iff } B\}$. While Σ neither implies A nor implies B, $Comp(P)$ implies both $\neg A$ and $\neg B$, which is exactly the same outcome obtained using CWA(Σ) instead. Observe, however, that CWA(Σ) declares $\neg A$ **explicitly** whereas $Comp(\Sigma)$ does not.

(b) Generally, completion is more conservative than the closed-world assumption in the negative facts that it implies. For example, with $\Sigma_1 = \{A \leftarrow A\}$, CWA($\Sigma_1$) $\models \neg A$, whereas COMP(Σ_1) $\not\models \neg A$. Similarly, with $\Sigma_2 = \{A \leftarrow \neg B\}$, CWA($\Sigma_2$) $\models \neg A \wedge \neg B$ (is inconsistent), whereas COMP(Σ_2) $\not\models \neg A$ though COMP(Σ_2) $\models \neg B$.

(c) On the other hand, COMP can be sometimes less conservative; for $\Sigma_3 = \{A \leftarrow \neg A\}$, COMP($\Sigma_3$) $\models$ *everything*, whereas CWA(Σ_3) $\models$ A and CWA(Σ_3) $\not\models \neg A$.

(d) Completed definitions capture the programmer's intentions more fully than do uncompleted definitions in the original program. For computational purposes however, the program alone happens to be sufficient for deducing all intended answers.

(e) In cases where Σ is indefinite, the elementary consequences of COMP(Σ) may arise from the joint contributions of Σ and only-if(Σ) and not from either of them alone.

(f) There is a syntactic bias built into the process of program completion. For example, if $\Sigma_1 = \{A \leftarrow \neg B\}$ and $\Sigma_2 = B \leftarrow \neg A$ their respective completions are COMP(Σ_1) $= \{A \text{ iff } \neg B, \neg B\}$ and COMP(Σ_2) $= \{B \text{ iff } \neg A, \neg A\}$. These completions are not logically equivalent despite the fact that the original programs are. The difference between their completions deliberately reflects the difference between the procedural intentions suggested by the programs' syntaxes: the first program anticipates queries of the form $?A$ whilst the second anticipates queries of the form $?B$.

Finite Failure Extension

We look at one further extension of SLD-resolution that is of special interest to us, namely, the idea of negation as "failure to prove". **We know that SLD**

alone is able to deal soundly and completely with all positive atomic queries A posed to this database. But with the finite failure facility we can now also ask directly, in our extended language, whether some atom is *not* in the database.

This extension enables one to express, via a call '*fail A*', the condition that a call A shall finitely fail. In Prolog the fail operator is denoted by '*not*' and is referred to as 'negation by failure'. The operational meaning of '*fail*' is summed up by the finite failure rule:

A call '*fail A*' succeeds *iff* its subcall A finitely fails.

This rule can be incorporated directly into the execution strategy for virtually no cost in terms of implementation overheads - to evaluate a call '*fail A*', the interpreter merely evaluates A in the standard way and then responds to the outcome as just prescribed. The phrase 'A finitely fails' means that the execution tree generated by the evaluation of A must be a finitely failed tree - that is, it must have finite depth, finite breadth and all the computations contained within it must terminate with failure. By contrast, should one or more of those computations terminate with success then the call '*fail A*' is itself deemed to have finitely failed. Finally, if the evaluation of of A neither succeeds nor finitely fails, then the same holds for the evaluation of '*fail A*'

The rule of "negation as finite failure" states that if all branches of an SLD-tree finitely fail for $\Sigma \cup \{\leftarrow A\}$ for a set of definite clauses Σ and a ground literal A, we can derive the ground literal $\neg A$ as a result. This combination of SLD-resolution and negation-as-failure results in the proof strategy we called SLDNF-resolution. Once again, let us look at an example:

C_0 : $\neg$ Fred is an ape

C_1 : Fred is an ape $\leftarrow$ *not* Fred is human, Fred is a primate

C_3 : Fred is a primate

Here, *not* stands for "not provable" (which is not the same as $\neg$). C_0 and C_1 resolve to give R_1 : $\leftarrow$ *not* Fred is human, Fred is a primate. With a rightmost literal computation rule as before, the next resolvent is R_2 : $\leftarrow$ *not* Fred is human. It is evident that Fred is human is not provable and $\Box$ results.

SLDNF is sound and complete for propositional definite clauses. However, there are some important issues in extending SLD with finite failure to first order logic such as (i) incompleteness when applied to activated non-ground fail calls and (ii) unsoundness in certain cases. To solve the latter, more serious problem, a selection policy called *safe computation rule* is used in practice; this policy sacrifices completeness for the sake of soundness. These and other concepts such as *floundering* will be discussed in a later section.

The SLD finite failure set

Relative to a given Σ and a given inference system R, the SLD finite failure $FF(\Sigma, R)$ set comprises exactly propositional atoms for which the query $?q$

finitely fails. We shall restrict our attention throughout to the case where Σ is definite and R is SLD. Once we know what the finite failure set is relative to SLD inference, we shall also be able to say something about the execution by SLDNF of queries containing 'fail calls' - subject, of course, to the assumption that the chosen computation rule is safe. The more general situation where 'fail' calls may occur also in clause bodies, is significantly more complicated, and will not be addressed here. For queries of the form $?q$ using a definite Σ under some SLD computation rule R partitions $B(\Sigma)$ into three species of atoms

1. $SS(\Sigma, R)$, or those for which $?q$ succeeds. Since, if $?q$ succeeded in one SLD tree, it succeeded in all SLD trees, the capacity for $?q$ to succeed is independent of the computation rule. So this set can be simply denoted by $SS(\Sigma)$.
2. $FF(\Sigma, R)$, or those for which $?q$ finitely fails
3. $IF(\Sigma, R)$, or those for which $?q$ infinitely fails.

However, $?q$ may finitely fail under one computation rule yet infinitely fail under another. A simple example is where Σ comprises just the clause $q \leftarrow B \wedge q$. Under the standard leftmost call rule, the query $?q$ finitely fails, whereas it infinitely fails under a rule which always selects the rightmost call. Thus the boundary between $FF(\Sigma, R)$ and $IF(\Sigma, R)$ depends upon R. There is a simple theoretical way of eliminating this dependence upon R. We invoke the idea of a particular sort of computation rule which ensures that any call introduced into a computation is selected after some arbitrary but finite number of execution steps. Such a rule is called a *fair computation rule.* Considering the example again, we might allow a fair computation rule to select 'q' calls any finite number of times, but sooner or later the fairness requirement would demand that an 'A' call be selected, thus immediately forcing finite failure. With this new concept, we can now state the following facts: $?q$ finitely fails under some computation rule *iff* it finitely fails under all fair computation rules[21] We can denote by $FF(\Sigma)$, the set of all atoms q for which $?q$ has some finitely failed fair SLD-tree, and by $IF(\Sigma)$, the set of all atoms q for which $?q$ has some infinitely failed fair SLD-tree. $FF(\Sigma)$ is called the (fair-) SLD finite failure set of Σ. Referring again to the example above, we shall have $FF(\Sigma) = \{q, A\}$ and $IF(\Sigma) = \emptyset$.

Suppose queries were also allowed to contain '*fail*' calls besides atomic ones, but with Σ still restricted to be definite. The following statements then hold true:

$$\begin{array}{llll} \text{for all } q \in B(\Sigma), & q \in FF(\Sigma) & \text{iff} & ?fail\ q \text{ succeeds under SLDNF} \\ & q \in SS(\Sigma) & \text{iff} & ?q \text{ succeeds under SLDNF} \end{array}$$

We can interpret finite failure (i) in terms of the position of $FF(\Sigma)$ within the lattice of interpretations and (ii) in terms of the classical negation ($\neg$), which

[21]Although fair computation rules are not normally implemented, they certainly tidy up our mathematical account of finite failure.; equivalently, at least one of its SLD-trees is finitely failed *iff* all of its fair SLD-trees are finitely failed.

provides a semantics for '*fail*', and some soundness and completeness results for SLDNF, based upon the logical consequences of COMP(Σ).

Completion Semantics for SLDNF

For pure Horn-clause programs and queries, we had a particularly simple connection between logical meaning and operational meaning:

for all $q \in B(\Sigma)$, $\Sigma \models q$ *iff* $q \in SS(\Sigma)$

The construction of a consequence-oriented semantics for the finite failure extension is more problematic. A program containing fail is not a construct of classical logic and so is not amenable to the notion of classical logical consequence. Nevertheless, a variety of analogous connections have been suggested. The best-known of these is based upon the so-called *completion semantics*, which relies upon two ideas:

1. Interpreting 'fail' as the classical negation connective ¬ (this is why fail is commonly referred to as 'negation by failure').
2. Relating the success or finite failure of calls to logical consequences of COMP(Σ) rather than of Σ alone.

On this basis we can then characterize, in logical terms, the *soundness* of execution of atomic queries under the (safe-)SLDNF implementation of fail:

for all $q \in B(\Sigma)$,	$\text{COMP}(\Sigma) \models q$	if $?q$	succeeds under SLDNF
	$\text{COMP}(\Sigma) \models \neg q$	if $?\ fail\ q$	succeeds under SLDNF

These results hold even if '*fai;*' calls occur in Σ. The 'only-if' part in the two statements above, which characterizes the *completeness* for SLDNF holds only when Σ is definite (so Σ will obviously not contain 'fail' calls).

There are two caveats in the above statement(s): (i) COMP(Σ) may itself be inconsistent and (ii) the SLDNF uses some safe computation rule. Also, the simplification of analysis of finite failure using *fair* computation rules is not possible when the query contains '*fail*' calls.

1.3.8 Lattice of Models

For any Σ that is a set of definite clausal formula, it can be shown that the set $\mathcal{M}(\Sigma)$ of models is a complete lattice ordered by set-inclusion (that is, for models $M1, M2 \in \mathcal{M}(\Sigma)$, $M1 \preceq M2$ if and only if $M1 \subseteq M2$), and with the binary operations of $\cap$ and $\cup$ as the glb and lub respectively. Recall that a complete lattice has a unique least upper bound and a unique greatest lower bound. Since we are really talking about sets thar are ordered by set-inclusion, this means that there is a a unique *smallest* one (the size being measured by the number of elements). This is called the *minimal model* of the formula, and it can be shown that this must be the intersection of all models for the formula. We will illustrate with an example. Consider a Σ consisting of the following clauses:

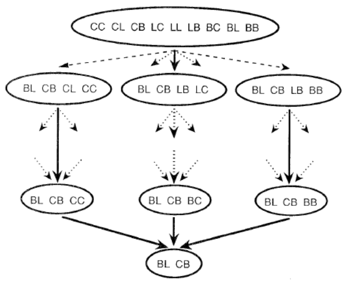


Figure 1.10: A complete lattice of models.

C_0 : CC ← CL
C_1 : CB ← ¬ BL
C_2 : CL ← LL
C_3 : BL

A fail-safe way of getting a model is to choose the entire set of propositions $B(\Sigma)$ ={CC, CL, CB, LC, LL, LB, BC, BL, BB}, which is called the *base* of Σ. for then every atom in Σ is assigned true and this in turn makes all of its clauses true. This model is the (unique) maximal model for Σ. However, such a choice is clearly excessive-many of the atoms in B(P) do not even occur in G(P) and so their truth values are irrelevant. Stripping out those irrelevant atoms leaves a somewhat smaller model {CC, CL, CB, BL, LL}. Which of these atoms must appear in any model? Clearly BL must, in order to make the program's fourth clause true. Then, since BL must, so also must CB in order to make the second clause true. The first and third clauses employ only the remaining three atoms CC, CL and LL, and both those clauses can he made true by making those atoms false. In conclusion, only BL and CB *must* be true – thus the (unique) minimal model for Σ is {BL,CB}.

Some indication of the complete lattice of models for the program is shown in Figure 1.10 where, edges stand for the *covers* relationship. Moreover, only a few of the models are shown-there are 64 models in the entire lattice. Note also that, in general, most subsets of $B(\Sigma)$ will be counter-models–for our example there are 448 of these, of which the smallest is $\emptyset$. It can be proved that if Σ is a set of definite clauses, it must always be satisfiable, and therefore, $\emptyset \notin \mathcal{M}(\Sigma)$

The reason why the discussion above has focused solely upon definite clauses is that, in general, a set of indefinite clauses does not yield a complete lattice and may therefore have multiple minimal models. The lack of a unique minimal model makes it harder to assign an unambiguous meaning to such a set.

There is an important result relating a set of definite clausal formulae Σ, its minimal model $\mathcal{MM}(\Sigma)$ and the atoms that are logical consequences of Σ:

Theorem 12 If α is an atom then $\Sigma \models \alpha$ if and only if $\alpha \in MM(\Sigma)$.

The proof of theorem 12 makes use of the so-called model intersection property:

Theorem 13 *If $M_1, \ldots, M_n$ are any models for a definite clause set Σ then their intersection is a model of Σ.*

Proof: It is easy to show this by induction. For any $k < n$, let I_k denote $M_1 \cap \ldots \cap M_k$. Now consider any clause $C \in \Sigma$:
C : $A \leftarrow B_1 \wedge \ldots \wedge B_m$
We shall prove that, for all $k \leq n$, I_k satisfies C.

1. *Base case ($k = 1$):* $I_1 = M_1$ and therefore satisfies C.
2. *Induction step ($1 \leq k \leq n$)*: Assume that M_k satisfies C;

if I_k satisfies C then	either	$B_i \notin I_k$ for some i
	or	$B_i \in I_k$ for all i, and $A \in I_k$
if M_{k+1} satisfies C then	either	$B_i \notin M_{k+1}$ for some i
	or	$B_i \in M_{k+1}$ for all i, and $A \in M_{k+1}$
it then follows that	either	$B_i \notin I_k \cap M_{k+1}$ for some i
	OR	$B_i \in I_k \cap M_{k+1}$ for all i, and $A \in I_k \cap M_{k+1}$
And hence		I_{k+1} satisfies C

Thus for all $k \leq n$, I_k satisfies C and-by a similar argument – every other clause in Σ.
□
We next prove theorem 12 making use of theorem 13.
Proof of theore 12: EXERCISE
We will first prove that the intersection I^* of all the models of Σ is the minimal model $MM(\Sigma)$, using contradiction as follows:

Suppose	I^* is not the minimal model
	then there must exist some model M_j such that $M_j \subset I^*$
	then there must exist some atom q such that $q \notin M_j$ and $q \in I^*$
	but $q \in I^*$ implies that $q \in M_i$, for all i, contradicting $q \in M_j$
	therefore the initial assumption is false.

Using this result it is now easy to prove the relationship between $MM(\Sigma)$ and any atomic consequence q of Σ:

If $\Sigma \models q$	then	q is true in every model of Σ
	then	$q \in I^*$
	then	$q \in MM(\Sigma)$
if $q \in MM(\Sigma)$	then	q is true in every model of Σ
	then	$\Sigma \models q$

□

Thus, the minimal model of a definite clausal formula is identical to the set of all ground atoms logically implied by that formula, which was defined as the success set $SS(\Sigma)$. Thus, the minimal model provides, in effect, the meaning (or semantics) of the formula. We shall see later that this is just one of several ways of giving significance to a program's minimal model.

We can envisage a procedure for enumerating the models of a formula. Consider the powerset of the base $B(\Sigma)$. Now, we know that this powerset ordered by $\subseteq$ necessarily forms a complete lattice, with binary operations $\cap$ and $\cup$. Some subset of this powerset is the set of all models, which we know is also a lattice ordered by $\subseteq$ with the same binary operations. So, the model lattice is a sublattice of the lattice obtained from the powerset of the base. Suppose now we start at some point s in this sublattice, and we move to a new point that consists only of those atoms of the formula made true by the model s. Let us call these atoms s_1. Then, a little thought should convince you that s_1 is also a member of the sublattice of models. Repeating the process with s_2 we can move to models s_2, s_3 and so on. Will this procedure converge eventually on the minimal model? Not necessarily, since we could end up moving back-and-forth between points of the sub-lattice. (When will this happen, and how can we ensure that we do converge on the minimal model?).

A slightly more general process can be formalised as the application of a function T_Σ that, for a set of clauses Σ, generates an interpretation (not necessarily a model) from another. That is:

$$I_{k+1} = T_\Sigma(I_k)$$

where

$$T_\Sigma(I) = \{a : a \leftarrow \mathit{body} \in \Sigma \text{ and } \mathit{body} \in I\}$$

It can be shown that T_Σ is both monotonic and continuous on the complete lattice obtained by ordering the powerset of the base by $\subseteq$. So, we know from the Knaster-Tarski Theorem mentioned on page 11, that there must be a least fixpoint for T_Σ in this lattice. We can prove that the procedure of obtaining I_{k+1} from application of T_Σ to I_k will yield that fixpoint, and further, that this fixpoint will be the minimal model.

Theorem 14 *$MM(\Sigma)$ is the least fixpoint of T_Σ.*

Proof: The definition of T_Σ requires that

for all $I \subseteq B(\Sigma)$ and for all q, $q \in T_\Sigma(I)$ iff $(\exists \mathit{body})[(q \leftarrow \mathit{body}) \in \Sigma$ and $\mathit{body} \in I]$

whereas, the definition of a model requires that

for all $I \subseteq B(\Sigma)$, I is a model for Σ iff for all q, $q \in I$ if $(\exists \mathit{body})[(q \leftarrow \mathit{body}) \in \Sigma$ and $\mathit{body} \in I]$.

These two sentences jointly imply

$$\begin{array}{lll} \text{for all } I \subseteq B(\Sigma) & & \\ I \text{ is a model for } \Sigma & \text{iff} & \text{for all } q,\ q \in I \text{ if } q \in T_\Sigma(I) \\ & \text{iff} & T_\Sigma(I) \subseteq I \\ & \text{iff} & I \text{ is a pre-fixpoint of } T_\Sigma \end{array}$$

Where, for a function f on $< S, \preceq >$, an element $u \in S$ is a pre-fixpoint of f if and only if $f(u) \preceq u$. Then, we can recall from the proof of the Knaster-Tarski theorem on page 11 that the least fix point is also the least pre-fixpoint. Hence, since the models are exactly the pre-fixpoints, the least model $MM(\Sigma)$, which is the glb of all the pre-fix points, must be the least pre-fixpoint. Finally, by the Knaster-Tarski theorem, $MM(\Sigma)$ must also be the least fixpoint. □

So we now have several equivalent characterizations of the minimal model, including the success set $SS(\Sigma)$ and the new one posed in terms of the least fixpoint of T_Σ, denoted $LFP(T_\Sigma)$

Mathematical Characterization of Finite Failure

It so happens that there is a somewhat related (to the T_Σ function) method of constructing the finite failure set $FF(\Sigma)$. The set of atoms which occur as headings in Σ is simply $T_\Sigma(B(\Sigma))$, as is plain from the definition of T_Σ. The simplest way for $?q$ to fail finitely is for there to be no clause in Σ who heading unifies with q. In this case the failure is said to occur within depth $k = 1$ and the set of all such atoms $q \in B(\Sigma)$ is denoted by $FF(\Sigma, 1)$ and can be characterized very easily using the T_Σ function as

$$FF(\Sigma, 1) = B(\Sigma) - T_\Sigma(B(\Sigma))$$

Generalizing this principle, $FF(\Sigma)$ just contains each atom q for which $?q$ finitely fails within some depth $k \in \mathcal{N}$.

$$FF(\Sigma) = \cup_{k \in \mathcal{N}} FF(\Sigma, k) = B(\Sigma) - \cap_{k \in \mathcal{N}} T^k_\Sigma(B(\Sigma))$$

When we start at the top element $B(\Sigma)$ of our lattice of interpretations, repeated application of the T_Σ function generates a monotonically decreasing sequence $B(\Sigma) \supseteq T_\Sigma(B(\Sigma)) \supseteq T^2_\Sigma(B(\Sigma)) \ldots$, whose limit is the greatest lower bound (glb) $T_\Sigma \downarrow$ of $\left\{T^k_\Sigma(B(\Sigma)) \mid k \in \mathcal{N}\right\}$. Thus, the mathematical characterization of $FF(\Sigma)$ (independent of the execution mechanism) is

$$FF(\Sigma) = B(\Sigma) - T_\Sigma \downarrow$$

Based on this equivalence, what is the value of $FF(\Sigma)$ for the last example that we discussed?

It can be proved that $T_\Sigma \uparrow = LFP(T_\Sigma) = MM(\Sigma) \subseteq T_\Sigma \downarrow$. There is an assymetry in the relationship between the limits and the extremal fixpoints of T_Σ. Whereas, $T_\Sigma \uparrow$ is always equal to the least fix point $LFP(T_\Sigma)$, $T_\Sigma \downarrow$ does not always equal the greatest fix point $GFP(T_\Sigma)$, though it does hold for most 'sensible' Σ (at the least for non-recursive clauses and in the case of first order logic, for function-free programs). The region between $T_\Sigma \uparrow$ and $T_\Sigma \downarrow$ corresponds to $IF(\Sigma)$, which comprises exactly those atoms which fail infinitely. In practice, most sensible programs have $IF(\Sigma) = \emptyset$, leading to $T_\Sigma \uparrow = T_\Sigma \downarrow$ and therefore $B(\Sigma) = SS(\Sigma) \cup FF(\Sigma)$.

1.4 First-Order Logic

Suppose you wanted to express logically the statement: 'All humans are apes.' One of two ways can be used to formalise this in propositional logic. We can use a single proposition that stands for the entire statement, or with a well-formed formula consisting of a lot of conjunctions: Human1 is an ape $\wedge$ Human2 is an ape .... Using a single proposition does not give any indication of the structure inherent in the statement (that, for example, it is a statement about two sets of objects—humans and apes—one of which is entirely contained in the other). The conjunctive expression is clearly tedious in a world with a lot of humans. Things can get worse. Consider the following argument:

Some animals are humans.
All humans are apes.
Therefore some animals are apes.

That the argument is valid is evident: yet it is beyond the power of propositional logic to establish it. If, for example, we elected to represent each of the statements with single propositions then all we would end up with is:

Statement	Formally
Some animals are humans.	P
All humans are apes.	Q
Therefore some animals are apes.	$\therefore R$

But the formal argument is clearly invalid, as it is easy to think up arguments where P, Q are *true* and R is *false*. What is needed is in fact something along the following lines:

Statement	Formally
Some animals are humans.	Some P are Q
All humans are apes.	All Q are R
Therefore some animals are apes.	$\therefore$ some P are R

Here, P, Q, and R do not stand for propositions, but for terms like *animals*, *humans* and *apes*. The use of terms like these related to each other by the expressions 'some' and 'all' will allow us to form sentences like the following:

All P are Q
No P are Q
Some P are Q
Some P are not Q

The expressions 'some' and 'all' are called *quantifiers*, which when combined with the logical connectives introduced in connection with proposition logic ($\neg, \wedge, \vee, \leftarrow$), results in the powerful framework of first-order or *predicate logic*.

1.4.1 Syntax

The language of predicate logic introduces many new constructs that are not found in the simpler, propositional case. We will first introduce these informally.

Constants. It is conventional in predicate logic to use lowercase letters to denote proper names of objects. For example, in the sentence 'Fred is human', Fred could be represented as $fred$.

Variables. Consider the statements:

All humans are apes
Some apes are not human

Using the letter x as a variable that can stand for individual objects, these can be expressed as:

For all x, if x is human then x is an ape
For some x, x is an ape and x is not human

Quantifiers. The language of predicate logic introduces the symbol $\forall$, called the *universal quantifier*, to denote 'for all.' The symbol $\exists$, called the *existential quantifier*, is used to denote 'for some' or, more precisely, 'for at least one.' The sentences above can therefore be written as:

$\forall x$ (if x is human then x is an ape)
$\exists x$ (x is an ape and x is not human)

Predicates. In their simplest case, these are are symbols used to attribute properties to particular objects. It is conventional in logic (but ungrammatical in English) to write the subject after the predicate. Thus the sentence 'Fred is human' would be formalised as $Human(fred)$ [22]. More generally, predicate symbols can be used to represent relations between two or more objects. Thus,

[22]Logicians are a parsimonius lot: they would represent 'Fred is human' as Hf. The representation here is non-standard, but preferred for clarity.

'Fred likes bananas' can be represented as: $Likes(fred, bananas)$. The general form is therefore a predicate symbol, followed by one or more *arguments* separated by commas and enclosed by brackets. The number of arguments is sometimes called the *arity* of the predicate symbol, and the predicate symbol is often written along with its arity (for example, $Likes/2$). Formalising sentences like those above would result in quantified variables being arguments:

$$\forall x \; (\textsf{if } Human(x) \textsf{ then } Ape(x))$$
$$\exists x \; (Ape(x) \textsf{ and not } Human(x))$$

Or, using the logical connectives that we have already come across:

$$\forall x(Ape(x) \leftarrow Human(x))$$
$$\exists x(Ape(x) \wedge \neg Human(x))$$

Functions. Consider the statement: 'The father of Fred is human.' Although we have not named Fred's father, it is evident that a a unique individual is being referred to, and it possible to denote him by using a *function* symbol. One way to formalise the statement is: $Human(father(fred))$. Here, it is understood that $father(fred)$ denotes Fred's father. A function symbol is one which, when attached to one or more terms denoting objects produces an expression that denotes a single object. It is important that that the result is unique: a function symbol could not be used to represent, for example, 'parent of Fred.' As with predicates, the number of arguments of the function is sometimes called its arity.

The following points would not be evident from this informal presentation:

1. Variables need not designate different objects. Thus, in $\forall x \forall y Likes(x, y)$, x and y could refer to the same object;

2. The choice of variable names is unimportant. Thus, $\forall x \forall y Likes(x, y)$ has the same meaning as $\forall y \forall z Likes(y, z)$;

3. The same variable name, if quantified differently, need not designate the same object. Thus, in $\forall x \forall y Likes(x, y) \wedge \forall x \forall y Hates(y, x)$ the x, y in $Likes(\cdots)$ need not be same as the x, y in $Hates(\cdots)$;

4. The order of quantifiers can matter when $\forall$ and $\exists$ are mixed. Thus, $\exists x \forall y Likes(x, y)$ has a different meaning to $\forall y \exists x Likes(x, y)$. However changing the order has no effect if the quantifiers are all of the same type. Thus, $\forall x \forall y Likes(x, y)$ has the same meaning as $\forall y \forall x Likes(x, y)$;

5. "Free" variables in a formula are those that are not quantified. For example, in the formula $\forall x Likes(x, y)$, y is a free variable. In contrast, quantified variables are called "bound" variables. It may not be immediately apparent that a variable can have both free and bound occurrences in a formula. For example in $\exists x(Likes(x, y) \wedge \exists y DisLikes(y, x))$, the variable y is free in the $Likes$ and bound in $Dislikes$. x on the other hand is

bound in both (by the outermost quantifier). It is normal to call a formula with no free variables a *sentence*, and it only really makes sense to ask about the truth of sentences;

6. Negation should be treated with caution. Thus, in 'Some apes are not humans', 'not' plays the role of *complementation*, by stating that the set of apes and the set of non-humans have at least one member in common. This can be formalised as $\exists x(Ape(x) \wedge \neg Human(x))$. One the other hand, 'not' plays the role of true negation in 'It is not true that some apes are humans' formalised as $\neg\exists x(Ape(x) \wedge Human(x))$;

7. It can be tricky to match English sentences to ones that use $\forall$ and $\exists$. Thus, in 'If something has a tail then it is not an ape', the use of 'something' suggests that formalisation would involve $\exists$. The statement is, in fact, a general one about apes not having tails, and involves universal quantification: $\forall x(\neg Ape(x) \leftarrow Tail(x))$;

8. By denoting 'at least one', the existential quantifier $\exists$ includes 'exactly one' and 'all'. This does not coincide exactly with the usual English notion of 'some', which denotes more than one, but less than all.

We can now examine the formal rules for constructing well-formed formulæ in predicate logic. For the language of predicate logic, we will restrict the vocabulary to the following:

Constant symbols:	A string of one or more lowercase letters (except those denoting variables)
Variable symbols:	A lowercase letter (except those denoting constants)
Predicate symbols:	Uppercase letter, followed by zero or more letters
Function symbols:	Lowercase letter, followed by zero or more letters (except those denoting constants or variables)
Quantifier symbols:	$\forall$, $\exists$
Logical connectives:	$\neg, \wedge, \vee, \leftarrow$
Brackets:	(,)

In addition, we will sometimes employ the device of using subscripts to denote unique symbols (for example, $x_1, x_2, \ldots$ for a string of variables).

With this vocabulary, a *term* is simply a constant, variable or a functional expression (that is, a function applied to a tuple of terms). The following are all examples of terms: x, $fred$, $father(fred)$, $father(father(fred))$, $father(x)$.

These, however, are not terms: $\mathit{Likes}(\mathit{fred}, \mathit{bananas})$, $\mathit{Likes}(\mathit{fred}, \mathit{father}(\mathit{fred}))$, $\mathit{father}(\mathit{Likes}(\mathit{fred}, \mathit{bananas}))$. An *atomic formula*, sometimes simply called an *atom* is a predicate symbol applied to a tuple of terms. Thus, $\mathit{Likes}(\mathit{fred}, \mathit{bananas})$ $\mathit{Likes}(\mathit{fred}, \mathit{father}(\mathit{fred}))$, $\mathit{Likes}(x, \mathit{father}(x))$ are all examples of atoms. Finally, a *ground atomic formula* or a *ground atom* is an atom without any variables. Well-formed formulæ (wffs) are then formed using the following rules:

1. Any ground atomic formula is a wff;
2. If α is a wff then $\neg\alpha$ is a wff;
3. If α and β are wffs then $(\alpha \wedge \beta), (\alpha \vee \beta)$, and $(\alpha \leftarrow \beta)$ are wffs; and
4. If α is wff containing a constant c and $\alpha^{c/x}$ be the result of replacing one or more occurences of c with a variable x that does not appear in α. Then $\forall x \alpha^{c/x}$ and $\exists x \alpha^{c/x}$ are wffs.

Rules 1–3 are like their propositional counterparts (page 15). Rule 4 is new, and requires further explanation. It is the only way variables are introduced into a formula. As an example, take the following statement: $(\mathit{Human}(\mathit{fred}) \wedge \mathit{Likes}(\mathit{fred}, \mathit{bananas}))$. That this a wff follows from an application of Rules 1 and 3. The following formulæ are all wffs, following a single application of Rule 4:

$$\begin{gathered}
\forall x(\mathit{Human}(x) \wedge \mathit{Likes}(\mathit{fred}, \mathit{bananas})) \\
\exists x(\mathit{Human}(x) \wedge \mathit{Likes}(\mathit{fred}, \mathit{bananas})) \\[4pt]
\forall x(\mathit{Human}(\mathit{fred}) \wedge \mathit{Likes}(x, \mathit{bananas})) \\
\exists x(\mathit{Human}(\mathit{fred}) \wedge \mathit{Likes}(x, \mathit{bananas})) \\[4pt]
\forall x(\mathit{Human}(x) \wedge \mathit{Likes}(x, \mathit{bananas})) \\
\exists x(\mathit{Human}(x) \wedge \mathit{Likes}(x, \mathit{bananas})) \\[4pt]
\forall x(\mathit{Human}(\mathit{fred}) \wedge \mathit{Likes}(\mathit{fred}, x)) \\
\exists x(\mathit{Human}(\mathit{fred}) \wedge \mathit{Likes}(\mathit{fred}, x))
\end{gathered}$$

A single application of Rule 4 therefore only introduces a single new variable. Subsequent applications will introduce more. For example, take the first formula above: $\forall x(\mathit{Human}(x) \wedge \mathit{Likes}(\mathit{fred}, \mathit{bananas}))$. The following wffs all result from applying Rule 4 to this statement:

$$\begin{gathered}
\forall y \forall x(\mathit{Human}(x) \wedge \mathit{Likes}(y, \mathit{bananas})) \\
\exists y \forall x(\mathit{Human}(x) \wedge \mathit{Likes}(y, \mathit{bananas})) \\[4pt]
\forall y \forall x(\mathit{Human}(x) \wedge \mathit{Likes}(\mathit{fred}, y)) \\
\exists y \forall x(\mathit{Human}(x) \wedge \mathit{Likes}(\mathit{fred}, y))
\end{gathered}$$

As with propositional logic, it is acceptable to drop outermost brackets:

$$(\forall x(Ape(x) \leftarrow Human(x)) \wedge \exists x(Ape(x) \wedge \neg Human(x)))$$

can be written as:

$$\forall x(Ape(x) \leftarrow Human(x)) \wedge \exists x(Ape(x) \wedge \neg Human(x))$$

Clausal Form

We are now in a position to expand on the notion of clauses and literals, first introduced on page 25. Consider the conditional statement:

$$\forall x(Ape(x) \leftarrow Human(x))$$

Recall the following from page 24:

$$(\alpha \leftarrow \beta) \equiv (\alpha \vee \neg\beta)$$

This means:

$$\forall x(Ape(x) \leftarrow (Human(x))) \equiv \forall x(Ape(x) \vee \neg Human(x))$$

The term in brackets on right-hand side is an example of a *clause* in first-order logic. In general, formulæ consisting of universally-quantified clauses all look alike:[23]

$$\forall x_1 \forall x_2 \ldots (\alpha_1 \wedge \alpha_2 \ldots)$$

That is, they consist of a prefix that consists only of universally quantifiers, and each α_i, or clause, is a quantifier-free formula that looks like:

$$\alpha_i = (\beta_1 \vee \beta_2 \vee \ldots \beta_n)$$

where each β_j, or *literal*. Thus, as in propositional logic, a clause is a disjunction of literals. Each literal, however, is not a proposition, but is either an atomic formula (like $Ape(x)$, sometimes called a *positive* literal) or a negated atomic formula (like $\neg Human(x)$, sometimes called a *negative* literal). It is sometimes convenient to use $\forall \mathbf{x}$ to denote $\forall x_1 \forall x_2 \cdots$ and to adopt a set-based notation to represent a clausal formula: $\{\forall \mathbf{x}\alpha_1, \forall \mathbf{x}\alpha_2, \ldots\}$. Here it is understood that the formula stands for a conjunction of clauses. Often, the quantification is taken to be understood and left out. Further, individual clauses are themselves sometimes written as sets of literals:

$$\alpha_i = \{\beta_1, \beta_2, \ldots, \beta_n\}$$

Clausal forms are of particular interest, computationally speaking. The language of logic programs (usually written in the Prolog language). is, at least in its ‘pure’ form, equivalent to clausal-form logic. Here is an example:

[23]We will take a few liberties here by not including some brackets.

Logic Program in Prolog	Clausal Form
grandfather(X,Y):- father(X,Z), parent(Z,Y).	$\{\forall x \forall y \forall z (Grandfather(x,y) \vee \neg Father(x,z) \vee \neg Parent(z,y)),$
father(henry,jane).	$Father(henry, jane),$
parent(jane,john).	$Parent(jane, john)\}$

There are three clauses in this example. The first clause has three literals and the remainder have one literal each. Further, each clause has exactly one positive literal: such clauses are called *definite* clauses. More generally, clauses that contain *at most* one positive literal are called *Horn* clauses. From now on, we may sometimes write clauses in a lazy manner that is somewhere in between the syntax of Prolog and the true clausal form:

Logic Program in Prolog

```
grandfather(X,Y):- father(X,Z), parent(Z,Y).
```

Clausal Form

$\{\forall x \forall y \forall z (Grandfather(x,y) \ \vee \ \neg Father(x,z) \ \vee \ \neg Parent(z,y))\}$

Lazy Clausal Form

$Grandfather(x,y) \leftarrow Father(x,z), Parent(z,y)$

Skolem Functions and Constants

Skolemization refers to a process of replacing an existentially quantified variable in a formula by a new term; it is merely the process of providing a *name* for something that already exists. Whether the new term is a functional expression or a constant depends on where the existential quantifier appears in the formula. If it is preceded by one or more universal quantifiers, like:

$$\forall x_1 \ldots \forall x_n \exists y \ \alpha$$

then a single skolemization step replaces all occurences of y in α by the functional expression $\mathbf{f}(x_1, \ldots, x_n)$. Here $\mathbf{f}(\cdots)$ is a function symbol, called a *Skolem function*, that does not appear anywhere in the formula. Thus, skolemization of the formula $\forall x \exists y Likes(x,y)$ results in $\forall x Likes(x, \mathbf{f}(x))$. In general, if the existential quantifier appears in between some universal quantifiers:

$$\forall x_1 \ldots \forall x_{i-1} \exists y \forall x_{i+1} \cdots \forall x_n \ \alpha$$

then all occurences of y in α can be replaced by the functional expression $\mathbf{f}(x_1, \ldots, x_{i-1})$. A special case arises if the existential quantifier precedes zero or more universal quantifiers:

$$\exists y \forall x_1 \ldots \forall x_n \; \alpha$$

In this case, a single skolemization step replaces all occurences of y in α by a Skolem function of arity 0 (that is, a constant) $\mathbf{c}$. Here $\mathbf{c}$ is a constant symbol, called a *Skolem constant*, that does not appear anywhere in the formula. Thus, skolemization of the formula $\exists y \forall x Likes(x, y)$ results in $\forall x Likes(x, \mathbf{c})$. You can see that in both cases, a single step of Skolemization reduces the number of existential quantifiers in a formula ϕ by 1. Let us denote this single step by $s(\phi)$. It should be easy to see that repeatedly performing Skolemization steps (that is, $s(s(\cdots s(\phi))))$ will result in a formula with just universal quantifiers.

Normal Forms

Universally-quantified clausal forms are a special case of specific kind of normal form for first-order formula, which we are now able to present, having described the process of Skolemization. A formula is said to be in *prenex normal form* or PNF if all its quantifiers are in front. That is, the formula looks something like $Q_1 x_1 \ldots Q_n x_n \phi$, where the Q_i is either a $\forall$ or $\exists$. Further, if ϕ is a conjunction of disjunction of literals, then the formula is said to be in *conjunctive prenex normal form*. It can be shown that every first-order formula ϕ can be expressed by an equivalent one ϕ' in conjunctive PNF. Here is an example: suppose we want to find the conjunctive PNF for $\phi : (\exists x \forall y DisLikes(x, y) \wedge \forall x \exists y Likes(x, y))$. We first rename variables to give $\phi' : (\exists x \forall y DisLikes(x, y) \wedge \forall u \exists v Likes(u, v))$. We can then move the quantifiers to the left giving: $\phi'' : \exists x \forall y \forall u \exists v (DisLikes(x, y) \wedge Likes(u, v))$, which is in conjunctive PNF.

For reasons that will become apparent, we are further interested here only in formulae in conjunctive PNFs in which all the quantifiers are universal ones (that is, $\forall$). Such formulæare said to be in *Skolem normal form*, or SNF. A SNF can be obtained from the conjunctive PNF by applying the Skolemization process to eliminate $\exists$ quantifiers. For example, suppose we want to “Skolemize” the formula ϕ above. We first find the conjunctive PNF (ϕ'' above). Using the Skolemization procedure described earlier, we replace x by a Skolem constant and v by a Skolem function. The SNF of ϕ is $\phi^S : \forall y \forall u (Dislikes(\mathbf{c}, y) \wedge Likes(u, \mathbf{f}(u)))$.

We can now return to clausal forms. Here are the steps for converting a formula into a set of clauses in clausal form:

1. Rename variables to ensure there are no variables with the same names in different quantifiers;
2. Eliminate $\leftarrow$'s and iff's;
   - α_1 iff $\alpha_2 \Rightarrow (\alpha_1 \leftarrow \alpha_2) \wedge (\alpha_2 \leftarrow \alpha_1)$
   - $\alpha_1 \leftarrow \alpha_2 \Rightarrow \alpha_1 \vee \neg\alpha_2$
3. Move $\neg$'s inwards;

- $\neg(\exists \mathbf{X})\alpha \Rightarrow (\forall \mathbf{X})\neg\alpha$
- $\neg(\forall \mathbf{X})\alpha \Rightarrow (\exists \mathbf{X})\neg\alpha$
- $\neg(\alpha_1 \vee \alpha_2) \Rightarrow \neg\alpha_1 \wedge \neg\alpha_2$
- $\neg(\alpha_1 \wedge \alpha_2) \Rightarrow \neg\alpha_1 \vee \neg\alpha_2$
- $\neg\neg\alpha \Rightarrow \alpha$

4. Distribute $\vee$'s over $\wedge$'s;
   - $\alpha \vee (\alpha_1 \wedge \alpha_2) \Rightarrow (\alpha \vee \alpha_1) \wedge (\alpha \vee \alpha_2)$
   - Assuming $\mathbf{X}$ does not occur in α_1: $\alpha_1 \vee (\forall\alpha_2) \Rightarrow (\forall \mathbf{X})(\alpha_1 \vee \alpha_2)$
   - Assuming $\mathbf{X}$ does not occur in α_1: $\alpha_1 \vee (\exists\alpha_2) \Rightarrow (\exists \mathbf{X})(\alpha_1 \vee \alpha_2)$
5. Distribute $\forall$'s;
   - $(\forall \mathbf{X})(\alpha_1 \wedge \alpha_2) \Rightarrow (\forall \mathbf{X})\alpha_1 \wedge (\forall \mathbf{X})\alpha_2$

   At this stage, the PNF form is generated.
6. Skolemise existentially quantified variables;
   - $(\forall X_1)(\forall X_2)\ldots(\forall X_n)(\exists Y)\alpha(Y) \Rightarrow (\forall X_1)(\forall X_2)\ldots(\forall X_n)\alpha(f(X_1, X_2, \ldots X_n))$

   If further applications of step 6 are possible, then they should be carried out.
7. Rewrite as clauses by dropping universal quantifiers; and
8. Standardise variables apart.

Here are some statements from a book by Lewis Carroll, written in first-order logic:

$S_1 : \forall x(\mathit{Scented}(x) \leftarrow \mathit{Coloured}(x)) \wedge$

$S_2 : \forall x(\mathit{DisLike}(x) \leftarrow \neg\mathit{GrownOpen}(x)) \wedge$

$S_3 : \neg(\exists x(\mathit{GrownOpen}(x) \wedge \neg\mathit{Coloured}(x))) \wedge$

$S_4 : \neg\forall x((\mathit{DisLike}(x) \leftarrow \neg\mathit{Scented}(x)))$

You should find that these sentences, when converted by the steps above, give the following set of clauses:

$C_1 : \{\neg\mathit{Coloured}(x), \mathit{Scented}(x)\}$

$C_2 : \{\mathit{GrownOpen}(y), \mathit{DisLike}(y)\}$

$C_3 : \{\neg\mathit{GrownOpen}(z), \mathit{Coloured}(z)\}$

$C_4 : \{\neg\mathit{Scented}(\mathbf{c})\}$

$C_5 : \{\neg DisLike(\mathbf{c})\}$

Recall that representing a clause by a set $\{L_1, L_2, \ldots, L_k\}$, is just short-form for the disjunction $L_1 \vee L2 \vee \cdots L_k$; and the actual clausal form for the formula $F : S_1 \wedge S_2 \wedge S_3 \wedge S_4 \wedge S_5$ is $C : \forall x \forall y \forall z (C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5)$. Is F equivalent to C? To answer this, we need to understand how meanings are assigned to first-order formulæ.[24]

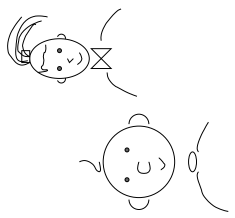
1.4.2 Semantics

As with propositional logic, the semantics of predicate logic is primarily concerned with interpretations, models, and logical consequence. Of these, it is only the notion of interpretation that requires a re-examination.

Interpretations

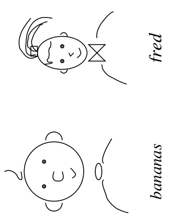
Recall that interpretations in propositional logic were simply assignments of *true* or *false* to propositional symbols. Matters are not as simple in predicate logic for two reasons. First, we have to deal with the additional complexity of expressions arising from a richer vocabulary. Thus, the truth-value (meaning) of $Likes(fred, bananas)$ will depend on the meanings of each of the symbols $Likes$, $fred$ and $bananas$. Second, we have to interpret sentences that contain quantifiers.

Informally, let us see what is required in constructing an interpretation that allows us to understand $Likes(fred, bananas)$. Before we begin, remove any pre-conceptions of what the words in the statement mean in English: for us they are simply a predicate symbol ($Likes$) and two constant symbols ($fred, bananas$) in some 'formal-world'. The first step then is to identify a domain of objects in the 'real-world'.

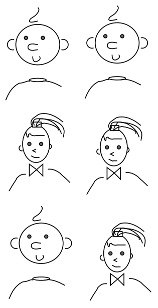


Next, we associate constant symbols in our formal-world to objects in the real-world (just to avoid any pre-conceptions, we have scrambled things a little bit):

[24]But the answer is "no": the problem, as you might have guessed, comes about because of the Skolemization step.



and associate the predicate symbols to a relation in the real-world:



We can now see that $Likes(fred, bananas)$ is *false* as the objects corresponding to the ordered pair $< fred, bananas >$ are not in the real-world relation represented by $Likes$.

Formally, an interpretation in predicate logic is a specification of:

1. A domain D;
2. A mapping of constants to elements in D;
3. A mapping of each n-argument predicate symbol to a relation on D^n, where $D^n = \{< d_1, \ldots, d_n > \mid d_i \in D\}$ is the n-fold Cartesian product; and
4. A mapping of each n-argument function symbol to a function from $D^n \rightarrow D$

You will also see sometimes that the term "structure" is used to describe what we have called an interpretation. In such cases, a distinction is made between the *vocabulary*, which consists of the constants, functions and predicate symbols; the and the *structure*, which consists of the domain D and the three mappings. We will continue to use "interpretation" to retain a similarity to propositional logic.

Given an interpretation, every atom—a predicate symbol with a tuple of terms as arguments—is assigned a truth-value according to whether the objects designated by the arguments are in the relation designated by the predicate symbol.

Well-formed formulæ in predicate logic consist of more than atoms. We also need rules for assigning truth-values to formulæ that contain logical connectives ($\neg, \wedge, \vee, \leftarrow$) and quantifiers ($\forall, \exists$). The semantics of the logical connectives in predicate logic are the same as those in propositional logic. Thus, as before, assigning meanings to formulæ with these connectives requires the use of the truth tables on page 17. For example, the formula $Human(fred) \wedge Likes(fred, bananas)$ is *true* only if the interpretation results in both the atoms $Human(fred)$ and $Likes(fred, bananas)$ being *true*. What though, of formulæ that contain quantified variables? The rules for these are:

1. Any wff $\forall x\alpha$ is *true* if and only if for every domain element that we can associate with x, α is *true*;

2. Any wff $\exists x\alpha$ is *true* if and only if for some domain element that we can associate with x, α is *true*.

Models and Logical Consequence

The meanings of these, and related concepts, are unchanged from propositional logic. Thus:

Models. Any interpretation that makes a wff *true* is called a model for that formula;

Validity and Unsatisfiability. A formula for which all interpretations are models is said to be valid. A formula for which none of the interpretations are a model is said to be unsatisfiable. A formula that has at least one model is said to be satisfiable;

Consequence. Given a conjunction of wffs Σ represented as the set $\{\beta_1, \ldots, \beta_n\}$ and a wff α if $\Sigma \models \alpha$ then every model for Σ is a model for α;

Deduction Theorem. Given a conjunction of wffs $\Sigma = \{\beta_1, \ldots, \beta_n\}$ and a wff α, $\Sigma \models \alpha$ if and only if $\Sigma - \{\beta_i\} \models (\alpha \leftarrow \beta_i)$. The proof is the same as was for that in the propositional case (*c.f.* theorem 6);

Equivalence. Given a pair of wffs α and β, if $\alpha \models \beta$ and $\beta \models \alpha$ then α and β are equivalent ($\alpha \equiv \beta$).

The following relations hold in the predicate logic (as usual, α, β are wffs):

$\neg(\alpha \vee \beta)$	$\equiv$	$\neg\alpha \wedge \neg\beta$	De Morgan's law
$\neg(\alpha \wedge \beta)$	$\equiv$	$\neg\alpha \vee \neg\beta$	De Morgan's law
$(\alpha \leftarrow \beta)$	$\equiv$	$(\alpha \vee \neg\beta)$	
$(\alpha \leftarrow \beta)$	$\equiv$	$(\neg\beta \leftarrow \neg\alpha)$	Conditional $\equiv$ Contrapositive
$\neg\forall x\alpha$	$\equiv$	$\exists x\neg\alpha$	
$\neg\exists x\alpha$	$\equiv$	$\forall x\neg\alpha$	
$\forall x\alpha$	$\equiv$	$\forall y\alpha^{x/y}$	Renaming of x by y
$\exists x\alpha$	$\equiv$	$\exists y\alpha^{x/y}$	Renaming of x by y
$\forall x\forall y\alpha$	$\equiv$	$\forall y\forall x\alpha$	
$\exists x\exists y\alpha$	$\equiv$	$\exists y\exists x\alpha$	
$\forall x(\alpha \wedge \beta)$	$\equiv$	$(\forall x\alpha \wedge \forall x\beta)$	Distributivity of $\forall$
$\exists x(\alpha \vee \beta)$	$\equiv$	$(\exists x\alpha \vee \exists x\beta)$	Distributivity of $\exists$
$\forall x\alpha$	$\models$	$\exists x\alpha$	
$(\forall x\alpha \vee \forall x\beta)$	$\models$	$\forall x(\alpha \vee \beta)$	
$\exists x(\alpha \wedge \beta)$	$\models$	$(\exists x \wedge \exists x\beta)$	
$\forall x\alpha^{y/\mathbf{f}(x)}$	$\models$	$\forall x\exists y\alpha$	Non-equivalence of Skolemized form
$\forall x\alpha^{y/\mathbf{c}}$	$\models$	$\exists y\forall x\alpha$	Non-equivalence of of Skolemized form

More on Normal Forms

We stated earlier that any first-order formula could be converted to a conjunctive prenex normal form, or conjunctive PNF. We further saw how a formula $\phi = Q_1x_1 \ldots Q_nx_n\phi_0(x_1, \ldots, x_n)$ could be "Skolemized" to give a formula ϕ^S in Skolem Normal Form, or SNF. The interest in SNFs lies in the following fact:

Theorem 15 *ϕ is satisfiable if and only if ϕ^S is satisfiable (you should be able to convince yourself that checking for satisfiability is equivalent to checking for logical consequence).*

Proof sketch: Recall that $\phi^S = s(s(\cdots s(\phi)))$ where $s(\cdot)$ denotes a single step of Skolemization. Now, it is sufficient to show that ϕ is satisfiable if and only if $s(\phi)$ is satisfiable (the full proof will follow by induction). We can also assume that ϕ is in conjunctive PNF. Now $s(\phi)$ results in either replacing a variable by a Skolem constant or by a Skolem function. Since the former is just a special case of the latter, we will just consider the case when $s(\phi)$ results in replacing a variable by a Skolem function. Let Q_i be the existential quantifier removed by the Skolemization step, and let $\psi(x_1, \ldots, x_i) = Q_{i+1} \ldots Q_n\phi_0(x_1, \ldots, x_n)$. Suppose $\forall x_1 \ldots \forall x_{i-1}\exists x_i\psi(x_1, \ldots, x_i)$ has some model M. Let $M_\mathbf{f}$ extend M by interpreting $\mathbf{f}$ in such a way that $M_\mathbf{f}$ is a model for $\psi(c_1, \ldots, c_{i-1}, \mathbf{f}(c_1, \ldots, c_{i-1}))$ for all possible values $c_1, \ldots, c_{i-1} \in M$ for the variables $x_1, \ldots, x_{i-1}$. Then, clearly, $M_\mathbf{f}$ is a model for $\forall x_1 \ldots \forall x_{i-1}\psi(x_1, \ldots, x_{i-1}, \mathbf{f}(x_1, \ldots, x_{i-1}))$. Now consider the converse. Suppose $\forall x_1 \ldots \forall x_{i-1} \psi(x_1, \ldots, x_{i-1}, \mathbf{f}(x_1, \ldots, x_{i-1}))$ has some model M then, it follows from the meaning of $\exists$ that M is also a model for $\forall x_1 \ldots \forall x_{i-1} \exists x_i \psi(x_1, \ldots, x_i)$. It follows therefore that ϕ is satisfiable if and only if $s(\phi)$ is satisfiable. $\Box$

We end this section on a note of caution: ϕ and $s(\phi)$ are not equivalent. That is, Skolemization does not preserve logical equivalence. A simple example should convince you of this. Let $\phi = \exists x First(x)$ and $s(\phi) = First(\mathbf{c})$. Clearly, we can find models for $\exists First(x)$ that are not models of $First(\mathbf{c})$.

Herbrand Interpretations

The 4-step specification above makes an interpretation in predicate logic much more elaborate than its counterpart in propositional logic (which was simply an assignment of *true* or *false* to propositions). The reference to 'real-world' objects, relations, and functions adds a further degree of complexity: how is all this to be conveyed to an automated procedure? In fact, many of these problems can be side-stepped by confining attention only to a domain that consists solely of formal symbols. Called the *Herbrand universe* (U_L), this is simply all the ground (or variable-free) terms that can be constructed using the constants and function symbols available in a first order language L. Consider as example a language that consists of:

Constant symbol:	$zero$
Predicate symbol:	$Nat/1$
Function symbols:	$pred, succ$

The Herbrand universe U_L in this instance consists of terms like $zero, pred(zero)$, $succ(zero)$, $pred(succ(zero))$, $succ(pred(zero))$ and so on. The *Herbrand base* B_L is the set of all ground atoms that can be constructed using the predicate symbols and terms from the Herbrand universe U_L. Here, the Herbrand base B_L consists of atoms like $Nat(zero), Nat(pred(zero)), Nat(succ(zero)), Nat(pred(succ(zero)))$ and so on. A Herbrand interpretation I_L is—quite like the propositional case—simply an assignment of *true* to some subset of B_L and *false* to the rest. In fact, it is common practice to associate 'Herbrand interpretation' only with the subset assigned *true*: it being understood that all other atoms in the Herbrand base are assigned *false*. Thus, $\{Nat(zero)\}$ is an I_L that assigns *true* to $Nat(zero)$ and *false* to all other atoms in B_L.

Herbrand Models

Since a model is an interpretation that makes a well-formed formula *true*, a *Herbrand model* M_L is simply a Herbrand interpretation I_L that makes a well-formed formula *true*. Let us return to the example presented earlier:

Constant symbol:	$zero$
Predicate symbol:	$Nat/1$
Function symbols:	$pred, succ$

Recall from page 68, that:

Herbrand universe (U_L):	$\{zero, pred(zero), succ(zero), pred(succ(zero)), \ldots\}$
Herbrand base (B_L):	$\{Nat(zero), Nat(pred(zero)), Nat(succ(zero)), \ldots\}$

Further, a Herbrand interpretation is simply a subset of the Herbrand base containing all atoms that are *true*. Thus, $I_1 = \{Nat(zero)\}$ is a Herbrand interpretation in which $Nat(zero)$ is *true* and all other atoms in the Herbrand base are are *false*. We can now examine whether I_1 is a Herbrand model for the formula:

$$\Sigma_1 : Nat(zero) \;\wedge\; \forall x(Nat(succ(x)) \leftarrow Nat(x))$$

Being a conjunctive expression, we require I_1 to be a Herbrand model for both $Nat(zero)$ and $\forall x(Nat(succ(x)) \leftarrow Nat(x))$. I_1 is clearly a model for $Nat(zero)$ as this atom is assigned *true* in the interpretation. But what about the conditional? The rule for the universal quantifier (page 66) dictates that the conditional statement is *true* if it is *true* for every element that the variable x can be associated with. In other words, the interpretation is a model for every element that x can be associated with. In the Herbrand world, x can be associated with any element of the Herbrand universe ($zero, succ(zero), pred(zero)$ and so on). Suppose x was associated with $zero$. Then we would require I_1 to be a model for $Nat(succ(zero)) \leftarrow Nat(zero)$. Since I_1 assigns $Nat(succ(zero))$ to *false* and $Nat(zero)$ to *true*, I_1 is not a model for $Nat(succ(zero)) \leftarrow Nat(zero)$ (line 2 in the truth-table for the conditional on page 17). Thus, I_1 is not a Herbrand model for $\forall x(Nat(succ(x)) \leftarrow Nat(x))$ and in turn for Σ_1. Consider, on the other hand, the formula:

$$\Sigma_2 : Nat(zero) \;\wedge\; \forall x(Nat(x) \leftarrow Nat(pred(x)))$$

As before, suppose x was associated with $zero$. The conditional then becomes $Nat(zero) \leftarrow Nat(pred(zero))$. With interpretation I_1, $Nat(zero)$ is *true* and $Nat(pred(zero))$ is *false*. I_1 is therefore a model for this formula (line 3 in the truth table for the conditional). All other associations for x result in both sides of the conditional being *false* and I_1 being a model for each such formula (line 1 in the truth table). Thus, I_1 makes $\forall x(Nat(x) \leftarrow Nat(pred(x)))$ *true* for every element that x can be associated with, and is a model for it and in turn for Σ_2.

Herbrand models are particularly relevant to the study of clausal forms (page 60). Recall that these are conjunctions of clauses, each of which contains only universally quantified variables and consists of a disjunction of literals. Both Σ_1 and Σ_2 above can be written in clausal form:

$$\Sigma_1{}' : Nat(zero) \;\wedge\; \forall x(Nat(succ(x)) \vee \neg Nat(x))$$

$$\Sigma_2{}' : Nat(zero) \;\wedge\; \forall x(Nat(x) \vee \neg Nat(pred(x)))$$

The *ground instantiation* of a clausal formula is the conjunction of ground (variable-free) clauses that result by replacing variables with terms from the Herbrand universe. For example, the ground instantiation of ${\Sigma_2}'$ is:

$$\begin{array}{lll}
\mathcal{G}({\Sigma_2}'): & Nat(zero) & \wedge \\
 & (Nat(zero) \vee \neg Nat(pred(zero))) & \wedge \\
 & (Nat(pred(zero)) \vee \neg Nat(pred(pred(zero)))) & \wedge \\
 & (Nat(succ(zero)) \vee \neg Nat(pred(succ(zero)))) & \wedge \\
 & \ldots &
\end{array}$$

You can therefore think of the ground instantantiation as making explicit the meaning of the universal quantifier $\forall$. Now, it should be clear that a Herbrand interpretation will determine the truth-value for all clauses in the ground instantiation of a clausal formula.

We present another example illustrating Herbrand models. Consider the following program P:

$$\begin{array}{l}
likes(john, X) \leftarrow likes(X, apples) \\
likes(mary, apples) \leftarrow
\end{array}$$

Suppose the language $\mathcal{L}$ contained no symbols other than those in P. Then, $\mathcal{B}(P)$ is the set $\{likes(john, john), likes(john, apples), likes(apples, john), likes(john, mary), likes(mary, john), likes(mary, apples), likes(apples, mary), likes(mary, mary), likes(apples, apples)\}$. Now, $\{likes(mary, apples), likes(john, mary)\}$ is a subset of $\mathcal{B}(P)$, and is a Herbrand interpretation. Moreover, it is also a Herbrand model for P. Similarly, $\{likes(mary, apples), likes(john, mary), likes(mary, john)\}$ is also a model for P. The ground instantiation $\mathcal{G}(P)$ for this program is:

$$\begin{array}{l}
likes(john, john) \leftarrow likes(john, apples) \\
likes(john, mary) \leftarrow likes(mary, apples) \\
likes(john, apples) \leftarrow likes(apples, apples) \\
likes(mary, apples) \leftarrow
\end{array}$$

It can be verified[25] that the interpretation $\{likes(mary, apples), likes(john, mary)\}$ is a model for the $\mathcal{G}(P)$ above.

The importance of Herbrand models for clausal formulæ stems from the following property:

Theorem 16 *A clausal formula Σ has a model if and only if its ground instantiation $\mathcal{G}(\Sigma)$ has a Herbrand model.*

Proof: $\Rightarrow$: Suppose Σ has a model M. Then we define the following Herbrand interpretation I as follows. Let P be an n-ary predicate symbol occurring in Σ. Then we define the function I_P from U_L^n to $\{T, F\}$ as follows: $I_P(t_l, \ldots, t_n) = T$ if $P(t_1, ..., t_n)$ is true under M, and $I_P(t_1, ..., t_n) = F$ otherwise. It can easily be shown that $I = \cup_{P \in \Sigma} I_P$ is a Herbrand model of Σ.

$\Leftarrow$: This is obvious (a Herbrand model is a model). $\Box$

[25]EXERCISE.

In other words, there must be *some* assignment of truth-values to atoms in the Herbrand base that makes all clauses in Σ *true*. In the example above, the Herbrand interpretation $I_1 = \{Nat(zero)\}$ that assigns $Nat(zero)$ to *true* and everything else to *false*, is clearly a model for $\mathcal{G}({\Sigma_2}')$. Therefore, from the property stated here, we can say that ${\Sigma_2}'$ has a model.

If we are dealing only with a *definite clausal formula*— a clausal formula in which all clauses have exactly one positive literal (${\Sigma_1}'$ and ${\Sigma_2}'$ are both of this type)—then more is known about the Herbrand models of the formula. Recall that a Herbrand model is nothing more than a set of ground atoms, which when assigned *true*, make the formula *true*.

1.4.3 From Datalog to Prolog

The statement *"Any animal that has hair is a mammal"* can be written as a clause using monadic predicates (*i.e.* predicates with arity 1):

$$\forall X \ \textit{is\_mammal(X)} \leftarrow \textit{has\_hair(X)}$$

Usually clauses are written without explicit mention of the quantifiers:

$$\textit{is\_mammal(X)} \leftarrow \textit{has\_hair(X)}$$

$$\textit{is\_mammal(X)} \leftarrow \textit{has\_milk(X)}$$

$$\textit{is\_bird(X)} \leftarrow \textit{has\_feathers(X)}$$

...

Datalog

Datalog is a subset of the language of first order language; it has all the components of first order logic (variables, constants and recursion), except functions. A Datalog "expert" system will encode these rules using monadic predicates as:

```
is_mammal(X) :- has_hair(X).
is_mammal(X) :- has_milk(X).
is_bird(X) :- has_feathers(X).
is_bird(X) :- can_fly(X), has_eggs(X).
is_carnivore(X) :- is_mammal(X), eats_meat(X).
is_carnivore(X) :- has_pointed_teeth(X), has_claws(X), has_pointy_eyes(X).
cheetah(X) :- is_carnivore(X), has_tawny_colour(X), has_dark_spots(X).
tiger(X) :- is_carnivore, has_tawny_colour(X), has_black_stripes(X).
penguin(X) :- is_bird(X), cannot_fly(X), can_swim(X).
```

Now here are some statements[26] particular to animals:

```
has_hair(peter).                fat(peter).
has_green_eyes(peter).          has_tawny_colour(peter).
eats_meat(peter).               has_black_stripes(peter).
has_milk(bob).                  eats_meat(bob)
has_tawny_colour(bob).          has_dark_spots(bob).
can_fly(bob).
```

[26]EXERCISE: What are the logical consequences of all the clauses?

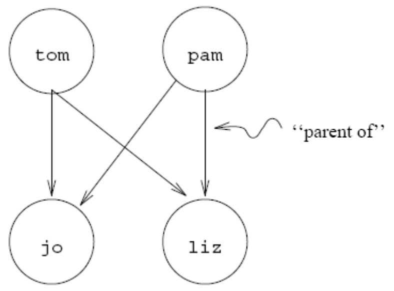


Figure 1.11: Graph representing 'parent of' relation.

However, monadic predicates: not expressive enough. While monadic predicates lets us make statements like *"Every son has a parent"*:

$$\forall X \exists Y \; parent(Y) \;\leftarrow\; son(X)$$

for more complex relationships, we will need predicates of arity > 1. Usually, relationships can be described pictorially by a directed acyclic graph (DAG) as in Figure 1.11 The parent-child relation could also be specified as a set of ordered pairs $< X, Y >$, or, as a set of definite clauses

$$\begin{aligned} &parent(tom, jo) \;\leftarrow \\ &parent(pam, jo) \;\leftarrow \\ &parent(tom, liz) \;\leftarrow \\ &parent(pam, liz) \;\leftarrow \end{aligned}$$

Consider the *predecessor* relation, namely, all ordered tuples $< X, Y >$ *s.t.* X is an ancestor of Y. This set will include Y's parents, Y's grandparents, Y's grandparents' parents, etc.

$$\begin{aligned} pred(X,Y) &\leftarrow parent(X,Y) \\ pred(X,Z) &\leftarrow parent(X,Y), parent(Y,Z) \\ pred(X,Z) &\leftarrow parent(X,Y1), parent(Y1,Y2), parent(Y2,Z) \\ \ldots & \end{aligned}$$

As can be seen through this example, variables and constants are not enough: we need *recursion*:

$\forall X, Z$ X is a predecessor of Z if
1. X is a parent of Z; or
2. X is a parent of some Y, and Y is a predecessor of Z

The predecessor relation is thus

$$\begin{aligned} pred(X,Y) &\leftarrow parent(X,Y) \\ pred(X,Z) &\leftarrow parent(X,Y), pred(Y,Z) \end{aligned}$$

and can be pictorially depicted as in 1.12

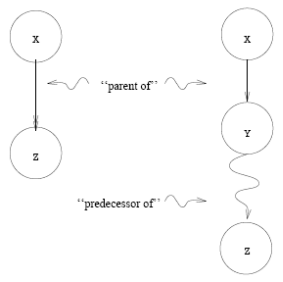


Figure 1.12: The predecessor relation.

Prolog = Predicates + Variables + Constants + Functions

Datalog (first order logic without functions) is however not expressive enough. To express arithmetic operations, lists of objects, etc. it is not enough to simply allow variables and constants as terms. We will also need *function* symbols as supported in Prolog.

Consider Peano's postulates for the set of natural numbers $\mathcal{N}$.

1. The constant 0 is in $\mathcal{N}$
2. if X is in $\mathcal{N}$ then $s(X)$ is in $\mathcal{N}$
3. There are no other elements in $\mathcal{N}$
4. There is no X in $\mathcal{N}$ $s.t.$ $s(X) = 0$
5. There are no X, Y in $\mathcal{N}$ $s.t.$ $s(X) = s(Y)$ and $X \neq Y$

We can write a definite clause definition using 1 constant symbol and 1 unary function symbol for enumerating the elements of $\mathcal{N}$:

$$natural(0) \leftarrow$$
$$natural(s(X)) \leftarrow natural(X)$$

The elements of $\mathcal{N}$ can be now generated by asking:

natural(N)?

Prolog also supports lists. Lists are simply collections of objects. For e.g. $1, 2, 3\ldots$ or $1, a, dog, \ldots$. Lists are defined as follows:

1. The constant *nil* is a list
2. If X is a term, and Y is a list then $.(X, Y)$ is a list

So the list $1, 2, 3$ is represented as:

$$.(1, .(2, .(3, nil)))$$

Usually logic programming systems use a "[" "]" notation, in which the constant *nil* is represented as [] and the list 1, 2, 3 is [1, 2, 3]. In this notation, the symbol | is used to separate a list into a "head" (the elements to the left of the |) and a "tail" (the list to the right of the |). Thus:

List	Represented as	Values of variables
$[1, 2, 3]$	$[X\|Y]$	$X = 1, Y = [2, 3]$
$[[1, 2], 3]$	$[X\|Y]$	$X = [1, 2], Y = [3]$
$[1]$	$[X\|Y]$	$X = 1, Y = []$
$[1\|2]$	$[X\|Y]$	$X = 1, Y = 2$
$[1]$	$[X, Y]$	
$[1, 2, 3]$	$[X, Y\|Z]$	$X = 1, Y = 2, Z = [3]$

1.4.4 Lattice of Herbrand Models

The discussion in this section is more or less similar to the discussion in Section 1.3.8 and the reader is referred to the proofs in that Section for proofs of most statements that will be made in this section. The only difference is that while Section 1.3.8, in this section, we will talk about Herbrand models.

For a definite clausal formula, it can be shown that the set H of Herbrand models is a complete lattice ordered by set-inclusion (that is, for Herbrand models $M1, M2 \in H$, $M1 \preceq M2$ if and only if $M1 \subseteq M2$), and with the binary operations of $\cap$ and $\cup$ as the glb and lub respectively. Recall that a complete lattice has a unique least upper bound and a unique greatest lower bound. Since we are really talking about sets that are ordered by set-inclusion, this means that there is a a unique *smallest* one (the size being measured by the number of elements). This is called the *minimal model* of the formula, and it can be shown that this must be the intersection of all Herbrand models for the formula.

In the example above, ${\Sigma_2}'$ has several Herband models ($\{Nat(zero)\}$ and $\{Nat(zero), Nat(pred(zero))\}$ are two examples). Of these $\{Nat(zero)\}$ is the smallest, and is the minimal model. There is an important result relating a definite clausal formula Σ, its minimal model $\mathcal{MM}(\Sigma)$ and the ground atoms that are logical consequences of Σ:

Theorem 17 *If α is a ground atom then $\Sigma \models \alpha$ if and only if $\alpha \in MM(\Sigma)$.*

Here $MM(\cdot)$ denotes the minimal model. Thus, the minimal model of a definite clausal formula is identical to the set of all ground atoms logically implied by that formula. Thus, the minimal model provides, in effect, denotes the meaning (or semantics) of the formula. The proof of this theorem follows nearly from theorem 12 that was proved earlier.

We can envisage a procedure for enumerating the Herbrand models of a formula. Consider the powerset of the Herbrand base of the formula. Now, we know that this powerset ordered by $\subseteq$ necessarily forms a complete lattice, with binary operations $\cap$ and $\cup$. Some subset of this powerset is the set of all Herbrand models, which we know is also a lattice ordered by $\subseteq$ with the same binary operations. So, the model lattice is a sublattice of the lattice obtained from the powerset of the Herbrand base. Suppose now we start at some point s in this sublattice, and we move to a new point that consists only of those ground atoms of the formula made true by the model s. Let us call these atoms s_1. Then, a little thought should convince you that s_1 is also a member of the sublattice of Herbrand models. Repeating the process with s_2 we can move to models s_2, s_3 and so on. Will this procedure converge eventually on the minimal model? Not necessarily, since we could end up moving back-and-forth between points of the sub-lattice. (When will this happen, and how can we ensure that we do converge on the minimal model?).

A slightly more general process can be formalised as the application of a function T_P that, for a clausal formula P, generates an interpretation (not necessarily a model) from another. That is:

$$I_{k+1} = T_P(I_k)$$

where

$$T_P(I) = \{a : a \leftarrow \mathit{body} \in \mathcal{G}(P) \text{ and } \mathit{body} \in I\}$$

where $\mathcal{G}(P)$ is the ground instantiation of P as before. It can be shown that T_P is both monotonic and continuous on the complete lattice obtained by ordering the powerset of the Herbrand base by $\subseteq$. So, we know from the Knaster-Tarski Theorem mentioned on page 11, that there must be a least fixpoint for T_P in this lattice. We can prove that the procedure of obtaining I_{k+1} from application of T_P to I_k will yield that fixpoint, and further, that this fixpoint will be the minimal model. As an inference procedure though, it is not really very practical: especially if all we needed to do is check if a particular atom was a logical consequence. It gets worse if the minimal model is not finite, in which case the procedure may not terminate in a finite number of steps. For all these reasons, we will need to do better.

1.4.5 Inference

Consider the following set of clauses S:

$$\begin{array}{l} likes(john, flowers) \leftarrow \\ likes(mary, food) \leftarrow \\ likes(mary, wine) \leftarrow \\ likes(john, wine) \leftarrow \\ likes(john, mary) \leftarrow \\ likes(paul, mary) \leftarrow \end{array}$$

If you entered these clauses into a program capable of executing logic programs (some implementation of Prolog), and asked:

$$likes(john, X)?$$

you will get a number of answers:

$$\begin{array}{l} X = flowers \\ X = wine \\ X = mary \end{array}$$

On the other hand, if the query were

$$likes(john, X), likes(mary, X)?$$

the answer should be:

$$X = wine$$

How this works will be examined in shortly. For now, consider *likes(john,X)?*. An intuitive procedure will be:

1. Start search from 1st clause
2. Search for any clause whose head has predicate $likes/2$, and 1st argument is $john$
3. If no clause is found *return* otherwise *goto* 4
4. X is associated ("instantiated") with the 2nd argument of the head literal, the clause position marked, and the value associated with X is output
5. Start search from clause marked, and *goto* 2

As in the propositional case, we will only be concerned here with the rule of resolution. In a broad sense, this remains similar to its propositional counterpart (page 29: it applies to clauses with a pair of complementary literals, and the result (or resolvent) is a clause with the complementary pair removed. However the intricacies of predicate logic require a bit more care. Take the following pair of conditionals (and their clausal forms):

Conditional	Clausal Form
$\forall x(Ape(x) \leftarrow Human(x))$	$\forall x(Ape(x) \vee \neg Human(x))$
$Human(fred) \leftarrow Human(father(fred))$	$Human(fred) \vee \neg Human(father(fred))$

For resolution to apply, we require the clausal forms to contain a pair of complementary literals. We nearly do have such a pair: $\neg Human(x)$ in the first clause and $Human(fred)$ in the second. It is apparent that if variable x in the first clause were to be restricted to the term $fred$, then we would indeed have a complementary pair, and the resolvent is:

Resolvent	Clausal Form
$Ape(fred) \leftarrow Human(father(fred))$	$Ape(fred) \vee \neg Human(father(fred))$

A single resolution step in predicate logic thus involves 'substituting' terms for variables so that a complementary pair of literals results. Here, such a pair would result if we could somehow 'match' the literals $Human(x)$ and $Human(fred)$. The resulting mapping of variables to terms is called the *unifier* of the two literals. Thus, mapping x to $fred$ is a unifier for the literals $Human(x)$ and $Human(fred)$.

Substitution

More generally, a *substitution* is a mapping from variables to terms that is usually denoted as $\theta = \{v_1/t_1, v_2/t_2, \ldots, v_n/t_n\}$. Applying a substitution θ to a well-formed formula α results in a *substitution instance*, usually denoted by $\alpha\theta$. Thus, applying the substitution $\theta = \{x/fred\}$ to $\alpha : \forall x(Ape(x) \vee \neg Human(x))$ results in the substitution instance $\alpha\theta : (Ape(fred) \vee \neg Human(fred))$. We usually require substitutions to have the following properties:

1. They should be *functions*. That is, each variable to the left of the / should be distinct. Thus, $\{x/fred, x/bill\}$ is not a legal substitution; and

2. They should be *idempotent*. That is, each term to the right of the / should not contain a variable that appears to the left of the /. Thus, $\{x/father(x)\}$ is not a legal substitution. This test is sometimes called the "occurs-check". The occur-check disallows self-referential bindings such as $X/f(X)$. However, the temptation to omit the occur-check in unification algorithms is very strong, owing to the high processing cost of including it; it is the only test in the comparison cycle which has to scrutinize the inner contents of terms, whereas all other tests examine only the terms' principal (outermost) symbols.

A pair of substitutions can be *composed* ('joined together'). For example, composing $\{x/father(y)\}$ with $\{y/fred\}$ results in $\{x/father(fred)\}$. In general, the result of composing substitutions

$$\theta_1 = \{u_1/s_1, \ldots, u_m/s_m\}$$

$$\theta_2 = \{v_1/t_1, \ldots, v_n/t_n\}$$

is (this may not be a legal substituition):

$$\theta_1 \circ \theta_2 = \{u_1/s_1\theta_2, \ldots, u_m/s_m\theta_2\} \cup \{v_i/t_i | v_i \notin \{u_1, \ldots, u_m\}\}$$

Theorem 18 *If α is a universally quantified expression that is not a term (i.e., a literal or a conjunction or disjunction of literals), and θ is a substitution, then the following holds: $\alpha \models \alpha\theta$. For example, $P(x) \vee \neg Q(y) \models P(a) \vee \neg Q(y)$, where we have used the substitution $\{x/a\}$.*

Proof sketch: The proof for this example is easy: suppose I is a model, with domain D, of $P(x) \vee \neg Q(y)$. Then for all $d_1 \in D$, and for all $d_2 \in D$, $I_P(d_1) = T$ or $I_Q(d_2) = F$. Suppose a is mapped to domain element d by I, then for all $d \in D$, $I_P(d) = T$ or $I_Q(d) = F$. Hence I is a model of $P(a) \vee \neg Q(y)$. It is clear that for different α or θ, a similar proof can always be given. Hence always $\alpha \models \alpha\theta$. $\Box$

Unifiers

We are now in a position to state more formally the notion of unifiers. To say that a substitution θ is a unifier for formulæ α_1 and α_2 means $\alpha_1\theta = \alpha_2\theta$. However, there can be many unifiers. For example, the formulæ $\alpha_1 : \forall x \forall z Parent(father(x), z)$ and $\alpha_2 : \forall y Parent(y, fred)$ have as unifiers $\theta_1 = \{x/fred, y/father(fred), z/fred\}$ and $\theta_2 = \{y/father(x), z/fred\}$. In the first case $\alpha_1\theta_1 = \alpha_2\theta_1 = Parent(father(fred), fred)$; and in the second case $\alpha_1\theta_2 = \alpha_2\theta_2 = \forall x Parent(father(x), x)$. Notice that θ_2 is, in some sense, more 'general' than θ_1 as it imposes less severe constraints on the variables. There is, in fact, a *most general unifier* (or mgu) for a pair of formulæ. The substitution θ is a most general unifier for α_1 and α_2 if and only if:

1. $\alpha_1\theta = \alpha_2\theta$ (that is, θ is a unifier for α_1 and α_2); and
2. For any other unifier σ for α_1 and α_2, there is a substitution μ such that $\sigma = \theta \circ \mu$ (that is, $\alpha_1\sigma$ is a substitution instance of $\alpha_1\theta$).

In the example just shown, θ_2 is the most general unifier.

Returning now to resolution, we can state the main steps involved for a pair of clauses C_1 and C_2:

1. Rename all variables in clause C_2 so that they cannot be confused with those in C_1 (for the variables in C_2 are independent of those in C_1 and the renamed clause is equivalent to C_2). This is sometimes called "standardising the clauses apart";
2. Identify complementary literals and see if an mgu exists;
3. Apply mgu and form the resolvent C.

Here is an example:

Formula	Clausal Form
$C_1 : \forall x(Ape(x) \leftarrow Human(x))$	$\forall x(Ape(x) \vee \neg Human(x))$
$C_2 : \forall x(Human(x) \leftarrow Human(father(x)))$	$\forall x(Human(x) \vee \neg Human(father(x)))$

The 3 steps above are:

1. Standardise apart. The two clauses are now:

$$\begin{aligned} &C_1 : \forall x(Ape(x) \vee \neg Human(x)) \\ &C_2 : \forall y(Human(y) \vee \neg Human(father(y))) \end{aligned}$$

2. Identify complementary literals and mgu. It is evident that $\neg Human(x)$ in C_1 and $Human(y)$ in C_2 are complementary. Their mgu is $\theta = \{x/y\}$;

3. Apply mgu and form resolvent. The resolvent C is as shown below:

$$C : \forall x(Ape(x) \vee \neg Human(father(x)))$$

As with propositional logic, the set-based notation used for clauses (page 60) allows us to present resolution in a compact (algebraic) form:

$$R = (C_1 - \{L\})\theta \cup (C_2 - \{M\})\theta$$

The difference to propositional logic is, of course, the appearance of θ, the mgu of literals L and $\neg M$. In fact, there is another problem that we have avoided. Suppose our clauses C_1 and C_2 are $C_1 : \forall x \forall y(Human(x) \vee Human(y))$ and $C_2 : \forall u \forall v(\neg Human(u) \vee \neg Human(v))$. Now it is clear that $\{C_1, C_2\}$ is unsatisfiable. But, unfortunately, we will not be able to get to the empty clause $\Box$ using resolution as we have just described it. Here is one possible resolvent: $R : \forall y \forall v(Human(y) \vee \neg Human(v))$. In fact, every possible resolvent of the two clauses will contain two literals, as will resolvents using those resolvents, and so on. What we really want to do is to eliminate redundant literals in any clause. For example, C_1 should really just be $\forall x Human(x)$ and C_2 should really just be $\forall u Human(u)$. The procedure that removes redundant literals in this manner is called *factoring*.

Factoring

Formally, if C is a clause, $L_1, \ldots, L_n (n \geq 1)$ some unifiable literals from C, and θ an mgu for the set $\{L_1, \ldots, L_n\}$, then the clause obtained by deleting $L_2\theta, \ldots, L_n\theta$ from $C\theta$ is called a *factor* of C. For example, $Q(a) \vee P(f(a))$ is a factor of the clause $\neg Q(a) \vee P(f(a)) \vee P(y)$ using $\{y/f(a)\}$ as an mgu for $\{P(f(a)), P(y)\}$. Also, $Q(x) \vee P(x, a)$ is a factor of $Q(x) \vee Q(y) \vee Q(z) \vee P(z, a)$.

Operationally, it finds a substitution that unifies one or more literals in a clause, and retains only a single copy of the unified literals. Semantically speaking, a literal L is redundant in a clause C, if it is equivalent to a clause without that literal. That is $C - \{L\} \equiv C$. Note that every non-empty clause C is a factor of C itself, using the empty substitution $\emptyset$ as mgu for one literal in C. It can easily be shown if C' is a factor of C, then $C \models C'$. We leave this to the reader to prove[27] From now on, we will assume that this elimination procedure has been executed on clauses, and we are only dealing with their "factors".

[27]Exercise.

Resolution

The rule of resolution remains sound for clauses in the predicate logic. That is, if C_1 and C_2 are clauses and R is a resolvent, then $\{C_1, C_2\} \models R$. The presence of variables and substitutions makes the proof of this a little more involved.

Theorem 19 *Suppose R is the result of resolving on literal L in C_1 and M in C_2. Let θ be the most general unifier of L and $\neg M$ that is used to obtain R. Then, the soundness of a single step of resolution means $\{C_1, C_2\} \models (C_1 - \{L\})\theta \cup (C_2 - \{M\})\theta$.*

Proof: Let M be a model for C_1 and C_2. Now, we know that either (a) $L\theta$ is true and $M\theta$ is false in M; or (b) $L\theta$ is false and $M\theta$ is true in M. Suppose the former. Since M is a model for C_2, it is a model for $C_2\theta$ (based on theorem 18). Therefore, at least one other literal $(C_2 - \{M\})\theta$ must be true in M. In other words, M is a model for $(C_1 - \{L\})\theta \cup (C_2 - \{M\})\theta$. Case (b) similarly results in M being a model for $(C_1 - \{L\})\theta$ and hence for R. So, a single resolution step is sound - the soundness of a proof consisting of several resolutions steps can be shown quite easily using the technique of induction. $\Box$

Recall the second property of resolution from propositional logic, namely that of refutation-completeness. In other words, if a formula (or a set of formulæ) is inconsistent, then the empty clause $\Box$ is derivable by the use of resolution. This property continues to hold for resolution in first-order logic. But before we look at that, we revisit an important result.

1.4.6 Subsumption Revisited

Recall that in propositional logic, a clause C subsumed a clause D if $C \subseteq D$. In first-order logic, this generalises as follows. A clause C *subsumes* a clause D if there is some substitution θ such that $C\theta \subseteq D$. What does this mean? It means that after applying the substitution θ to C, every literal in C appears in D. Here are a pair of clauses C and D such that C subsumes D:

$$C : Primate(x) \leftarrow Ape(x)$$

$$D : Primate(Henry) \leftarrow Ape(Henry), Human(Henry)$$

Here, a substitution of $\theta = \{x/Henry\}$ applied to C makes $C\theta \subseteq D$. In general:

Theorem 20 *If C and D are clauses such that $C\theta \subseteq D$ for some substitution θ, then $C \models D$.*

Proof: Since C is a universally quantified formula, by theorem 18, we must have $C \models C\theta$. Also, since clauses are disjunctions of literals and $C\theta \subseteq D$, clearly, $C\theta \models D$ and the result follows. $\Box$

However, unlike propositional logic, the reverse does not hold. That is, $C \models D$ does not necessarily mean that C subsumes D. Here is an example of this:

$$C : Human(x) \leftarrow Human(father(x))$$

$$D : Human(y) \leftarrow Human(father(father(y)))$$

With a little thought (let us not get too entangled in the species problem here), you should be able to convince yourself that $C \models D$. But you will find it impossible to find a substitution θ that will make $C\theta \subseteq D$. What makes the difference to the propositional case? The difference between implication and subsumption in first-order logic arises because of self-recursive clauses of the kind shown: a short, but influential paper by Georg Gottlob shows that it is indeed only the self-recursive case that results in the difference.

1.4.7 Subsumption Lattice over Atoms

The subsumption relation is an example of a quasi-order. Let us take the simple case of definite clauses with a single literal (that is, atoms). Consider the set $\mathcal{A}$ of all atoms in some language, and $\mathcal{A}^+ = \mathcal{A} \cup \{\top, \bot\}$. Let the binary relation $\succeq$ be such that:

- $\top \succeq \mathbf{l}$ for all $\mathbf{l} \in \mathcal{A}^+$
- $\mathbf{l} \succeq \bot$ for all $\mathbf{l} \in \mathcal{A}^+$
- $\mathbf{l} \succeq m$ iff there is a substitution θ such that $\mathbf{l}\theta = m$, for $\mathbf{l}, \mathbf{m} \in \mathcal{A}$

We will represent a list of elements $e_1, \ldots, e_n$ as the(as the language Prolog does) by $[e_1, \ldots, e_n]$, and let $\mathbf{l} = Mem(x, [x, y])$ and $\mathbf{m} = Mem(1, [1, 2])$ then $\mathbf{l} \succeq \mathbf{m}$ with $\theta = \{x/1, y/2\}$. It is easy to see that $\succeq$ is a quasi-order over $\mathcal{A}^+$: clearly $\mathbf{l} \succeq \mathbf{l}$, with the empty substitution $\theta = \emptyset$ (that is, $\succeq$ is reflexive). Now, let $\mathbf{l} \succeq \mathbf{m}$ and $\mathbf{m} \succeq \mathbf{l}$. That is, there are some substitutions θ_1 and θ_2 such that $\mathbf{l}\theta_1 = \mathbf{m}$ and $\mathbf{m}\theta_2 = \mathbf{l}$. That is, $(\mathbf{l}\theta_1) \circ \theta_2 = n$. With $\theta = \theta_1 \circ \theta_2$ it follows that $\mathbf{l} \succeq \mathbf{l}$.

Since $\succeq$ is a quasi-order, we know a partial ordering must result from the partition of $\mathcal{A}^+$ into a set of equivalence classes $\mathcal{A}^+_E$. In fact, the partitions are $\{[\top]\}, \{[\bot]\}, X_1, \ldots$ where $[\mathbf{l}]$ denotes all atoms that are alphabetic variants[28] of $\mathbf{l}$. That is, if $\mathbf{l}, \mathbf{m} \in X_i$ then there are substitutions μ and σ s.t. $\mathbf{l}\mu = \mathbf{m}$ and $\mathbf{m}\sigma = \mathbf{l}$. That is, $\succeq$ is a partial ordering over the set of equivalence classes of atoms $(\mathcal{A}^+_E)$. ($Mem(x_1, [x_1, y_1]), Mem(x_2, [x_2, y_2]) \ldots$ are examples of members of an equivalence class.)

Recall that the difference between subsumption and implication in first-order logic arose with the appearance of self-recursive clauses. Since there is no possibility of this with atoms in first-order logic, subsumption and implication are equivalent, and we can see that logical implication (*models*) over atoms is also a quasi-order over atoms.

[28]Two atoms are subsume-equivalent iff they are variants. This is not true for clauses in general.

As soon as we have a quasi-order, we can effectively construct a partial-order over equivalence classes. So, the quasi-order of subsumption over atoms results in a partial order over equivalence classes of atoms. In fact, $\mathcal{A}_E^+$ is a lattice with the binary operations $\sqcap$ and $\sqcup$ defined on elements of $\mathcal{A}_E^+$ as follows (here, we have used $[\cdot]$ to represent an equivalence class):

- $[\bot] \sqcap [\mathbf{l}] = [\bot]$, and $[\top] \sqcap [\mathbf{l}] = [\mathbf{l}]$
- If $\mathbf{l}_1, \mathbf{l}_2 \in \mathcal{A}$ have a most general unifier (see page 78) θ then $[\mathbf{l}_1] \sqcap [\mathbf{l}_2] = [\mathbf{l}_1\theta] = [\mathbf{l}_2\theta]$.

  This can be proved as follows. Let $[\mathbf{u}] \in \mathcal{A}_E^+$ such that $[\mathbf{l}_1] \succeq [u]$ and $[\mathbf{l}_2] \succeq [\mathbf{u}]$, then we need to show that $[\mathbf{l}_1\theta] \succeq [\mathbf{u}]$. If $[\mathbf{u}] = [\bot]$, this is obvious. If $[\mathbf{u}]$ is conventional, then there are substitutions σ_1 and σ_2 such that $[\mathbf{l}_1\sigma_1] = [\mathbf{u}] = [\mathbf{l}_2\sigma 2]$. Here we can assume σ_1 only acts on variables in $\mathbf{l}_1$, and σ_2 only acts on variables in $\mathbf{l}_2$. Let $\sigma = \sigma_1 \cup \sigma_2$. Notice that σ is a unifier for $\{[\mathbf{l}_1], [\mathbf{l}_2]\}$. Since θ is an mgu for $\{[\mathbf{l}_1\sigma_1], [\mathbf{l}_2\sigma_2]\}$, there is a γ such that $\theta\gamma = \sigma$. Now $[\mathbf{l}_1\theta\gamma] = [\mathbf{l}_1\sigma] = [\mathbf{l}_1\sigma_1] = [\mathbf{u}]$, so $[\mathbf{l}_1\theta] \succeq [\mathbf{u}]$.
- If $\mathbf{l}_1, \mathbf{l}_2 \in \mathcal{A}$ do not have a most general unifier θ then $[\mathbf{l}_1] \sqcap [\mathbf{l}_2] = [\bot]$.

  Since $\mathbf{l}_1$ and $\mathbf{l}_2$ are not unifiable, there is no conventional atom $\mathbf{u}$ such that $[\mathbf{l}_1] \succeq [\mathbf{u}]$ and $[\mathbf{l}_2] \succeq [\mathbf{u}]$. Hence $[\mathbf{l}_1] \sqcap [\mathbf{l}_2] = [\bot]$.
- $[\bot] \sqcup [\mathbf{l}] = [\mathbf{l}]$, and $[\top] \sqcup [\mathbf{l}] = [\top]$
- If $\mathbf{l}_1$ and $\mathbf{l}_2$ have an "anti-unifier" $\mathbf{m}$ then $[\mathbf{l}_1] \sqcup [\mathbf{l}_2] = [\mathbf{m}]$; otherwise $[\mathbf{l}_1] \sqcup [\mathbf{l}_2] = [\top]$

Henceforth, we will drop the square backets $[\cdot]$ to denote equivalence classes and will instead implicitly assume their presence. The "anti-unifier" in the join operation is not something we have come across before, and needs some explanation. To get started, let us look at the atom $Mem(1, [1, 2])$. The list $[1, 2]$ written out in long-hand is really a term composed of the constants 1, 2 and the empty list, which we will denote by the constant *nil*. That is, $[1, 2]$ is really the term $list(1, list(2, list(nil)))$, where *list* is a function, and $Mem(1, [1, 2])$ is really $Mem(1, list(1, list(2, nil)))$. Now, we can devise a "term-place" notation to identify the occurrence of each term in any atom. In $Mem(1, list(1, list(2, nil)))$, the 1 is a term that occurs in two places: in the first argument (or "place") of Mem, and as the first argument of the second place of Mem. We can denote these two occurrences as $(1, \langle 1\rangle)$ and $(1, \langle 2, 1\rangle)$. Similarly, we can encode the occurrences of other terms: $(2, \langle 2, 2, 1\rangle)$ and $(nil, \langle 2, 2, 2\rangle)$.

You should convince yourself that the occurrence of every term t in an atom can indeed be represented by the pair (t, p), where p is a sequence of places. We now have all we need to be able to describe the anti-unification algorithm for a pair of literals with the same predicate symbol (adapted from Plotkin, 1970):

Input: A pair of atoms $\mathbf{l}_1$ and $\mathbf{l}_2$ with the same predicate symbol

Output: $\mathbf{l}_1 \sqcup \mathbf{l}_2$

1. Let $\mathbf{l} = \mathbf{l}_1$ and $\mathbf{m} = \mathbf{l}_2$, $\theta = \emptyset$, $\sigma = \emptyset$
2. If $\mathbf{l} = \mathbf{m}$ return $\mathbf{l}$ and stop.
3. Try to find terms t_1 and t_2 that have the same (leftmost) place in $\mathbf{l}$ and $\mathbf{m}$ respectively, such that $t_1 \neq t_2$ and either t_1 and t_2 begin with different function symbols, or at least one of them is a variable.
4. If there is no such t_1, t_2, return $\mathbf{l}$ and stop.
5. Choose a variable x that does not occur in either $\mathbf{l}$ or $\mathbf{m}$ and wherever t_1 and t_2 occur in the same place in $\mathbf{l}$ and $\mathbf{m}$, replace each of them by x
6. Set θ to $\theta \cup \{x/t_1\}$ and σ to $\sigma \cup \{x/t_2\}$
7. Go to Step 3

The Table 1.1 shows the progressive construction of lubs starting with terms and culminating in literals.

Lub	Definition	Examples
lub of terms $lub(t_1, t_2)$	1. $lub(t, t) = t$, 2. $lub(f(s1, \ldots, s_n), f(t_1, \ldots, t_n)) = f(lub(s_1, t_1), \ldots, lub(s_n, t_n))$, 3. $lub(f(s_1, \ldots, s_m), g(t_1, \ldots, t_n)) = V$, where $f \neq g$, and V is a variable which represents $lub(f(s_1, \ldots, s_m), g(t_1, \ldots, t_n))$, 4. $lub(s, t) = V$, where $s \neq t$ and at least one of s and t is a variable; in this case, V is a variable which represents $lub(s, t)$.	• $lub([a, b, c], [a, c, d]) = [a, X, Y]$. • $lub(f(a, a), f(b, b)) = f(lub(a, b), lub(a, b)) = f(V, V)$ where V stands for $lub(a, b)$. • When computing lggs one must be careful to use the same variable for multiple occurrences of the lubs of subterms, *i.e.*, $lub(a, b)$ in this example. This holds for lubs of terms, atoms and clauses alike.
lub of atoms $lub(\mathbf{a}_1, \mathbf{a}_2)$	1. $lub(P(s_1, \ldots, s_n), P(t_1, \ldots, t_n)) = P(lub(s_1, t_1), \ldots, lub(s_n, t_n))$, if atoms have the same predicate symbol P, 2. $lub(P(s_1, \ldots, s_m), Q(t_1, \ldots, t_n))$ is undefined if $P \neq Q$.	
lub of literals $lub(\mathbf{l}_1, \mathbf{l}_2)$	1. if $\mathbf{l}_1$ and $\mathbf{l}_2$ are atoms, then $lub(\mathbf{l}_1, \mathbf{l}_2)$ is computed as defined above, 2. if both $\mathbf{l}_1$ and $\mathbf{l}_2$ are negative literals, $\mathbf{l}_1 = \overline{\mathbf{a}_1}$, $\mathbf{l}_2 = \overline{\mathbf{a}_2}$, then $lub(\mathbf{l}_1, \mathbf{l}_2) = lub(\overline{\mathbf{a}_1}, \overline{\mathbf{a}_2}) = \overline{lub(\mathbf{a}_1, \mathbf{a}_2)}$, 3. if $\mathbf{l}_1$ is a positive and $\mathbf{l}_2$ is a negative literal, or vice versa, $lub(\mathbf{l}_1, \mathbf{l}_2)$ is undefined.	• $lub(Parent(ann, mary), Parent(ann, tom)) = Parent(ann, X)$. • $lub(Parent(ann, mary), \overline{Parent(ann, tom)}) = undefined$. • $lub(Parent(ann, X), Daughter(mary, ann)) = undefined$.

Table 1.1: Table showing progressive definitions of lubs, starting with terms and culminating in literalss.

Let us look at an example of constructing the anti-unifier of $Mem(1, [1, 2])$ and $Mem(2, [2, 4]$. That is, $\mathbf{l}_1 = Mem(1, list(1, list(1, list(2, nil))))$ and $\mathbf{l}_2 = Mem(2, list(2, list(2, list(4, nil))))$. You should be able to work through the

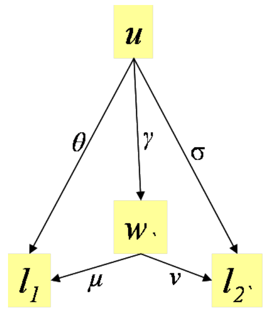


Figure 1.13: Illustration of the proof of theorem 22.

steps of the algorithm to find that it terminates with $\mathbf{l} = \mathbf{m} = Mem(x, list(x, list(y, nil)))$ with $\theta = \{x/1, y/2\}$ and $\sigma = \{x/2, y/4\}$.

But is the procedure correct? That is, does it really return a lub of a pair of atoms $\mathbf{l}_1$ and $\mathbf{l}_2$? Suppose the procedure returned an atom $\mathbf{l}$, and let $\theta = \{x_1/s_1, \ldots, x_k/s_k\}$ and $\sigma = \{x_1/t_1, \ldots, x_k/t_k\}$. That is $\mathbf{l}\theta = \mathbf{l}_1$ and $\mathbf{l}\sigma = \mathbf{l}_2$. Now suppose there is some other atom $\mathbf{l}'$ such that $\mathbf{l}' \succeq \mathbf{l}_1$ and $\mathbf{l}' \succeq \mathbf{l}_2$. Then, we have to show that $\mathbf{l}' \succeq \mathbf{l}$ for any such $\mathbf{l}'$.

The proof of this is a bit laborious and will dealt with subsequently. The truth of the next lemma is easy to see:

Theorem 21 *After each iteration of the Anti-Unification Algorithm, there are terms $s_1, \ldots, s_i$ and $t_1, \ldots, t_i$ such that:*

1. $\theta = \{z_1/s_1, \ldots, z_i/s_i\}$ *and* $\sigma = \{z_1/t_1, \ldots, z_i/t_i\}$.
2. $\mathbf{l}\theta = \mathbf{l}_1$ *and* $\mathbf{m}\sigma = \mathbf{l}_2$.
3. *For every* $1 \leq j \leq i$, s_j *and* t_j *differ in their first symbol.*
4. *There are no* $1 \leq j, k \leq i$ *such that* $j \neq k$, $s_j = s_k$ *and* $t_j = t_k$.

Theorem 22 *Let $\mathbf{l}_1$ and $\mathbf{l}_2$ be two atoms with the same predicate symbol. Then the Anti- Unification Algorithm with $\mathbf{l}_1$ and $\mathbf{l}_2$ as inputs returns $\mathbf{l}_1 \sqcup \mathbf{l}_2$.*

Proof: It is easy to see that the algorithm terminates after a finite number of steps, for any $\mathbf{l}_1$, $\mathbf{l}_2$. Let $\mathbf{u}$ be the atom that the algorithm returns, and let $\theta = \{z_1/s_1, \ldots, z_i/s_i\}$ and $\sigma = \{z_1/t_1, \ldots, z_i/t_i\}$ be the final values of θ and σ in the computation of $\mathbf{u}$ (so $\mathbf{u}$ equals the final values of $\mathbf{l}$ and $\mathbf{m}$ in the execution of the algorithm). Then $\mathbf{u}\theta = \mathbf{l}_1$ and $\mathbf{u}\sigma = \mathbf{l}_2$ by Theorem 21, part 2. Suppose $\mathbf{v}$ is an atom such that $\mathbf{v} \succeq \mathbf{l}_1$ and $\mathbf{v} \succeq \mathbf{l}_2$. In order to show that $\mathbf{u} = \mathbf{l}_1 \sqcup \mathbf{l}_2$, we have to prove $\mathbf{v} \succeq \mathbf{u}$.

Let $\mathbf{w} = \mathbf{u} \sqcap \mathbf{v}$, which exists by the proof on page 82. Then $\mathbf{u} \succeq \mathbf{w}$ and $\mathbf{v} \succeq \mathbf{w}$. Since $\mathbf{w} = \mathbf{u} \sqcap \mathbf{v}$, $\mathbf{u} \succeq \mathbf{l}_1$ and $\mathbf{v} \succeq \mathbf{l}_1$, we must have $\mathbf{w} \succeq \mathbf{l}_1$. Similarly

$\mathbf{w} \succeq \mathbf{l}_2$. Thus there are substitutions γ, μ, ν, such that $\mathbf{u}\gamma = \mathbf{w}$, $\mathbf{l}_1 = \mathbf{w}\mu = \mathbf{u}\gamma\mu$ and $\mathbf{l}_2 = \mathbf{w}\nu = \mathbf{u}\gamma\nu$. Then $\mathbf{u}\theta = \mathbf{l}_1 = \mathbf{u}\gamma\mu$ and $\mathbf{u}\sigma = \mathbf{l}_2 = \mathbf{u}\gamma\nu$ (see Figure 1.13 for illustration). Hence, if x is a variable occurring in $\mathbf{u}$, then $x\theta = x\gamma\mu$ and $x\sigma = x\gamma\nu$.

We will now show that $\mathbf{u}$ and $\mathbf{w} = \mathbf{u}\gamma$ are variants, by showing that γ is a renaming substitution for $\mathbf{u}$. Suppose it is not. Then γ maps some variable x in $\mathbf{u}$ to a term that is not a variable, or γ unifies two distinct variables x, y in $\mathbf{u}$.

Suppose x is a variable in $\mathbf{u}$, such that $x\gamma = t$, where t is a term that is not a variable. If x is not one of the z_j's, then $x\gamma\mu = x\theta = x$, contradicting the assumption that $x\gamma = t$ is not a variable. But on the other hand, if x equals some z_j, then $t\mu = x\gamma\mu = x\theta = s_j$ and $t\nu = x\gamma\nu = x\sigma = t_j$. Then s_j and t_j would both start with the first symbol of t, contradicting theorem 21, part 3. So this case leads to a contradiction.

Suppose x and y are distinct variables in $\mathbf{u}$ such that γ unifies x and y. Then,

1. If neither x nor y is one of the z_j's, then $x\gamma\mu = x\theta = x \neq y = y\theta = y\gamma\mu$, contradicting $x\gamma = y\gamma$
2. If x equals some z_j and y does not, then $x\gamma\mu = x\theta = s_j$ and $x\gamma\nu = x\sigma = t_j$, so $x\gamma\mu \neq x\gamma\nu$ by theorem 21, part 3. But $y\gamma\mu = y\theta = y = y\sigma = y\gamma\nu$, contradicting $x\gamma = y\gamma$.
3. Similarly for the case where y equals some z_j and x does not.
4. If $x = z_j$ and $y = z_k$, then $j \neq k$, since $x \neq y$. Furthermore, $s_j = x\theta = x\gamma\mu = y\gamma\mu = y\theta = s_k$ and $t_j = x\sigma = x\gamma\nu = y\gamma\nu = y\sigma = t_k$. But this contradicts theorem 21, part 4.

Thus, the assumption that γ unifies two variables in $\mathbf{u}$ also leads to a contradiction. Thus γ is a renaming substitution for $\mathbf{u}$, and hence $\mathbf{u}$ and $\mathbf{w}$ are variants. Finally, since $\mathbf{v} \succeq \mathbf{w}$, we have $\mathbf{v} \succeq \mathbf{u}$. $\Box$

It is however a more straightforward matter to see the following:

Theorem 23 *If $\mathcal{A}_E^+$ is a set of equivalance classes of atoms $\mathcal{A}$ augmented by the two elements $\top$ and $\bot$, then for any pair of elements $\mathbf{l}_1, \mathbf{l}_2 \in \mathcal{A}_E^+$, $\mathbf{l}_1 \sqcup \mathbf{l}_2$ always exists.*

Proof: The possibilities for each of $\mathbf{l}_1$ and $\mathbf{l}_2$ are that they are either: (1) some variant of $\top$; (2) some variant of $\bot$; or (3) an atom from S. It can be verified that $\mathbf{l}_1 \sqcap \mathbf{l}_2$ is defined in all 9 cases.

1. If $\mathbf{l}_1 = \top$ or $\mathbf{l}_2 = \top$, then $\mathbf{l}_1 \sqcup \mathbf{l}_2 = \top$. If $\mathbf{l}_1 = \bot$, then $\mathbf{l}_1 \sqcup \mathbf{l}_2 = \mathbf{l}_2$. If $\mathbf{l}_2 = \bot$, then $\mathbf{l}_1 \sqcup \mathbf{l}_2 = \mathbf{l}_1$.
2. If $\mathbf{l}_1$ and $\mathbf{l}_2$ are conventional atoms with the same predicate symbol, $\mathbf{l}_1 \sqcup \mathbf{l}_2$ is given by the Anti-Unification Algorithm.

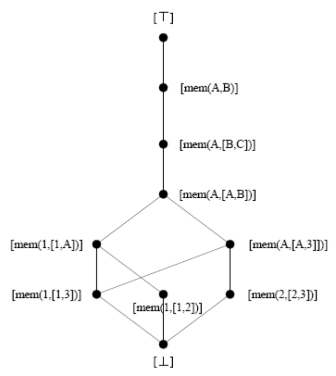


Figure 1.14: An example subsumption lattice over atoms.

3. If $\mathbf{l}_1$ and $\mathbf{l}_2$ are conventional atoms with different predicate symbols, then $\mathbf{l}_1 \sqcup \mathbf{l}_2 = \top$.

□

Now that we have established the existence of an lub and glb of any $\mathbf{l}_1, \mathbf{l}_2 \in \mathcal{A}_E^+$, we have shown that the set of atoms ordered by subsumption, is a lattice.

Theorem 24 *Let $\mathcal{A}$ be the set of atoms. Then $< \mathcal{A}_E^+, \succeq >$ is a lattice.*

Figure 1.14 shows an example subsumption lattice over atoms in $S^+ = \{ \top, \bot, mem(1,[1,3]), mem(1,[1,2]), mem(2,[2,3]), mem(1,[1,A]), mem(A,[A,B]), mem(A,[A,3]), mem(A,[B,C]), mem(A,[B|C])\ mem(A,B) \}$ Note that

- $l = mem(A,[A,B]) \succeq mem(1,[1,2]) = m$ since with $\theta = \{A/1, B/2\}$, $l\theta = m$
- $mem(A1,[A1,B1]), mem(A2,[A2,B2]) \ldots$ are all members of the same equivalence class

Recall that, for atoms $l, m \in S$, subsumption is equivalent to implication. That is, if $l \models m$ then $l \succeq m$. Least-general-generalisation of atoms will be enountered in Lab Nos. 5, 6.

What about subsumption over clauses with more than just one literal? Is this still a quasi-order, with a lattice structure over equivalence classes of clauses? The short answer is "yes", but more on this in Chapter 2.

1.4.8 Covers of Atoms

What about the covers relation in the subsumption lattice of atoms? Recall that covers are the smallest non-trivial steps between individual atoms that we can take in the lattice. Since $\mathbf{l}_2$ is a downward cover of $\mathbf{l}_1$ iff $\mathbf{l}_1$ is an upward cover of $\mathbf{l}_2$, we will first restrict attention to downward covers.

Downward Covers

Theorem 25 *Let $\mathbf{l}_1$ be a conventional atom, f an n-ary function symbol (recall that f can be of zero arity and therefore a constant), z a variable in $\mathbf{l}_1$, and $x_1, \ldots, x_n$, distinct variables not appearing in $\mathbf{l}_1$. Let*

1. *$\theta = \{z/f(x_1, \ldots, x_n)\}$ and*
2. *$\sigma = \{x_i/x_j\}$, $i \neq j$*

Then $\mathbf{l}_2 = \mathbf{l}_1\theta$ and $\mathbf{l}_3 = \mathbf{l}_1\sigma$ are both downward covers of $\mathbf{l}_1$. In fact, every downward cover of $\mathbf{l}_1$ must of one of these two forms (note that a special instance of the first case is when the function f has arity 0 and is therefore a constant). The substitutions θ and σ are termed elementary substitutions. In ILP, these 3 operations define a "downward refinement operator"

Proof:

Proof for (1): It is clear that $\mathbf{l}_1$ and $\mathbf{l}_2$ are not variants, so $\mathbf{l}_1 \succ \mathbf{l}_2$. Suppose there is a $\mathbf{m}$ such that $\mathbf{l}_1 \succ \mathbf{m}\mathbf{l}_2$. Then there are γ, μ, such that $\mathbf{l}_1\gamma = \mathbf{m}$ and $\mathbf{m}\mu = \mathbf{l}_2$, hence $\mathbf{l}_1\gamma\mu = \mathbf{l}_2 = \mathbf{l}_1\theta$. Here γ only acts on variables in $\mathbf{l}_1$, and μ only acts on variables in $\mathbf{m}$.

Let (x, p) be a term occurrence in $\mathbf{l}_1$, where x is a variable. Suppose $x \neq z$, then $x\theta = x$, so (x, p) must also be a term occurrence in $\mathbf{l}_2$. Hence $x\gamma$ must be a variable, for otherwise $(x\gamma\mu, p)$ in $\mathbf{l}_2$ would contain a constant or a function. Thus γ must map all variables other than z to variables. Furthermore, γ cannot unify two distinct variables in $\mathbf{l}_1$, for then $\mathbf{l}_2$ would also have to unify these two variables, which is not the case.

If $z\gamma$ is also a variable, then γ would map all variables to variables, and since γ cannot unify distinct variables, it would map all distinct variables in $\mathbf{l}_1$ to distinct variables. But then γ would be a renaming substitution for $\mathbf{l}_1$, contradicting $\mathbf{l}_1 \succ \mathbf{m}$. Hence γ must map z to some term containing a function symbol.

Now the only way we can have $\mathbf{l}_1\gamma\mu = \mathbf{l}_2$, is if $z\gamma = f(y_1, \ldots, y_n)$ for distinct y_i not appearing in $\mathbf{l}_1$, and no variable in $\mathbf{l}_1$ is mapped to some y_i by γ. But then $\mathbf{l}_1\gamma$ and $\mathbf{l}_2$ would be variants, contradicting $\mathbf{l}_1\gamma = \mathbf{m} \succ \mathbf{l}_2$. Therefore such a $\mathbf{m}$ does not exist, and $\mathbf{l}_2$ is a downward cover of $\mathbf{l}_1$.

Proof for (2): It is clear that $\mathbf{l}_1 \succ \mathbf{l}_3$. Suppose there is a $\mathbf{m}$ such that $\mathbf{l}_1 \succ \mathbf{m} \succ \mathbf{l}_3$. Then there are γ, μ, such that $\mathbf{l}_1\gamma = \mathbf{m}$ and $\mathbf{m}\mu = \mathbf{l}_3$, hence $\mathbf{l}_1\gamma\mu = \mathbf{l}_3 = \mathbf{l}_1\sigma$. Here γ only acts on variables in $\mathbf{l}_1$, and μ only on variables in $\mathbf{m}$. Note that γ and μ can only map variables to variables, since otherwise $\mathbf{l}_1\gamma\mu = \mathbf{l}_3$ would contain more occurrences of functions or constants than $\mathbf{l}_1$, contradicting $\mathbf{l}_1\sigma = \mathbf{l}_3$, since σ does not add any occurrences of function symbols to $\mathbf{l}_1$.

If γ does not unify any variables in $\mathbf{l}_1$, then $\mathbf{l}_1$ and $\mathbf{m}$ would be variants, contradicting $\mathbf{l}_1 \succ \mathbf{m}$. If γ unifies any other variables than z and x, then we could not have $\mathbf{l}_1\gamma\mu = \mathbf{l}_3$. Hence γ must unify z and x, and cannot unify any other variables. But then $\mathbf{l}_1\gamma$ and $\mathbf{l}_3$ would be variants, contradicting $\mathbf{l}_1\gamma = \mathbf{m} \succ \mathbf{l}_3$. Therefore such a $\mathbf{m}$ does not exist, and $\mathbf{l}_3$ is a downward cover of $\mathbf{l}_1$ $\Box$

Application of the elementary substitutions on $\top$ results in its downward covers called *most general atoms*, that consist of all n-ary predicate symbols, each with n-distinct variables as arguments. Dually, $\mathbf{l}_2$ is an upward cover of $\mathbf{l}_1$ iff $\mathbf{l}_1$ is a downward cover of $\mathbf{l}_2$. Thus the upward covers of some conventional atom $\mathbf{l}_1$ are also of two types, which can be constructed by inverting the two elementary substitutions discussed in theorem 25. Trivially, every ground atom[29] is an upward cover of $\perp$.

Further, it can be proved that given two atoms $\mathbf{l}_1$ and $\mathbf{l}_2$ such that $\mathbf{l}_1 \succ \mathbf{l}_2$ ($\mathbf{l}_2 \succ \mathbf{l}_1$), there is a finite sequence of downward (upward) covers from $\mathbf{l}_1$ ($\mathbf{l}_2$) to a variant of $\mathbf{l}_2$ ($\mathbf{l}_1$). This means that if we want to get from $\mathbf{l}_1$ to (a variant of) $\mathbf{l}_2$, we only need to consider downward (upward) covers of $\mathbf{l}_1$, downward (upward) covers of downward (upward) covers of $\mathbf{l}_1$, *etc.* In fact, there is a *finite downward cover chain algorithm* for this purpose, which is outlined in Figure 1.15.

```
INPUT: Conventional atoms $\mathbf{l}, \mathbf{m}$, such that $\mathbf{l} \succ \mathbf{m}$.
OUTPUT: A finite chain $\mathbf{l} = \mathbf{l}_0 \succ \mathbf{l}_1 \succ \ldots \succ \mathbf{l}_{n-1} \succ \mathbf{l}_n = \mathbf{m}$, where each $\mathbf{l}_{i+1}$ is a downward cover of $\mathbf{l}_i$.
Set $\mathbf{l}_0 = \mathbf{l}$ and $i = 0$, let $\theta_0$ be such that $\mathbf{l}\theta_0 = \mathbf{m}$; (1)
if No term in $\theta_i$ contains a function or a constant; (2) then
    Goto 3.
else if $x/f(t_1, \ldots, t_n)$ is a binding in $\theta_i$ ($n \geq 0$) then
    Choose new distinct variables $z_l, \ldots, z_n$;
    Set $\mathbf{l}_{i+1} = \mathbf{l}_i\{x/f(z_1, \ldots, z_n)\}$;
    Set $\theta_{i+1} = (\theta_i \setminus \{x/f(t_1, \ldots, t_n)\}) \cup \{z_1/t_1, \ldots, z_n/t_n\}$;
    Set $i$ to $i + 1$ and goto 2;
end if
if There are distinct variables $x, y$ in $\mathbf{l}_i$, such that $x\theta_i = y\theta_i$ (3) then
    Set $\mathbf{l}_{i+1} = \mathbf{l}_i\{x/y\}$;
    Set $\theta_{i+1} = \theta_i \setminus \{x/x\theta_i\}$;
    Set $i$ to $i + 1$ and goto 3;
else if Such x, y do not exist then
    Set $n = i$ and stop;
end if
```

Figure 1.15: Finite Downward Cover Chain Algorithm.

The subsumption ordering on atoms can be summarized through the following example:

- $l = mem(A, [A, B]) \succeq mem(1, [1, 2]) = m$ since with $\theta = \{A/1, B/2\}$, $l\theta = m$
- $mem(A1, [A1, B1]), mem(A2, [A2, B2]) \ldots$ are all members of the same equivalence class

[29]Number of ground atoms can be infinite if the language consists of a function symbol of arity ≥ 1.

Upward Covers

To construct a finite chain of upward covers to $\mathbf{l}_1$, starting from $\mathbf{l}_2$, where $\mathbf{l}_1 \succ \mathbf{l}_2$, the algorithm 1.15 needs to be reversed. While algorithm 1.15 first instantiates variables to functions and constants, and then unifies some variables, the reverse algorithm "undoes" unifications and instantiations using *inverse substitution* (which is, strictly speaking, not a function).

One asymmetry of the downward and upward cases concerns the upward covers of $\perp$. In case of a language without constants but with at least one function symbol of arity ≥ 1, the bottom element $\perp$ has no upward covers at all, let alone a finite complete set of upward covers. In case of a language with at least one constant and at least one function symbol of arity ≥ 1, there are an infinite number of conventional ground atoms, each of which is an upward cover of $\perp$. Together these ground atoms comprise a complete set of upward covers of $\perp$, but again $\perp$ has no finite complete set of upward covers in this case. However, each conventional atom does have a finite complete set of upward covers. The top element $\top$ does not have any upward covers at all, but it has the empty set as a finite complete set of upward covers, since no element lies "above" $\top$.

1.4.9 The Subsumption Theorem Again

The Subsumption Theorem holds for first-order logic, just as it did for propositional logic:

> If Σ is a set of first-order clauses and D is a first-order clause. Then $\Sigma \models D$ if and only if D is a tautology or there is a clause C such that there is a derivation of C from Σ using resolution ($\Sigma \vdash_R C$) and C subsumes D.

By "derivation of a clause C" here, we mean the same as in propositional logic (page 30), that is, there is a sequence of clauses $R_1, \ldots, R_k = C$ such that each R_i is either in Σ or is a resolvent of a pair of clauses in $\{R_1, \ldots, R_{i-1}\}$. While extending the proof of theorem 9, the proof of this is a bit involved and we do not present it here: we refer you to [NCdW97] for a complete proof that shows that the result does indeed hold.

An immediate consequence is that the refutation-completeness of resolution follows for first-order logic as well. It is not possible, therefore, to decide, using resolution, whether a set Σ of Horn clauses is, in fact, unsatisfiable (that is, $\Sigma \models \Box$: in fact, we will see later that the roots of this undecidability is more fundamental than just to do with Horn clauses or resolution).[30] All that we are saying with refutation-completeness is that $\underline{\text{if}}$ $\Sigma \models \Box$ then there is a resolution

[30]This gives us another difference between implication and subsumption. Unlike implication, subsumption between a pair of clauses *is* decidable, although not necessarily efficiently in all cases. We can informally show that it is decidable whether a clause C subsumes a clause D. If $C \succeq D$, then there is a substitution θ which maps each $\mathbf{l}_i \in C$ to some $\mathbf{l}_j \in D$. If C contains n literals, and D contains m literals, then there are m^n ways in which the literals in C can be paired up with literals in D. Then we can decide $C \succeq D$ by checking whether for at least one of those m^n ways of pairing the n literals in C to some of the m literals in D, there is a

proof that ends in $\Box$. We cannot, however use an algorithm with a resolution-based theorem prover to determine if some set Σ is, in fact, unsatisfiable. What can go wrong? Well, if a $\Box$ is found, then the algorithm can terminate with "success". But, it may end up with cases where it will be impossible to tell if the resolution process will terminate. We will see an example of this shortly.

1.4.10 Proof Strategies Once Again

Recall what we have been using so far: the derivation of a clause C from a set of clauses Σ means there is a sequence of clauses $R_1, \ldots, R_k = C$ such that each C_i is either in Σ or is a resolvent of a pair of clauses in $\{R_1, \ldots, R_{i-1}\}$. This remains unchanged for first-order logic, and results in the unconstrained form of a proof for C. So, just as in the propositional case, we will say that there is a *linear* derivation for C from Σ if there is a sequence $R_0, \ldots, R_k = C$ such that $R_0 \in \Sigma$ and each R_i $(1 \leq i \leq k)$ is a resolvent of R_{i-1} and a clause $C_i \in \Sigma \cup \{R_0, \ldots, R_{i-2}\}$. By requiring side clauses to be only from Σ, we obtain (as before) the strategy called input resolution. Also, as in propositional logic, linear resolution is refutation-complete but input (and SLD) resolution for arbitrary clauses is not, except when restricted to Horn clauses.

To recap refutation, consider $P \models q$. Based on the deduction theorem:

$$P \models q \;\equiv\; P \models (q \leftarrow) \text{ iff:}$$

$$P \models (FALSE \leftarrow\sim q) \text{ iff:}$$

$$P \cup \{\sim q\} \models FALSE$$

That is $P \models q$ iff $P \cup \{\sim q\}$ is unsatisfiable. Thus, logical consequence can be checked by refutation.

We will illustrate general resolution in first order logic with an example. First of all, resolving a pair of clauses requires a substitution that unifies a pair of complementary literals.

Let $D = \{logical(A), \neg android(A)\}$ and $\theta = \{A/Y\}$

$D\theta =$

The resolvent of C, D is $E = \{likes(X,Y), \neg vulcan(X), \neg android(Y)\}$

$\mathbf{E} = (C - \{l\})\theta \cup (D - \{m\})\theta = (C\theta - \{l\}\theta) \cup (D\theta - \{m\}\theta)$ where $l\theta = \neg m\theta$

The typical resolution steps are:

Step 0. Given a pair of clauses:

$$C_1 : \; likes(steve, X) \;\leftarrow\; buys(X, ilp\_book)$$
$$C_2 : \; buys(X, ilp\_book) \;\leftarrow\; sensible(X), rich(X)$$

Step 1. Rename all variables apart.

θ such that $\mathbf{l}_i\theta = \mathbf{m}_j$, for each $(\mathbf{l}_i, \mathbf{m}_j)$ in the pairing. If so, there is a θ such that $C\theta \subseteq D$, and hence $C \succeq D$. If not, then there is no such θ, and $C \not\succeq D$.

$$C_1 : \ likes(steve, A) \ \leftarrow \ buys(A, ilp\_book)$$
$$C_2 : \ buys(B, ilp\_book) \ \leftarrow \ sensible(B), rich(B)$$

Step 2. Identify complementary literals and see if mgu exists.

$$buys(B, ilp\_book)\theta = buys(A, ilp\_book)\theta$$
$$\theta = \{A/B\}$$

Step 3. Apply θ and form resolvent C.

1. Let $C_1\theta = h_1 \vee \sim l_1 \vee \sim l_2 \ldots \vee \sim l_j$
2. Let $C_2\theta = l_1 \vee \sim m_1 \vee \sim m_2 \ldots \vee \sim m_k$
3. Then $C = h_1 \vee \sim m_1 \vee \ldots \vee \sim m_k \vee \sim l_2 \ldots \vee \sim l_j$

Two questions arise with SLD-resolution: how is the negative literal to be selected from the R_{i-1}; and if more than one clause in Σ can resolve with the selected literal, which one should be selected? We illustrate this with an example. Let Σ be the set of clauses:

$$\begin{aligned}
&C_0 : \forall x_1 \forall x_2 \neg Grandparent(x_1, x_2)\\
&C_1 : \forall x \forall y \forall z (Grandparent(x, y) \ \vee\\
&\qquad\qquad \neg Parent(x, z) \ \vee \ \neg Parent(z, y))\\
&C_2 : Parent(henry, jane)\\
&C_3 : Parent(jane, john)
\end{aligned}$$

A little thought should convince you that $\Sigma \models \Box$. We want to see if $\Sigma \vdash_{SLD} \Box$. It is evident that C_0 and C_1 resolve with mgu $\{x_1/x, x_2/y\}$. The resolvent is R_1:

$$\begin{aligned}
&C_0 : \forall x_1 \forall x_2 \neg Grandparent(x_1, x_2)\\
&C_1 : \forall x \forall y \forall z (Grandparent(x, y) \ \vee\\
&\qquad\qquad \neg Parent(x, z) \ \vee \ \neg Parent(z, y))\\
&R_1 : \forall u \forall v \forall w (\neg Parent(u, w) \ \vee \ \neg Parent(w, v))\\
&C_2 : Parent(henry, jane)\\
&C_3 : Parent(jane, john)
\end{aligned}$$

Since we are using SLD, one of the resolvents for the next step has to be R_1. The other resolvent has to be one of the C_i's. Clearly, R_1 can resolve with either C_2 or C_3. Which one should be selected? Having selected one of C_2 or C_3, it is clear that either of the negative literals in R_1 ($\neg Parent(u, w)$ or $\neg Parent(w, v)$) could act as the complementary literal. Which one should be selected? These two issues—which clause and which literal—are decided by a *search rule* and a *computation rule* respectively. A simple search rule is to select clauses in the order they have been shown. A simple computation rule is to select the "leftmost" literal first. In fact, these are the rules used by most Prolog systems. With this search and computation rule, you should be able to derive the empty clause $\Box$. Here is the input resolution diagram for this (Exercise - fill this in):

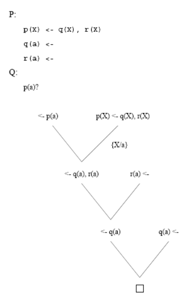


Figure 1.16: An example SLD-tree.

You should be able to draw the diagram for a different choice of search and computation rule: for example a search rule that select clauses in reverse order of appearance, and a "rightmost" literal first computation rule. It is more common, especially in the logic-programming literature, to present instead the search process confronting a SLD-resolution theorem prover in the form of a tree-diagram, called an *SLD-tree*. Such a tree effectively contains all possible derivations that can be obtained using a particular computation rule. Each node in the tree is a "goal" of the form $\leftarrow L_1, L_2, \ldots, L_k$. That is, it is a clause of the form $\forall x_1 \forall x_2 \cdots (\neg L_1 \vee \neg L_2 \vee \cdots \vee \neg L_k)$. Given a set of clauses Σ, the children of a node in the SLD-tree are the result of resolving with clauses in Σ (nodes representing the empty clause $\Box$ have no children). SLD-trees have three kinds of branches: those that end in $\Box$ ("success" branches); those that end in goals (clauses) that cannot be resolved any further ("failure" branches); and those that continue indefinitely (infinite branches).[31] An example SLD tree is produced in Figure 1.16

We can now visualise the effect of the proof strategy, search and computation rules. The SLD proof strategy effectively determines the nodes that can appear in the SLD-tree. Assuming some convention for drawing the tree, the effect of

[31]We do not like these.

changing the computation rules is then to alter the ordering of nodes in an SLD-tree. The search rule represents the procedural aspect of actually conducting the search (for example, in forcing a depth-first search of the tree). We can also clarify how completeness is affected by these choices. First, the refutation-completeness for Horn (and definite) clauses for SLD-resolution can be restated as meaning that if a set of clauses is unsatisfiable then there will be a leaf in the SLD-tree with the empty clause $\Box$. It can be shown that the choice of computation rule will not alter this (informally, you can see that different computation rules will simply move the location of the $\Box$ around). If there is a $\Box$ in the tree, will a specific search rule always find it? The answer is "no": an example is shown below, in which a search rule that does a depth-first search (enumerating clauses in the conventional leftmost first manner), will never recover from the infinite branch. This was illustrated on page 97.

1.4.11 Execution of Logic Programs

Executing definite-clause definitions can sometimes lead to *non-termination* ("infinite loops") or even *unsound* behaviour (recall the idiosyncratic behaviour of *not*/1 in the propositional case). How are logic programs executed?

1. Execution of propositional logic programs
2. Execution of programs without recursion or negation
3. Execution of programs with recursion but no negation
4. Execution of programs with recursion and negation

Searching for answers: Proplog

Consider the following program:

1. $is\_mammal \leftarrow has\_hair$
2. $is\_mammal \leftarrow has\_milk$
3. $is\_carnivore \leftarrow is\_mammal, eats\_meat$
4. $is\_carnivore \leftarrow has\_pointed\_teeth, has\_claws, has\_forward\_eyes$
5. $is\_cheetah \leftarrow is\_carnivore, has\_tawny\_colour, has\_dark\_spots$
6. $is\_tiger \leftarrow is\_carnivore, has\_tawny\_colour, has\_dark\_stripes$
7. $has\_hair \leftarrow$
8. $eats\_meat \leftarrow$
9. $has\_tawny\_colour \leftarrow$
10. $has\_dark\_spots \leftarrow$
11. $is\_dangerous \leftarrow is\_cheetah$
12. $is\_dangerous \leftarrow is\_tiger$

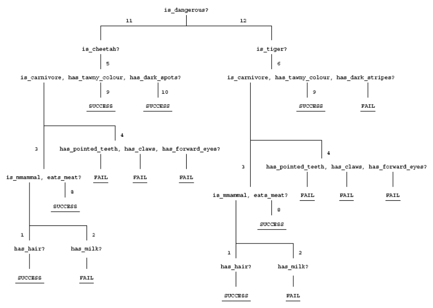


Figure 1.17: The search space for query *is_dangerous*?.

Recall, based on the discussion of SLD resolution starting on page 37 that the search space for the query *is_dangerous*? will be as shown in Figure 1.17

Even with this simple Proplog program, a number of choices have to be when searching this space to see if *is_dangerous* is a logical consequence of clauses $1 - 12$. As discussed ealier, research into *proof procedures* in logic programming has been concerned with searching such spaces *efficiently* keeping in mind the properties of *soundness* or *completeness*:

- Anything that is derived should be a logical consequence
- Any logical consequence should be derivable

Typically, executing a logic program involves solvingqueries of the form: $\mathbf{l}_1, \mathbf{l}_2, \ldots, \mathbf{l}_n$? where the $\mathbf{l}_i$ are literals. Two problems confront us when solving this query:

1. Which literal of the $\mathbf{l}_i$ should be solved first?
    - the rule governing this is called the *computation* rule
2. Which clause should be selected first, when more than one can be used to solve the literal selected?
    - the rule governing this is called the *search* rule

For a given program and query, the *computation* rule determines a tree of choices. The *search* rule determines the order in which this tree is searched (i.e. depth-first, breadth-first *etc.*). Computation proceeds as follows. Given a program P and a query: $\mathbf{l}_1, \mathbf{l}_2, \ldots, \mathbf{l}_{j-1}, \mathbf{l}_j, \ldots \mathbf{l}_n$?

1. Use the computation rule to select $\mathbf{l}_j$
2. Use the search rule to select a clause: $\mathbf{l}_j \leftarrow \mathbf{b}_1, \mathbf{b}_2, \ldots \mathbf{b}_k$ in P that can solve l_j. If none found, *STOP*
3. Solve the query: $\mathbf{l}_1, \mathbf{l}_2, \ldots, \mathbf{l}_{j-1}, \mathbf{b}_1, \mathbf{b}_2, \ldots \mathbf{b}_k \ldots \mathbf{l}_n$?

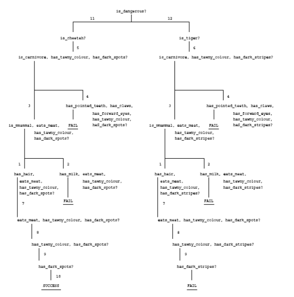

is_dangerous?
11
12
is_cheetah?
5
is_tiger?
6
is_carnivore, has_tawny_colour, has_dark_spots?
is_carnivore, has_tawny_colour, has_dark_stripes?
3
4
has_pointed_teeth, has_claws, has_forward_eyes, has_tawny_colour, has_dark_spots?
FAIL
is_mammal, eats_meat, has_tawny_colour, has_dark_spots?
has_pointed_teeth, has_claws, has_forward_eyes, has_tawny_colour, has_dark_stripes?
is_mammal, eats_meat, has_tawny_colour, has_dark_stripes?
1
2
has_hair, eats_meat, has_tawny_colour, has_dark_spots?
has_milk, eats_meat, has_tawny_colour, has_dark_spots?
has_hair, eats_meat, has_tawny_colour, has_dark_stripes?
has_milk, eats_meat, has_tawny_colour, has_dark_stripes?
7
eats_meat, has_tawny_colour, has_dark_spots?
8
has_tawny_colour, has_dark_spots?
has_tawny_colour, has_dark_stripes?
9
has_dark_spots?
has_dark_stripes?
10
SUCCESS

Figure 1.18: Search space for query $is_dangerous$? using the leftmost literal first computation rule in Figure 1.17.

- As will be seen shortly, the head of the clause selected does not have to match *exactly* the literal selected. It will be enough if the two can *unify*, *i.e.* there is some substitution of variables for terms in the two literals that makes them the same.
- The step of replacing the literal selected with the literals comprising the body of the clause (Step 3) is an application of the rule of inference, *resolution*

In Figure 1.18 is the earlier query with a computation rule that selects the leftmost literal first in the query: The search rule will determine how this tree is searched for a leaf terminating in $SUCCESS$ (for e.g. depth-first left-to-right, depth-first, right-to-left *etc.*)

- Different choices will affect efficiency, and sometimes even the ability to find the $SUCCESS$ leaf
- Trees like this are known as *SLD-trees*: a reference to the trees obtained using a particular computation rule in conjunction with the inference rule of resolution for definite clause programs.

What about proofs for datalog programs without recursion? Consider again the example

1. $gparent(X, Z) \;\leftarrow\; parent(X, Y), parent(Y, Z)$
2. $parent(tom, jo) \;\leftarrow$
3. $parent(jo, bob) \;\leftarrow$
4. $parent(jo, jim) \;\leftarrow$

The rightmost literal first computation rule yields the proof tree in Figure 1.19 What about Datalog programs with recursion? Consider the following program:

1. $less\_than(X, Y) \;\leftarrow\; succ(Y, X)$
2. $less\_than(X, Y) \;\leftarrow\; succ(Z, Y), less\_than(Z, Y)$

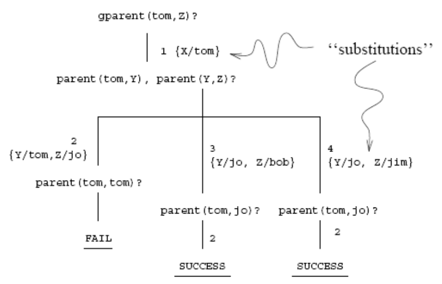


Figure 1.19: Search space for query *gparent*(tom, Z)? using the rightmost literal first computation rule.

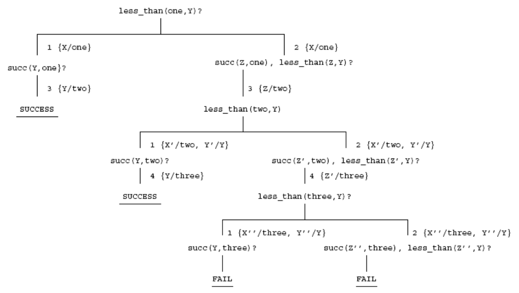


Figure 1.20: Search space for query *less_than(one,Y)?* using the leftmost literal first computation rule.

3. $succ(two, one) \leftarrow$

4. $succ(three, two) \leftarrow$

Leftmost literal rule for the query *less_than(one,Y)* yield the search tree in Figure 1.20

What about completeness with respect to the choice of computation and search rules? One way to search the trees obtained so far is depth-first, left-to-right. Since clauses that appear first (reading top to bottom) in the program have been drawn on the left, this search rule selects clauses in order of appearance in the program. In fact, most logic programs are executed using the following:

Computation rule. Leftmost literal first

Search rule. Depth first search for clauses in order of appearance

However, will a logic-programming system with an arbitrary computation rule, and a depth-first search of clauses in some fixed order always find a leaf terminating in *SUCCESS* (if one exists)? The answer to this is a **No**.

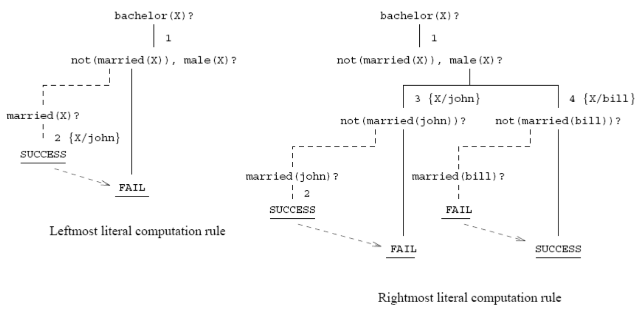


Figure 1.21: Two proof trees with different computation rules for the query *bachelor*(X)?.

Finite failure: programs with negation

Consider the following program:

1. $bachelor(X) \leftarrow not(married(X)), male(X)$
2. $married(john) \leftarrow$
3. $male(john) \leftarrow$
4. $male(bill) \leftarrow$

Two trees for the query *bachelor*(X)? with different computation rules are outlined in Figure 1.21. where, the query *not*(q)? succeeds iff q? "finitely fails". A finitely failed tree is such that it has finite depth, finite depth and all computations end in $FAIL$ As discussed for propositional logic in Section 1.22, the query *not*(q)? succeeds iff q? "finitely fails". A finitely failed tree is a derivation tree of finite depth, finite depth and for which, all computations end in $FAIL$. Given a program P, a computation rule R and a search rule S, one could think of all the ground queries that can be asked of P. For *e.g.* with the program with *less_than*/2 and *succ*/2 clauses, these are of the form: *less_than(one,two)?*, *less_than(two,one)?* ..., *succ(one,two)?* ... Such queries fall into 3 categories:

1. Those which are answered *yes* because a $SUCCESS$ branch is found
2. Those which are answered *no* because the query finitely fails
3. Those that are neither in categories 1 or 2

The three categories of ground clauses are depicted in Figure 1.22.

1.4.12 Answer Substitutions

The example we have just seen is a simple form of the kind of refutation-based theorem proving using resolution adopted by systems based on logic programming. There, we are typically interested in "answer substitutions" for a query, given some set of definite clauses Σ (recall that these are a special kind of Horn

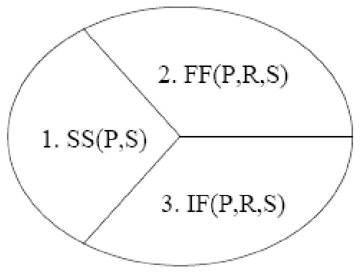


Figure 1.22: Three categories of ground clauses.

clause, in which there is exactly one positive literal). That is, we want to know if there are any substitutions for $x_1, \ldots, x_n$ such that $Q(x_1, \ldots, x_n)$ is a logical consequence of Σ? That is, we are seeking substitutions for $x_1, \ldots, x_n$ such that:

$$\Sigma \models \exists x_1 \exists x_2 \ldots \exists x_n Q(x_1, x_2 \ldots, x_n)$$

holds. Substitutions for the $x_1, \ldots, x_n$ are then said to be *correct* answers.

Using $\exists \mathbf{x} Q(\mathbf{x})$ to denote the formula on the right, we can apply the deduction theorem in manner shown on page 25:

$$\Sigma \models \exists \mathbf{x} Q(\mathbf{x}) \text{ if and only if } \Sigma \cup \{\neg \exists \mathbf{x} Q(\mathbf{x})\} \models \Box$$

Since $\neg \exists x \alpha \equiv \forall x \neg \alpha$ (page 66):

$$\Sigma \models \exists \mathbf{x} Q(\mathbf{x}) \text{ if and only if } \Sigma \cup \{\forall \mathbf{x} \neg Q(\mathbf{x})\} \models \Box$$

Now, Σ is a set definite clauses and $\forall \mathbf{x} \neg Q(\mathbf{x})$ is a Horn clause. Taking as an article of faith what we said earlier about the refutation completeness of SLD-resolution: $\Sigma \cup \{\forall \mathbf{x} \neg Q(\mathbf{x})\} \models \Box$ then $\Sigma \cup \{\forall \mathbf{x} \neg Q(\mathbf{x})\} \vdash_{SLD} \Box$. This provides one way of determining *computed* answers for $x_1, \ldots, x_n$ such that $Q(x_1, \ldots, x_n)$ is a logical consequence of a set of Horn clauses Σ? Answers are obtained by:

1. Adding $\forall \mathbf{x} \neg Q(\mathbf{x})$ to Σ. The result is a set of clauses;
2. Finding all substituitions for $x_1, \ldots, x_n$ that result in a derivation of $\Box$ using SLD-resolution.

Are computed answers always correct? If some care is taken to ensure there are no variables shared between queries and clauses in Σ before commencing an SLD derivation, the answer to this question is "yes". (In fact, if we are only in checking for a proof for $\Box$, then these additional precautions are not needed.) We know, of course, that resolution is refutation-complete. And we also know that the minimal model of Σ, or $MM(\Sigma)$ is identical to the ground atomic logical consequences of Σ. That is, $MM(\Sigma)$ consists of all ground atoms q such that $\Sigma \models q$. The set of ground atoms q for which $\Sigma \cup \{\neg q\} \vdash_{SLD} \square$ is called the *success set* of Σ, or $SS(\Sigma)$. You should now be able to convince yourself that $SS(\Sigma) = MM(\Sigma)$.

The resolution procedure as described here has a limitation concerned with term evaluation

- Consider a function $sqr/1$ that accepts a natural number and returns its square
- The mgu algorithm cannot unify $p(sqr(2))$ and $p(2)$

Extensions are possible to overcome this

- Resolution with "paramodulation" performs term rewrites to achieve this
- But, logic programming systems use a special predicate that forces term evaluation
- Thus, $p(sqr(2))$ is usually written as $X\ is\ 2 * 2, p(X)$. $p(X)$ unifies with $p(4)$ after forced evaluation the value of X by the $is/2$ predicate

Finally, the extension of SLD to negation-as-failure requires some care. SLDNF-resolution for first-order clauses is only sound if we are checking the proof of a ground literal.

1.4.13 Theorem Proving for Full First-Order Logic

1.5 Adequacy

We now turn to the questions of what can, and cannot be achieved by using a representation based on Horn clauses, and inference based on SLD-resolution. First of all, the representation problem. Based on the work of Stephen Kleene, we know that the class of *partial recursive functions* is effectively computable. By the latter, we mean that these functions can be calculated by a Turing machine. By the former, we mean a function (more on functions later) that takes a fixed number of natural numbers as arguments and returns a single natural number as a result (the "partial" part comes from the fact that the function may not be defined for all values of its arguments). In fact, Kleene actually proved that the class of partially recursive functions is identical to the class of functions computable by a Turing machine, which are the only problems solvable by a computer. Now, it has been shown that every partial recursive function can be represented by a definite clause program, and every evaluation of such a function can be encoded as a query in the form we have seen earlier. So, in principle at least, the language of Horn clauses is adequate for representing all computable problems.

But representability does not mean convenient representability. It is for the latter reasons it may be preferable to use different logics, or even other programming languages. We have had a glimpse earlier of how theorem-proving for first-order logic can be accomplished by converting these statements to Horn clauses. So we know that such conversions can be done, and attention turns to the soundness and completeness of the inference mechanisms which we have already considered—specifically, SLD-based resolution—for Horn clause programs. To summarise, ignoring for the moment, the business of search and computation rules:

- Without the "occurs"-check (page 77), SLD resolution can be unsound. In practice logic programming systems like PROLOG omit this check, hoping that the situations where it was needed would not come up.
- The extension of SLD resolution with negation-as-failure can be unsound if attempting to prove atoms that are non-ground.
- SLD-resolution is refutation-complete for Horn clauses.
- SLD-resolution is not affirmation-complete for Horn clauses.
- For general clauses, resolution is refutation-complete if we perform factoring.
- For general clauses, resolution is not affirmation-complete.

When we add on the constraints imposed by the search rule, we lose even refutation-completeness with some kinds of "unfair" search rules (we saw an example of this earlier, when a depth-first search of the SLD-tree went down an infinite path). But, assuming we are using a fair rule, we can confirm the validity any sentence that *is* valid. On the other hand, we cannot deal with problems that are unsolvable, or reliably identify an implication that does not hold. That is, the property of validity is only *semi-decidable*. This is not especially because of the use of Horn clauses or SLD-resolution, but because it is an unavoidable aspect of first-order logic. It is known, for example, from a result due to Schmidt-Schauss in 1988 that implication between a pair of general clauses in first-order logic is undecidable (that is, we will never be able to construct a procedure that will terminate in all cases with the answer). In 1992, it was shown by Marcinkowski and Pacholski that even implication between a pair of Horn clauses is undecidable.

Chapter 2

Exploring a Structured Search Space

ILP is concerned with the automatic construction of "general" logical statements from "specific" ones. For example, given $mem(1, [1, 2]) \leftarrow$ construct $mem(A, [A|B]) \leftarrow$. What do the words "general" and "specific" mean in a logical setting? Can statements of increasing (decreasing) generality be enumerated in an orderly manner? These are questions about the mathematics of "generality" ILP identifies "generality" with $\models$. That is, C_1 is "more general" than C_2 iff $C_1 \models C_2$ The relation $\models$ results in a quasi-ordering over a set of clauses. ILP systems are programs that search such quasi-ordered sets.

The result (theorem 24) that $< \mathcal{A}_E^+, \succeq >$ is a lattice shows that the set of atoms is well structured. The more structured a set is, the better it is suited to be searched for candidates to include in a theory. This search usually procedes by small upward steps (generalization) or downward steps (specialization) in the lattice. In this chapter, we will first dwell upong the subsumption lattice over clauses If we want to generalize or specialize a set of clauses to a single clause, we can use a least generalization or greatest specialization of this set. On the other hand, we may also want to generalize or specialize an individual clause to another individual clause using the idea of covers - the smallest non-trivial steps between individual clauses that we can take in the lattice.

Subsumption is the generality order that is used most often in ILP. It is used much more than logical implication. The reasons for this are mainly practical: subsumption is more tractable and more efficiently implementable than implication. For instance, subsumption between clauses is decidable whereas implication is not (Section 1.4.9).

2.1 Subsumption Lattice over Clauses

Let C and D be clauses. We say $C \succeq D$ iff C subsumes D and $C \succ D$ iff C properly subsumes D. Clearly, the $\succeq$ relation on clauses is reflexive and transitive. Thus it imposes a quasi-order on the set of clauses. Since $\succeq$ is a quasi-order, we know a partial ordering must result from the partition of the set of clauses $\mathcal{C}$ into a set of equivalence classes[1] $\mathcal{C}_E$.

An example of subsumption ordering on clause-sets is produced below:

$$\{mem(A, [A|B]) \leftarrow, mem(A, [B, A|C]) \leftarrow\}$$

$$\succeq$$

$$\{mem(1, [1, 2]) \leftarrow, mem(2, [1, 2]) \leftarrow\}$$

A clause is always subsume-equivalent with a clause that does not contain literals more than once. So for example $P(x) \vee Q(a) \vee P(x)$ is obviously subsume-equivalent with $P(x) \vee Q(a)$. Similarly, the order of literals in a clause does not matter much. For instance, $P(a) \vee P(b) \equiv P(b) \vee P(a)$. This amounts to treating a clause as a set of literals, instead of a disjunction of literals. Thus we may use the set $\{P(a), Q(x)\}$ to represent the clauses $\{Q(x) \vee P(a), P(a) \vee Q(x) \vee P(a), Q(x) \vee P(a) \vee Q(x)\}$, *etc.*

However, the above condition does not give a sufficient condition for subsume-equivalence. While two atoms are subsume-equivalent iff they are variants, this is not true for clauses in general. For instance, $C = \{P(x, x)\} \equiv \{P(x, x), P(x, y)\} = D$, since $C \subseteq D$ and $D\{y/x\} \equiv C$, yet C and D are not variants. In fact, the subsume-equivalence class of this C contains an infinite number of clauses which are not variants. For example, for each n, the clause $D_n = \{P(x, x), P(x, x_1), P(x_1, x_2), \ldots, P(x_{n_1}, x_n)\}$ is subsume-equivalent with $C = \{P(x, x)\}$.

While subsume-equivalent clauses need not be variants, *reduced* subsume-equivalent clauses are however variants. A clause C is said to be *reduced* if there is no proper subset D of C ($D \subset C$) such that $C \equiv D$. Equivalently, a clause C is *reduced* if there is no substitution θ such that $C\theta$ is a proper subset of C. A reduced clause D such that $C \equiv D$ and $D \subseteq C$ is called a *reduction* of C. Thus, $C = \{P(x, y), P(y, x)\}$ is reduced whereas $D = \{P(x, x), P(x, y), P(y, x)\}$ is not reduced, since $D' = \{P(x, x)\} \subset D$ and $D \equiv D'$ is a reduction of D. Note that a subset of a reduced clause need not be reduced itself; if $C = \{\neg Q(x, a), \neg Q(y, a)\}$ and $D = \{P(x, y), \neg Q(x, a), \neg Q(y, a)\}$ then D is reduced, while $C \subset D$ is not reduced, since $C\{x/y\}$ is a proper subset of C. Thus, a reduced clause is a canonical member of the subsume-equivalence class. It can be obtained using the algorithm outlined in Figure 2.1. The algorithm itself is based on the following theorem:

Theorem 26 *Let C be a clause. If for some θ, $C\theta \subseteq C$, then there is a reduced clause $D \subseteq C\theta$ such that $C \equiv D$.*

Proof: Let $C_1 = C\theta$. Since $C \models C\theta$ and $C\theta \models C$, clearly $C \equiv C_1$. If C_1 is reduced, then let $D = C_1$, and we are done. Otherwise, there is a substitution

[1]Note that applying the same substitution θ to two subsume-equivalent clauses may yield two clauses which are no longer subsume-equivalent. For example, if $C = \{P(x, y), P(z, u)\}$ and $D = \{P(x, y)\}$, then $C \equiv D$. Let $\theta = \{y/f(x), z/f(x), u/x\}$. Then $C\theta = \{P(x, f(x)), P(J(x), x)\}$ and $D\theta = \{P(x, f(x))\}$, which are no longer equivalent.

θ_1 such that $C_2 = C_1\theta_1 \subset C_1$. So C_2 is a proper subset of C_1 which is subsume-equivalent to C_1. Since $C_1 \equiv C$, we also have $C_2 \equiv C$, in fact $C\theta\theta_l = C_2 \subset C$. If C_2 is still not reduced, we can go on defining $_3 = C_2\theta_2 \subset C_2$, *etc.* Since C a only contains a finite number of literals, this cannot go on indefinitely. Hence we must arrive at a $D = C_n$ such that D is reduced and $C \equiv D$. □

```
INPUT: A clause $C$.
OUTPUT: A reduction $D$ of $C$.
Set $D = C$, $\theta =$;
repeat
  Set $D$ to $D\theta$;
  Find a literal $\mathbf{l} \in D$ and a substitution $\theta$ such that $D\theta \subseteq D \setminus \{\mathbf{l}\}$;
until Such a $(\mathbf{l}, \theta)$ does not exist;
return $D$.
```

Figure 2.1: Plotkin's reduction algorithm.

2.1.1 Lattice Structure of Clauses

It can be proved that every finite set $\mathcal{S}$ of (general or horn) clauses has a greatest lower bound under subsumption (also called greatest specialization under subsumption or GSS in the ILP literature), as well as a least upper bound under subsumption (also called least generalization under subsumption or LGG in the ILP literature). This holds both for clausal languages $\mathcal{C}$, and for Horn languages $\mathcal{H}$.

Greatest Specialization Under Subsumption

It, is straightforward to show that the GSS of some finite set $\mathcal{S}$ of clauses in $\mathcal{C}$ is simply the union of all clauses in $\mathcal{S}$ after they are standardized apart (EXERCISE). Proving the existence of a GSS of every finite set of Horn clauses in $\mathcal{H}$ requires a little more work. We will assume that $\mathcal{H}$ contains an artificial bottom element[2] $\perp$, such that $C \succeq \perp$ for every $C \in \mathcal{H}$, and $\perp \succeq C$ for every $C \in \mathcal{H}$.

Theorem 27 *Let $\mathcal{H}$ be the Horn language $\mathcal{H}$, with an additional bottom element $\perp \in \mathcal{H}$. Then for every finite non-empty $\mathcal{S} \subseteq \mathcal{H}$, there exists a GSS (glb) of $\mathcal{S}$ in $\mathcal{H}$.*

Proof: Suppose $\mathcal{S} = \{D_1, \ldots, D_n\} \subseteq \mathcal{H}$. Without loss of generality we assume the clauses in S are standardized apart, $D_1, \ldots, D_k$ are the definite program clauses in S, and $D_{k+l}, \ldots, D_n$ are the definite goals in S.

1. If $k = 0$ (*i.e.*, if S only contains goals), then it is easy to show that $D_1 \cup \ldots \cup D_n$ is a GSS of S in $\mathcal{H}$.

[2]Note that $\perp$ is not subsume-equivalent with other tautologies. Two tautologies need not be subsume-equivalent either.

$$\begin{tikzcd}
D_1 \arrow[dr, "\sigma"] \arrow[ddr, "\theta_1"'] & & D_2 \arrow[dl, "\sigma"'] \arrow[ddl, "\theta_2"] \\
& D \arrow[d, "\gamma"] & \\
& C &
\end{tikzcd}$$

Figure 2.2: D is a GSS of D_1 and D_2.

2. If $k \geq 1$ and the set $\{D_1^+, \ldots, D_k^+\}$ (the set of heads of clauses in S), is not unifiable, then $\perp$ is a GSS of S in $\mathcal{H}$.

3. Otherwise, let σ be an mgu for $\{D_1^+, \ldots, D_k^+\}$, and let $D = D_1\sigma \cup \ldots \cup D_n\sigma$ (note that actually $D_i\sigma = D_i$ for $k+1 \leq i \leq n$, since the clauses in S are standardized apart). Since D has exactly one literal in its head, it is a definite program clause. Furthermore, we have $D_i \succeq D$ for every $1 \leq i \leq n$, since $D_i\sigma \subseteq D$.

   To show that D is a GSS of S in $\mathcal{H}$, suppose $C \in \mathcal{H}$ is some clause such that $D_i \succeq C$ for every $1 \leq i \leq n$. For every $1 \leq i \leq n$, let θ_i be such that $D_i\theta_i \subseteq C$, and θ_i only acts on variables in D_i. Let $\theta = \theta_1 \cup \ldots \cup \theta_n$. For every $1 \leq i \leq k$, $D_i^+\theta = D_i^+\theta_i = C^+$, so *theta* is a unifier for $\{D_1^+, \ldots, D_k^+\}$. But σ is an mgu for this set, so there is a γ such that $\theta = \sigma\gamma$. Now $D\gamma = D_1\sigma\gamma \cup \ldots \cup D_n\sigma\gamma = D_1\theta \cup \ldots \cup D_n\theta = D_1\theta_1 \cup \ldots \cup D_n\theta_n \subseteq C$. Hence $D \succeq C$, so D is a GSS of S in $\mathcal{H}$. See Figure 2.2 for illustration of the case where $n = 2$.

   For example, $D = P(a) \leftarrow P(f(a)), Q(y)$ is a GSS of $D_1 : P(x) \leftarrow P(f(x))$ and $D_2 : P(a) \leftarrow Q(y)$. Note that D can be obtained by applying $\sigma = \{x/a\}$ (the mgu for the heads of D_1 and D_2) to $D_1 \cup D_2$, the GSS of D_1 and D_2 in $\mathcal{C}$.

□

Least Generalization Under Subsumption

Proving the existence of the least generalization LGG (lub) is a little harder. Let C and D be clauses. A *selection* of C and D is a pair of compatible literals $(\mathbf{l}, \mathbf{m})$, such that $\mathbf{l} \in C$, $\mathbf{m} \in D$. Two literals are said to be *compatible* if they have the same sign and predicate symbol. For example, $C = \{P(x), P(y), \ldots, P(a)\}$ and $D = \{Q(b), P(a), \ldots, P(b)\}$ have three selections: $(P(x), P(a))$, $(P(y), P(a))$, and $(\neg P(a), \neg P(b))$. Given two clauses C and D, there is only a finite number of selections. Suppose C and D have a total of n selections. Then we can order these in a sequence $S = (\mathbf{l}_1, \mathbf{m}_1), (\mathbf{l}_2, \mathbf{m}_2), \ldots, (\mathbf{l}_n, \mathbf{m}_n)$, and construct two compatible ordered clauses $C_S = \mathbf{l}_1 \vee \ldots \vee \mathbf{l}_n$ and $D_S = \mathbf{m}_1 \vee \ldots \vee \mathbf{m}_n$. Treating $C_S = \vee(\mathbf{l}_1, \ldots, \mathbf{l}_n)$ and $D_S = \vee(\mathbf{m}_1, \ldots, \mathbf{m}_n)$ as atoms, it can be shown that the lub of C_S and D_S constructed using the anti-unification algorithm on page 84 is also an LGS of C and D. As an example, if

- $G = \{\mathbf{l}_1, \mathbf{l}_2, \mathbf{l}_3\}$, for $\mathbf{l}_1 = P(f(a), f(x))$, $\mathbf{l}_2 = P(f(x), g(a))$, $\mathbf{l}_3 = Q(a)$
- $d = \{\mathbf{m}_1, \mathbf{m}_2\}$, for $\mathbf{m}_1 = P(f(b), x)$, $\mathbf{m}_2 = P(y, g(b))$.

then, the set of all selections of C and D can be ordered in the following sequence:

$$S = (\mathbf{l}_1, \mathbf{m}_1), (\mathbf{l}_1, \mathbf{m}_2), (\mathbf{l}_2, \mathbf{m}_1), (\mathbf{l}_2, \mathbf{m}_2)$$

leading to the constructing of the following clauses:

$$C_S = \vee(\mathbf{l}_1, \mathbf{l}_1, \mathbf{l}_2, \mathbf{l}_2)$$
$$D_S = \vee(\mathbf{m}_1, \mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_2)$$

Note that the order and duplication of literals is not ignored in the atomic order. The clauses C_S and D_S are compatible, and have the following lub:

$$E_S = P(f(z_1), z_2) \vee P(z_3, z_4) \vee P(f(z_5), z_6) \vee P(z_7, g(z_1))$$

This lub of C_S and D_S can be shown to be also an LGS of C and D and can be reduced to the LGS $E = \{P(f(z_1), z_2), P(z_7, g(z_1))\}$. Note that the predicate Q does not appear in the lub or LGS.

Theorem 28 *Let $\mathcal{C}$ be a clausal language. Let $C, D \in \mathcal{C}$ be clauses, and S be a sequence of all selections of C and D. Then an $lub(C_S, D_S)$ is an LGS of $\{C, D\}$.*

Proof: First of all if C and D are clauses, and S a sequence of (not necessarily all) selections of C and D. Then $lub(C_S, D_S) \succeq C$ and $lub(C_S, D_S) \succeq D$. This is easy to see. If $E_S = lub(C_S, D_S)$, then $E \succeq C_S$. Also, $C_S \succeq C$, since the literals in C_S form a subset of C. Hence $E \succeq C$, by the transitivity of $\succeq$. Similarly $E \succeq D$.

Let $F = \{\mathbf{l}_1, \ldots, \mathbf{l}_m\}$ be a clause such that $F \succeq C$ and $F \succeq D$. In order to establish that E is an LGS of $\{C, D\}$, we need to prove $F \succeq E$. Since $F \succeq C$ and $F \succeq D$, there are θ_1 and θ_2, and $\mathbf{l}_1, \ldots, \mathbf{l}_m \in C$ and $\mathbf{m}_1, \ldots, \mathbf{m}_m \in D$, such that $\mathbf{l}_i\theta_1 = \mathbf{l}_i$, and $\mathbf{l}_i\theta_2 = \mathbf{m}_i$, for every $1 \leq i \leq m$. Then $S' = (\mathbf{l}_1, \mathbf{m}_1), \ldots, (\mathbf{l}_m, \mathbf{m}_m)$ is a sequence of selections of C and D. Let $C_{S'} = \vee(\mathbf{l}_1, \ldots, \mathbf{l}_m)$, $D_{S'} = \vee(\mathbf{m}_1, \ldots, \mathbf{m}_m)$, let $G = \vee(\mathbf{k}_1, \ldots, \mathbf{k}_m)$ be $lub(C_{S'}, D_{S'})$, and σ_1 and σ_2 be such that $G\sigma_1 = C_{S'}$ and $G\sigma_2 = D_{S'}$. Since $\vee(\mathbf{l}_1, \ldots, \mathbf{l}_m)\theta_1 = C_{S'}$ and $\vee(\mathbf{l}_1, \ldots, \mathbf{l}_m)\theta_2 = CS'$, there must be a γ such that $\vee(\mathbf{l}_1, \ldots, \mathbf{l}_m)\gamma = \vee(\mathbf{k}_1, \ldots, \mathbf{k}_m)$. Thus we have the situation given in Figure 2.3.

$\vee(\mathbf{l}_1, \ldots, \mathbf{l}_m)\gamma = G$, so we have $F \succeq G$. Since every selection in S' also occurs in S, we can prove[3] that $G \succeq E$. Hence $F \succ E$. □

Thus the LGS of any two clauses exists, and can be computed by the method made explicit in the algorithm in Figure 2.4

[3]We leave this an EXERCISE. You will need to iron out duplications from S and S' and permute them so that S' is a prefix of S.

$$\begin{tikzcd}
N_1 \vee \ldots \vee N_m \arrow[d, "\gamma"'] \arrow[dr, "\theta_1 \text{ (resp. } \theta_2)"] & \\
G = K_1 \vee \ldots \vee K_m \arrow[r, "\sigma_1 \text{ (resp. } \sigma_2)"'] & C_{S'} = L_1 \vee \ldots \vee L_m \\
 & (\text{resp. } D_{S'} = M_1 \vee \ldots \vee M_m)
\end{tikzcd}$$

Figure 2.3: Illustration of the proof of theorem 2.3.

```
**INPUT:** Two clauses $C$ and $D$;
**OUTPUT:** An LGS of $\{C, D\}$;
Let $(\mathbf{l}_1, \mathbf{m}_1), \ldots, (\mathbf{l}_n, \mathbf{m}_n)$ be a sequence of all selections of $C$ and $D$;
Obtain $lub(\vee(\mathbf{l}_1, \ldots, \mathbf{l}_n), \vee(\mathbf{m}_1, \ldots, \mathbf{m}_n)) = \vee(\mathbf{l}_1, \ldots, \mathbf{l}_n)$ from the Anti-Unification Algorithm;
**return** $\{\mathbf{l}_1, \ldots, \mathbf{l}_n\}$;
```

Figure 2.4: The LGS Algorithm.

The LGS of any finite set of clauses can be computed by repeatedly applying this algorithm. Notice that if two clauses C and D have no selections for instance, when they have no predicates in common - then their LGS is the empty clause $\Box$. Thus $\Box$ can play the role of top element here, which means that we do not need to add an artificial top element $\top$ to the language $\mathcal{C}$.

Note that if all literals in C and D have the same sign and predicate symbol, then C and D have $|C|.|D|$ selections. Accordingly, the LGS of C and D that can be obtained from these selections may also contain $|C|.|D|$ distinct literals. Thus the number of literals in an LGS may increase quite rapidly.

Since we have now proved the existence of a GSS and LGS of every two clauses, it follows that a clausal language ordered by subsumption has a lattice-structure (we do not need an artificial bottom element $\perp$ for this).

Theorem 29 *Let $\mathcal{C}$ be a clausal language. Then $< \mathcal{C}_E, \succeq >$ is a lattice. Further, if $\mathcal{H}$ be the horn clause language, including an addition symbol $\perp$, then $< \mathcal{H}_E, \succeq >$ is also a lattice.*

Proof: The proof for $(\mathcal{C}_E, \succeq)$ being a lattice follows naturally from the fact that lub (LGS) and glb (GSS) exist for every pair of clauses from $\mathcal{C}$.

As for $\mathcal{H}_E$, since there is at most one selection possible from the heads of a set of Horn clauses $\mathcal{H}$, the LGS of $\mathcal{H}$ has at most one positive literal, and hence is itself also a Horn clause. Therefore $(\mathcal{H}_E, \succeq)$ is a lattice. Here we need the bottom element $\perp$ to guarantee the existence of a GSS of two definite program clauses with different predicate symbols in their head.

Thus, the p.o. set of equivalence classes of atoms $\mathcal{C}_E$ ($\mathcal{H}_E$) is a lattice with the binary operations $\sqcap$ and $\sqcup$ defined on elements of $\mathcal{C}_E$ ($\mathcal{H}_E$) as follows (note that $\perp \in \mathcal{C}$ is the empty set $\emptyset$):

- $[\perp] \sqcap [C] = [\perp]$, and $[\top] \sqcap [C] = [C]$
- If $C_1, C_2 \in \mathcal{C}$ ($C_1, C_2 \in \mathcal{H}$) have a GSS (glb) D then $[C_1] \sqcap [C_2] = [D] = [C_2\theta]$ otherwise $[C_1] \sqcap [C_2] = [\perp]$
- $[\perp] \sqcup [C] = [C]$, and $[\top] \sqcup C] = [\top]$
- If C_1 and C_2 have LGS (lub) M then $[C_1] \sqcup [C_2] = [M]$ otherwise $[C_1] \sqcup [C_2] = [\top]$

For $\mathcal{C}$,

- the GSS is simply the union $C_1 \cup C_1$ (union translates to 'or'ing the two clauses)
- the LGS of C_1 and C_2 is obtained using the LGS algorithm outlined in Figure 2.4

whereas, for $\mathcal{H}$,

- the GSS (or meet operation) is given as follows: If the set of positive literals (heads) in $C_1 \cup C_2$ have an mgu θ, then $GSS(C_1, C_2) = (C_1 \cup C_2)\theta$. Otherwise $GSS(C_1, C_2) = \perp$
- the LGS of C_1 and C_2 is obtained using the LGS algorithm outlined in Figure 2.4.

□

As an example, if

$\mathcal{H} = \{ \square, \perp,$

$is\_tiger(tom) \leftarrow has\_stripes(tom), is\_tawny(tom)$,

$is\_tiger(bob) \leftarrow has\_stripes(bob), is\_white(bob)$,

$is\_tiger(tom) \leftarrow has\_stripes(tom)$,

$is\_tiger(tom) \leftarrow is\_tawny(tom)$,

$is\_tiger(bob) \leftarrow has\_stripes(bob)$,

$is\_tiger(bob) \leftarrow is\_white(tom)$,

$is\_tiger(X) \leftarrow has\_stripes(X)$,

$is\_tiger(X) \leftarrow is\_tawny(X)$,

$is\_tiger(X) \leftarrow is\_white(X)$,

$is\_tiger(X) \leftarrow \}$

then the diagram of p.o. set $\mathcal{H}_E$ is as outlined in Figure 2.5

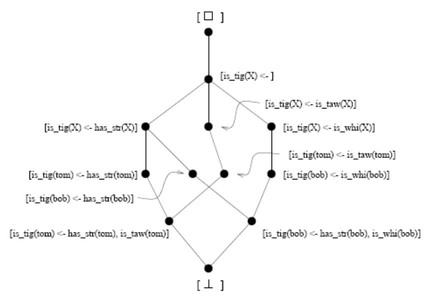

[□]
[is_tig(X) <-]
[is_tig(X) <- is_taw(X)]
[is_tig(X) <- has_str(X)]
[is_tig(X) <- is_whi(X)]
[is_tig(tom) <- is_taw(tom)]
[is_tig(tom) <- has_str(tom)]
[is_tig(bob) <- is_whi(bob)]
[is_tig(bob) <- has_str(bob)]
[is_tig(tom) <- has_str(tom), is_taw(tom)]
[is_tig(bob) <- has_str(bob), is_whi(bob)]
[⊥]

Figure 2.5: The lattice of $\mathcal{H}_E$ for the *is_tiger* relation.

2.1.2 Covers in the Subsume Order

The least generalization and the greatest specialization respectively concern generalizing or specializing a set of clauses to a single clause. What about generalizing and specializing single clauses? We will address this issue by investigating *covers* of clauses in the subsumption order.

It can be shown that there exist clauses which have no complete set of upward covers in the subsumption order. In fact, there are clauses which have no upward covers at all. For example, the clause $C = \{P(x_1, x_1)\}$ has no upward cover in $< \mathcal{C}, \succeq >$. This can be proved by defining

$$C_n = \{P(x_i, x_j) \mid i \neq j \text{ } and \text{ } 1 \leq i, j \leq n\}, \quad n \geq 2$$

and noting that each C_n properly subsumes C_{n+1} and C. That is $C_2 \succ C_3 \succ \ldots C_n \succ \ldots C$. From this result, we know that C has no complete set of upward covers.

Dually, for the downward cover there exists a clause C which has no finite complete set of downward covers[4]. So if this particular C does have a complete set of downward covers, this set must he infinite. This result is sufficient to later prove the negative result that an ideal downward refinement operator does not exist for the subsumption order. One such C is

$$C = \{P(x_1, x_2), P(x_2, x_1)\}$$

If we define[5]

$$E_n = \{P(y_1, y_2), P(y_2, y_3), \ldots, P(y_{n-1}, y_n), P(y_n, y_1)\}, \ n \geq 2$$

and

$$C_n = C \cup E_n, \ n \geq 3$$

[4]Whether all clauses have a (sometimes infinite) complete set of downward covers remains an open question.

[5]The clauses E_n have a special structure called a *cycle*.

then it can be shown that for any $n = 3^k \quad (k \geq 1)$, $C \succ C_n$. Further, by contradiction, one can show that there is no downward cover D of C, such that $D \succeq C_{3^k}$ for every $k \geq 1$ and subsequently, that, C has no finite complete set of downward covers in $< \mathcal{C}, \succeq >$.

2.2 The Implication Order

It is easy to see that implication is reflexive and transitive, and hence a quasi order. Implication between (Horn) clauses is undecidable, but using implication as a generality order is more desirable than using subsumption, for some important reasons:

1. It is better able to deal with recursive clauses. A clause C which implies another clause D, need not subsume this D. For instance, take

$$\begin{aligned} C = &\quad P(f(x)) \leftarrow P(x) \\ D = &\quad P(f^2(x)) \leftarrow P(x) \end{aligned}$$

   Then $C \models D$, but $C \not\succeq D$. Subsumption is too weak in this case. A further sign of this weakness is the fact that two tautologies need not be subsume equivalent, even though they are logically equivalent.

2. For the construction of least generalizations, subsumption is again not fully satisfactory and can over-generalize. For example,

   - If S consists of the clauses $D_1 = P(f^2(a)) \leftarrow P(a)$ and $D_2 = P(f(b)) \leftarrow P(b)$, then the LGS of S is $P(f(y)) \leftarrow P(x)$. the other hand, the clause $P(f(x)) \leftarrow P(x)$ seems more appropriate as a least generalization of S, since it implies D_1 and D_2, and is implied by the LGS. However, it does not subsume D_1.
   - Even for clauses without function symbols, the subsumption order may still be unsatisfactory. Consider $D_1 = P(x, y, z) \leftarrow P(y, z, x)$ and $D_2 = P(x, y, z) \leftarrow P(z, x, y)$. The clause D_1 is a resolvent of D_2 with D_2, and D_2 is a resolvent of D_1 with D_1; so D_1 and D_2 are logically equivalent. This means that D_1 is a least generalization under implication (LGI) of the set $\{D_1, D_2\}$. Yet the LGS of these two clauses is $P(x, y, z) \leftarrow P(u, v, w)$, which is clearly an over-generalization.

   As these examples also show, the subsumption order is particularly unsatisfactory if we consider recursive clauses: clauses where the same predicate symbol occurs both in a positive and a negative literal. Thus it is desirable to make the step from the subsumption order to the more powerful implication order.

$$\begin{tikzcd}
& \phi_1 \land \phi_2 \arrow[dl] \arrow[dr] & \\
\phi_1 \arrow[dr] & & \phi_2 \arrow[dl] \\
& \phi_1 \lor \phi_2 &
\end{tikzcd}$$

Figure 2.6: Least generalization and greatest specialization in first-order logic.

3. A further advantage of the implication order is that one can easily compare a set of clauses (a theory) with another theory or clause. For example, if $\Sigma = \{(P \leftarrow Q), (Q \leftarrow R)\}$ and $C = P \leftarrow R$, then we have $\Sigma \models C$. On the other hand, subsumption cannot be used here to compare the generality of Σ and C, because neither member of Σ subsumes C.

The main results on the implication order can be summed up as follows:

1. If α_1 and α_2 are two arbitrary first-order formulas, then it can be easily shown that their least generalization under implication (LGI) is just $\alpha_1 \land \alpha_2$, and their greatest specialization under implication (GSI) is just $\alpha_1 \lor \alpha_2$. See Figure 2.6. However, if α_l and α_2 are clauses, $\alpha_l \land \alpha_2$ is not a clause. Instead of using $\alpha_l \land \alpha_2$, we have to find some least clause as LGI, which implies both clauses α_l and α_2. Such a clause appears quite hard to find sometimes, as we will see subsequently. However, $\alpha_l \lor \alpha_2$ remains a clause and remains the GSI under the subsumption order on clauses.

2. Every finite set of clauses Σ which contains at least one function-free nontautologous clause, has a computable least generalization (LGI) under implication in $\mathcal{C}$. A related observation is that the problem whether $\Sigma \models C$, where Σ is a finite set of ground clauses and C is a ground clause, is decidable. This can be proved as follows: Let $C = L_1 \lor \ldots \lor L_n$ and $\mathcal{A}$ be the finite set of all ground atoms occurring in Σ and C. Now:

$$\begin{array}{lll} \Sigma \models C & \text{iff} & \Sigma \cup \{\neg L_1, \ldots, \neg L_n\} \text{ is unsatisfiable} \\ & \text{iff} & \Sigma \cup \{\neg L_1, \ldots, L_n\} \text{ has no Herbrand model} \\ & \text{iff} & \text{no subset of } \mathcal{A} \text{ is a Herbrand model of } \Sigma \cup \{\neg L_1, \ldots, L_n\} \end{array}$$

Since $\mathcal{A}$ is finite, the last statement is decidable. What follows from this is that the problem whether $\Sigma \models C$, where Σ is a finite set of function-free clauses and C is a clause, is decidable. The algorithm for computing LGI is outlined in Figure 2.7.

```
INPUT: A finite set $\mathcal{S}$ of clauses, containing at least one non-tautologous function-free clause;
OUTPUT: An LGl of $\mathcal{S}$ in $\mathcal{C}$;
Remove all tautologies from $\mathcal{S}$, call the remaining set $\mathcal{S}'$;
Let $m$ be the number of distinct terms (including subterms) in $\mathcal{S}'$, let $V = \{x_1, \ldots x_m\}$;
Let $\mathcal{G}$ be the (finite) set of all clauses which can be constructed from predicate symbols and constants in $\mathcal{S}'$ and variables in $V$;
Let $\{U_1, \ldots, U_n\}$ be the set of all subsets of $\mathcal{G}$;
Let $H_i$ be an LGS (computed using algorithm in Figure 2.4) of $U_i$, for every $1 \leq i \leq n$;
Remove from $\{H_1, \ldots, H_n\}$ all clauses which do not imply $\mathcal{S}'$ (since each $H_i$ is function-free, this implication is decidable), and standardize the remaining clauses $\{H_1, \ldots, H_q\}$ apart. ;
return $H = H_1 \cup \ldots \cup H_q$;
```

Figure 2.7: The LGI Algorithm.

3. Every finite set of clauses has a greatest specialization (GSI) under implication in $\mathcal{C}$. A GSI of a finite set $\mathcal{S}$ is the same as the GSS of $\mathcal{S}$ (refer Section 2.1.1), namely the union of the clauses in $\mathcal{S}$ after these are standardized apart.

   Finding least generalizations (under implication) of sets of clauses is common practice in ILP. On the other hand, the greatest specialization, which is the dual of the least generalization, is used hardly ever. Nevertheless, the GSI of two clauses D; and D2 might be useful. For example, suppose we have one positive example e^+, and two negative examples $e_1^- and e_2^-$, and suppose that D_1 implies e^+ and e_1^-, while D_2 implies e^+ and e_2^-. Then it might very well be that the GSI of D_1 and D_2 still implies e^+, but is consistent with respect to $\{e_1^-, e_2^-\}$. Thus we could obtain a correct specialization by taking the GSI of D_1 and D_2.

4. As a corollary, if $\mathcal{C}$ is a function-free clausal language, then $< \mathcal{C}, \models >$ is a lattice.

5. There exist pairs of Horn clauses which do not have an LGI in $\mathcal{H}$.

6. Similarly, there exist pairs of Horn clauses which do not have a GSI in $\mathcal{H}$.

7. For general clauses which all contain function symbols, the LGI-question is still open.

8. For covers, the negative results from the subsumption order carryover to the implication order.

   - Some clauses, such as $\{P(x_1, x_2)\}$, have no upward covers.

- Some clauses, such as $\{P(x_1, x_2), P(x_2, x_1)\}$, have no finite complete set of downward covers.

2.3 Inverse Reduction

Plotkin's reduction algorithm (Figure 2.1) finds a reduction D of C. In this section we present an algorithm which does the inverse: given a reduced clause D, the algorithm constructs (possibly non-reduced) members C of the subsume-equivalence class of D. This will be useful in our treatment of refinement operators. Since the subsume-equivalence class of D is infinite, we have to limit the scope of the algorithm. This is done by restricting the number of literals is C.

We will next state a lemma, which will find use subsequently.

Theorem 30 *Let C and D be clauses. If D is a reduction of C, then there is a substitution θ such that $C\theta = D$ and $L\theta = L$ for every $L\theta D$.*

Proof: Suppose D is a reduction of C, then there is a σ such that $C\sigma \subseteq D$. Then also $D\sigma \subseteq C\sigma \subseteq D$, since $D \subseteq C$. If $D\sigma \neq D$, then D would not be reduced, hence $D\sigma = C\sigma = D$.

Since σ is injective, for all $L_1, L_2 \in D$, if $L1\sigma \neq L_2\sigma$ then $L_1\sigma \neq L_2\sigma$, for otherwise $|D\sigma| < |D|$. Hence if $L_1\sigma = L_2\sigma$, then $L_1 = L_2$. For each $L \in D$, consider the following infinite sequence:

$$L, L\sigma, L\sigma^2, L\sigma^3, \ldots$$

Since $D\sigma = D$, each literal in this sequence is a member of D. D contains only a finite number of literals, so for some $i < j$ we must have $L\sigma^i = L\sigma^j$. Then from the injectivity, also $L = L\sigma^{j-1}$. For this L, define $n(L) = j - i$. Notice that $L = L\sigma^m$ if m is a multiple of $n(L)$. Let k be the least common multiple of all $n(L)$. Then $L\sigma^k = L$ for every $L \in D$.

Define $\theta = \sigma^k$. Then since $C\sigma = D$ and $D\sigma = D$, we have $C\theta = D$. $\Box$

From theorem 30, we know that for every non-reduced C such that $D \subset C$ and $D \equiv C$, we can find a θ such that $C\theta = D$ and θ only acts on variables not appearing in D. Thus C can be reduced by mapping $E = C \setminus D$ to literals in D. **In the inverse direction, we can find C by adding a set E to D, such that $E\theta \subseteq D$, where θ does not act on variables in D.** This is the idea used in the algorithm in Figure 2.8. If D is a reduced clause and m is some positive integer, then the algorithm finds a variant of every non-reduced C with m or less literals in the subsume-equivalence class of D.

As an illustration of the algorithm in Figure 2.8, if $D = \{P(x, x)\}$ and $m = 3$, then upto variants, $M_1 \in \{P(x, y), P(y, y)\}$ and $M_2 = \{P(y, z), P(x, z), P(y, x), P(z, z)\}$.

2.4 Generality order and ILP

The subsumption and implication orders discussed thus far are important for learning for the following reasons:

```
INPUT: A reduced clause $D$ and an integer $m$;
OUTPUT: Variants of every $C$ such that $D \equiv C$ and $|C| \leq m$;
Set $k = 0$;
If $|D| \leq m$, then output $D$;
while $l < (m - |D|)$ do
  Set $l$ to $l + 1$;
  for Every sequence $L_1, \ldots, L_i$ such that each $L_i \in D$, but the $L_i$'s are
  not necessarily distinct do
    Find every (up to variants) set $E = \{M_1, \ldots, M_t\}$ such that
     1.  Every $M_i$ contains at least one new variable not in $D$, and
     2.  If $x_1, \ldots, x_n$ are all those new variables, then there is a $\theta =$
         $x_1/t_1, \ldots, x_n/t_n$, such that $M_i\theta = L_i$, for $i = 1, \ldots, t$.
    ;
    return $D \cup E$;
  end for
end while
```

Figure 2.8: The Inverse Reduction Algorithm: It finds a *variant* of every non-reduced equivalent clause (equivalent to D) with m or less literals.

- They provide a generality ordering for hypotheses, thus structuring the hypothesis space.
- They can be used to prune large parts of the search space.
  1. When generalizing C to C', $C' \succ C$, all the examples covered by C will also be covered by C' (since if $\mathcal{B} \cup \{C\} \models e$ (e being an example) holds then also $\mathcal{B} \cup \{C'\} \models e$ holds). This property is used to prune the search of more general clauses when e **is a negative example: if e is inconsistent (covers a negative example) then all its generalizations will also be inconsistent**. Hence, the generalizations of C do not need to be considered.
  2. When specializing C to C', $C \succ C'$, an example not covered by C will not be covered by any of its specializations either (since if $\mathcal{B} \cup \{C\} \not\models e$ holds then also $\mathcal{B} \cup \{C'\} \not\models e$ holds). This property is used to prune the search of more specific clauses when e **is an uncovered positive example: if C does not cover a positive example none of its specializations will**. Hence, the specializations of C do not need to be considered.
- The generality orderings provide the basis for two important ILP techniques:
  1. bottom-up building of least general generalizations from training examples, relative to background knowledge, and

2. top-down searching of refinement graphs.

The principal generality orderings of interest are subsumption ($\succeq_\theta$) and implication ($\succeq_\models$). For clauses C, D, subsumption is *not* equivalent to implication

$$if\ C \succeq_\theta D \text{ then } C \succeq_\models D$$

$$\text{but}$$

$$\text{not vice} - \text{versa}$$

For example, the above holds for:

$C:\ natural(s(X)) \leftarrow natural(X)$

$D:\ natural(s(s(X))) \leftarrow natural(X)$

Recall the subsumption theorem; it is a key theorem linking subsumption and implication:

> If Σ is a set of clauses and D is a clause, then $\Sigma \models D$ iff D is a tautology, or there exists a clause $D' \succeq_\theta D$ which can be derived from Σ using some form of resolution.

When Σ contains a single clause C then the only clauses that can be derived are the result of *self-resolutions* of C. Thus, the difference between $C \succeq_\models D$ and $C \succeq_\theta D$ arises when C is self-recursive or D is tautological

Is there a principled approach for comparing generality orderings? Fortunately, the answer is yes.

> Given a set of clauses S, clauses $C, D \in S$ and quasi-orders $\succeq_1$ and $\succeq_2$ on S, then $\succeq_1$ is *stronger* than $\succeq_2$ if $C \succeq_2 D$ implies $C \succeq_1 D$. If also for some $C, D \in S$ $C \not\succeq_2 D$ and $C \succeq_1 D$ then $\succeq_1$ is *strictly stronger* than $\succeq_2$

It can be seen that as per above definition, the implication ordering is strictly stronger than the subsumption ordering. Quasi-orders that are increasinhgly weaker can be devised from stronger ones. Here are some other generality orderings, listed in decreasing order of strength.

- $C \succeq_\models D$ iff $C \models D$
- $C \succeq_\theta$ iff there is a substitution θ s.t. $C \subseteq D$
- $C \succeq_{\theta'} D$ iff every literal in D is *compatible* to a literal in C and $C \succeq_\theta D$.
- $C \succeq_{\theta''} D$ iff $|C| \geq |D|$ and $C \succeq_{\theta'} D$

What generality ordering should we choose for the ILP search procedure? We would like the strongest ordering that is practical In terms of tractability, logical implication between clauses is undecidable (even for Horn clauses). Subsumption is decidable but NP-complete (even for Horn clauses). Restrictions to the form of clauses can make subsumption efficient. Here are two example restrictions that make subsumption efficient.

- Determinate Horn clauses. There exists an ordering of literals in C and exactly one substitution θ s.t. $C\theta \subseteq D$.
- $k-local$ Horn clauses. Partition a Horn clause into k "disjoint" sub-parts and perform k independent subsumption tests.

The strongest quasi-order that is practical appears to be subsumption. Even that often requires restrictions (such as the two listed above) on the clauses being compared.

In summary, the subsumption order on clausal languages is used most often in ILP as the generality order (in contrast to the implication order), owing to its following properties:

1. Further, subsumption is more tractable and efficiently implementable. Subsumption between clauses is a decidable relation, whereas implication is not. The flip side is that subsumption is a weaker relation.
2. Equivalence classes under subsumption can be represented by a single reduced clause. Reduction can be undone by inverse reduction (*c.f.* Section 2.3).
3. Every finite set of clauses (function free or not)has a least generalization (LGS) and greatest specialization (GSS) under subsumption in $\mathcal{C}$. Hence $<\mathcal{C}, \succeq>$ is a lattice. The same is not true for the implication quasi-ordering $\succeq_{\models}$ (for restricted languages *lubs* for $\succeq_{\models}$ may well exist).

Order	*lub*	*glb*
$\succeq_\theta$	$\surd$	$\surd$
$\succeq_{\models}$	$\times$	$\surd$

4. Every finite set of Horn clauses has a least generalization (LGS) and greatest specialization (GSS) under subsumption in $\mathcal{H}$. Hence $<\mathcal{H}, \succeq>$ is a lattice. The same does not hold for the implication order.
5. The negative results for covers hold for subsumption as well as implication.

2.5 Incorporating Background Knowledge

Why does background knowledge matter? The answer is that combining the examples With what we already know often allows for the construction of a more satisfactory theory than can be glanced from the examples by themselves. To illustrate this, we consider [Bun88] the following two clauses as positive examples (not just ground atoms as examples):

$$D_1 = \quad CuddlyPet(x) \leftarrow Small(x), Fluffy(x), Dog(x)$$
$$D_2 = \quad CuddlyPet(x) \leftarrow Fluffy(x), Cat(x)$$

Given only these clauses, the most obvious way to generalize them is to take their LGS or LGI, which is the rather general clause $C = CuddlyPet(x) \leftarrow Fluffy(x)$.

However, suppose we have the following definite program $\mathcal{B}$ which expresses our background knowledge.

$$\begin{array}{ll} B_1 & Pet(x) \leftarrow Cat(x) \\ B_2 & Pet(x) \leftarrow Dog(x) \\ B_3 & Small(x) \leftarrow Cat(x) \end{array}$$

Given $\mathcal{B}$, we may also use the following clause as generalization:

$$D = CuddlyPet(x) \leftarrow Small(x), Fluffy(x), Pet(x).$$

since D together with $\mathcal{B}$ implies both examples. Note that without the background knowledge $\mathcal{B}$, our clause D neither subsumes nor implies the examples. If we interpret this example in human terms, the generalization D is much more satisfactory than C. After all, not every fluffy object (such as a teddy bear) is a cuddly pet. Thus the use of background knowledge allows us to find a better theory. Given the usefulness of background knowledge, it is important to outline a formalized way to reckon with it in our generality order.

There are three generality orders which are able to take background knowledge into account: (i) Plotkin's relative subsumption ($\succeq_{\mathcal{B}}$), (ii) Relative implication ($\models_{\mathcal{B}}$) and Buntine's (iii) Generalized subsumption ($\geq_{\mathcal{B}}$). Relative subsumption and relative implication apply to arbitrary clauses and the background knowledge B may be an arbitary finite set of clauses. Generalized subsumption only applies to definite program clauses and the background knowledge should be a definite program. We will state the existence of least generalizations in each of these orders, both in case we are dealing with a Horn language $\mathcal{H}$, and for a general clausal language $\mathcal{C}$. Further, each of the three orders can be related to some kind of deduction. Since empty background knowledge reduces relative and generalized subsumption to ordinary subsumption, and relative implication to ordinary implication, the negative results on covers for subsumption and implication carry over to the three orders: some clauses do not have finite complete sets of upward or downward covers in these orders. As to the existence and non-existence of least generalizations or greatest specializations in the three orders, we will only pay attention to least generalizations, since these are used much more often than their dual. In general, for all three, least generalizations do not always exist in the presence of background knowledge. However, certain restrictions on the background knowledge guarantee the existence of a least generalization for each of the three.

2.5.1 Plotkin's relative subsumption ($\succeq_{\mathcal{B}}$)

Definition 1 *Let C and D be clauses, and $\mathcal{B}$ be a set of clauses. We say $C \succeq_{\mathcal{B}} D$, if there is a substitution θ such that $\mathcal{B} \models \forall(C\theta \rightarrow D)$ (note that $C \rightarrow D$ need not be a clause).*

With respect to the last example, we can verify that $B_3 \succeq_{\{B_1,E\}} F$, where $E = CuddlyPet(x) \leftarrow Small(x), Fluffy(x), Pet(x)$ and $F = CuddlyPet(x) \leftarrow Fluffy(x), Cat(x)$. Informally, this can be seen as follows: suppose for every x it holds that if x is a cat, then x is small (*i.e.*, C is true). Using B_1, we also have that if x is a cat, then x is a pet. Thus if x is a fluffy cat, then x is small, fluffy, and a pet, and by E, x is a cuddly pet (*i.e.*, D is true). Following are some properties of relative subsumption:

1. Reflexivity and transitivity are easily proved, so relative subsumption is a quasi-order on clauses. Note that each set of clauses $\mathcal{B}$ induces its own quasi-order: the quasi-orders induced by $\mathcal{B} = \{P(a)\}$ and $\mathcal{B} = \{P(a), P(b)\}$ are different. Note also that if D is a tautology, then $C \succeq_{\mathcal{B}} D$ for any C and $\mathcal{B}$. Furthermore, it is also easy to see that if $C \succeq_{\mathcal{B}} D$ and $\mathcal{B} \subseteq \mathcal{B}'$, then $C \succeq_{\mathcal{B}'} D$.

2. Relative subsumption is strictly stronger than subsumption.

   (a) Firstly, it is easy to see that subsumption implies relative subsumption. If $C \succeq D$, then $C \succeq_{\mathcal{B}} D$, for some $\mathcal{B}$. But then $\forall(C\theta \succeq D)$ is a tautology, and $\mathcal{B} \models \forall(C\theta \rightarrow D)$ for any $\mathcal{B}$. Hence if $C \succeq D$, then $C \succeq_{\mathcal{B}} D$.

   (b) Now consider propositional atoms P, Q, and R. Let $C = P$, $D = Q$, and $\mathcal{B} = \{Q \leftarrow P\}$. Then $B \models (C \rightarrow D)$, so $C \succeq_{\mathcal{B}} D$. Since $C \not\succeq D$, we see that relative subsumption does not imply subsumption. This even holds for the case where $\mathcal{B}$ is empty and D is a tautology: if $\mathcal{B} = \emptyset, C = Q$, and $D = P \leftarrow P$, then $C \succeq_{\mathcal{B}} D$, but $C \not\succeq D$.

3. If C and D are non-tautologous clauses, and $\mathcal{B}$ a finite set of ground literals such that $\mathcal{B} \cap D = \emptyset$, then[6] $C \succeq_{\mathcal{B}} D$ iff $C \succeq (D \vee \overline{\mathcal{B}})$.

4. Relative subsumption coincides with ordinary subsumption for non-tautologous clauses and empty background knowledge[7]. That is, if C and D are non-tautologous clauses, then $C \succeq_{\mathcal{B}} D$ iff $C \succ D$.

5. $C \succeq_{\mathcal{B}} D$ iff there exists a deduction of $D from \{C\} \cup \mathcal{B}$ in which C occurs at most once as a leaf. The proof of this long and windy and will not be dealt with here.

6. Least generalizations under relative subsumption (abbreviated to LGRSs) need not exist in the general case. The following counter example, adapted from [NIb88] shows the non-existence of LGRSs both for the case of a clausal language $\mathcal{C}$, and for a Horn language $\mathcal{H}$.

   Let

$$D_1 = Q(a) \qquad D_2 = Q(b)$$
$$\mathcal{B} = \{P(a,y), P(b,y)\}$$

[6] Prove: EXERCISE

[7] Prove: EXERCISE

It can be shown there is no LGRS of $\{D_1, D_2\}$ relative to $\mathcal{B}$. Consider the following infinite sequence of clauses:

$$\begin{aligned}
C_1 &= Q(x) \leftarrow P(x, f(x)) \\
C_2 &= Q(x) \leftarrow P(x, f(x)), P(x, f^2(x)) \\
C_3 &= Q(x) \leftarrow P(x, f(x)), P(x, f^2(x)), P(x, f^3(x))
\end{aligned}$$

It is easy to see that $C_i \succ_{\mathcal{B}} C_{i+1}$ for every $i \geq 1$. We also have $C_i \succ_{\mathcal{B}} D_i$ and $C_i \succ_{\mathcal{B}} D_2$, for every $i \geq 1$. Suppose some clause D is an LGRS of $\{D_1, D_2\}$ relative to $\mathcal{B}$, then we should have $C \succeq_{\mathcal{B}} D$, for every $i \geq 1$. Then by point 5 above, for every $i \geq 1$, there exists a deduction of D from $\{C_i\} \cup \mathcal{B}$ in which C_i occurs at most once as a leaf. C_i cannot occur zero times as a leaf, because then a clause from $\mathcal{B}$ would simply subsume D, which is impossible. Thus for every $i \geq 1$, there exists a deduction of D from $\{C_i\} \cup B$, in which C, occurs once as a leaf. It cannot be that every C_i subsumes D, for then D would contain an instance of the term $f^i(x)$, for every $i \geq 1$. Thus for some j the deduction of D from $\{C_j\} \cup \mathcal{B}$ involves at least one resolution step. Since the members of $\mathcal{B}$ are atoms and C_j only occurs once as a leaf, the parent clauses in the first resolution step must be C_j and a member of $\mathcal{B}$. Suppose this member of $\mathcal{B}$ is $P(a, y)$. Then $P(a, y)$ must be unified with an atom of the form $P(x, f''(x))$ in the body of C_j. Then the head of the clause C_j is instantiated to $Q(a)$. This head will not be changed anymore in later resolution steps, so D would have $Q(a)$ as head - but then there is no deduction of $D_2 = Q(b)$ from $\{D\} \cup \mathcal{B}$.

Similarly, if the member of $\mathcal{B}$ in the resolution step had been $P(b, y)$ instead of $P(a, y)$, D would have had $Q(b)$ as head, and there would be no deduction of $D_1 = Q(a)$ from $\{D\} \cup \mathcal{B}$. Either way, the assumption that D is an LGRS of $\{D_1, D_2\}$ relative to $\mathcal{B}$ leads to a contradiction.

7. We however can identify a restriction on the background knowledge which guarantees existence (and computability) of an LGRS of any finite set of (horn) clauses. If $\mathcal{C}$ ($\mathcal{H}$) is a (horn) clausal language and $\mathcal{B} \subseteq \mathcal{C}$ ($\mathcal{B} \subseteq \mathcal{H}$) is a finite set of ground literals, then every finite non-empty set $\mathcal{S} \subseteq \mathcal{C}$ ($\mathcal{S} \subseteq \mathcal{H}$) of clauses has an LGRS in $\mathcal{C}$ ($\mathcal{H}$). This can proved as follows: If a clause D is a tautology or $\mathcal{B} \cap D \neq \emptyset$, then $\mathcal{B} \models D$, hence for any clause C we have $C \succeq_{\mathcal{B}} D$. Remove from S all tautologies and all D for which $B \cap D \neq \emptyset$, call the remaining set $\mathcal{S}'$. If $\mathcal{S}'$ is empty, any tautology is an LGRS of $\mathcal{S}$. If $\mathcal{S}' = \{D_1, ..., D_n\}$ is non-empty, then it follows easily from point 3) above that an LGS of $\{(D_1 \vee \overline{\mathcal{B}}), \ldots, (D_n \vee \overline{\mathcal{B}})\}$ (or equivalently of $\{(D_1 \leftarrow \mathcal{B}), \ldots, (D_n \leftarrow \mathcal{B})\}$) in $\mathcal{C}$ is an LGRS of $\mathcal{S}'$ in $\mathcal{C}$, and hence also of $\mathcal{S}$. The existence of such an LGS follows from Theorem 28. Thus if the background knowledge $\mathcal{B}$ is a finite set of ground literals, then we can construct an LGRS of a set $S = \{D_1, \ldots, D_n\}$ simply by constructing an LGS of $T = \{(D_1 \vee \overline{\mathcal{B}}), \ldots, (D_n \vee \overline{\mathcal{B}})\}$. Additionally, if all clauses in $\mathcal{S}$ are

horn clauses and each literal in $\mathcal{B}$ is a ground atom, then each $D_i \vee \overline{\mathcal{B}}$ is also a horn clause and so will the LGS of T. So the result follows for $\mathcal{H}$. This result has been exploited in the GOLEM system.

8. The non-existence of finite chains of covers, in lattices of (Horn) clauses ordered by subsumption carries over to the lattice of clauses ordered by relative subsumption.

Relative Subsumption and ILP

We will develop further on point (7) discussed above. When presented with an example e, candidate hypothesis in the form of a clause C and background knowledge $\mathcal{B}$, what does it mean for clause C to "relatively subsume" example e. Recall normal subsumption: $C \succeq e$ means $\exists\theta\ s.t.\ C\theta \subseteq e$. This also means $C\theta \models e$ or $\models (e \leftarrow C\theta)$.

$$\begin{array}{rc}
e: & gfather(henry, john) \leftarrow \\
B: & father(henry, jane) \leftarrow \\
 & father(henry, joe) \leftarrow \\
 & parent(jane, john) \leftarrow \\
 & parent(joe, robert) \leftarrow \\
C: & gfather(X,Y) \leftarrow father(X,Z), parent(Z,Y)
\end{array}$$

For this B, C, e with $\theta = \{X/henry, Y/john,\ Z/jane\}$, $B \cup \{C\theta\} \models e$ That is: $C \succeq_B e$ or equivalently, $B \models (e \leftarrow C\theta)$ (by subsumption theorem). However, note that $C \not\succeq e$. Clearly if $B = \emptyset$ normal subsumption between clauses results. Using the Deduction Theorem

$$\begin{aligned}
\mathcal{B} \models (e \leftarrow C\theta) &\equiv \mathcal{B} \cup \{C\theta\} \models e \\
&\equiv \mathcal{B} \cup \overline{e} \models \overline{C\theta} \\
&\equiv \{C\theta\} \models \overline{\mathcal{B} \cup \overline{e}} \\
&\equiv\ \models (\overline{\mathcal{B} \cup \overline{e}} \leftarrow C\theta)
\end{aligned}$$

That is, $C \succeq_{\mathcal{B}} e$ means $C \succeq \overline{\mathcal{B} \cup \overline{e}}$ or $C \models \overline{\mathcal{B} \cup \overline{e}}$. Recall that if $C_1 \succeq C_2$ then $C_1 \models C_2$. In fact, if $C_{1,2}$ are not self-recursive, then $C_1 \succeq C_2 \equiv C_1 \models C_2$

Let $a_1 \wedge a_2 \ldots$ be the ground literals true in all models of $B \cup \overline{e}$, that is these are all the members of the minimal model $MM(\mathcal{G}(\mathcal{B} \cup \overline{e}))$ (*c.f.* theorem 12 as well as theorem 17). Then

$$\begin{aligned}
\mathcal{B} \cup \overline{e} &\models a_1 \wedge a_2 \ldots \\
\overline{a_1 \wedge a_2 \wedge \ldots} &\models \overline{\mathcal{B} \cup \overline{e}} \equiv e \leftarrow \mathcal{B}
\end{aligned}$$

Let $\bot(\mathcal{B}, e) = \overline{a_1 \wedge a_2 \wedge \ldots}$. Note that $\bot(\mathcal{B}, e)$ (the 'most specific clause' given $\mathcal{B}, e$) may not always be finite. If $\mathcal{B}$ and e are both ground, it follows that $\bot(\mathcal{B}, e) \equiv (e \leftarrow \mathcal{B})$.

Now, if $D \succeq \bot(\mathcal{B}, e)$ then $D \models \bot(\mathcal{B}, e)$ and therefore $D \models \overline{\mathcal{B} \cup \overline{e}}$. Now, by transitivity of $\models$, if $D \models \bot(\mathcal{B}, e)$ then because $\bot(\mathcal{B}, e) \models e \leftarrow \mathcal{B}$, it should follow that $D \models e \leftarrow \mathcal{B}$. It can be further shown that if D, e are not self-recursive and $D \succeq \bot(\mathcal{B}, e)$ then $D \succeq \overline{\mathcal{B} \cup \overline{e}}$ (that is, $D \succeq (e \leftarrow \mathcal{B})$ or $D \succeq_B e$). In fact, if C, e are non self-recursive then

$$\begin{array}{ll} C \succeq \bot(B, e) & \equiv \\ C \models \overline{\mathcal{B} \cup \overline{e}} & \equiv \\ C \succeq \overline{\mathcal{B} \cup \overline{e}} & \equiv \\ C \succeq e \leftarrow \mathcal{B} & \end{array}$$

An example of finding $\bot$ is:

$\mathcal{B}$**:**

gfather(X,Y) $\leftarrow$ father(X,Z), parent(Z,Y)
father(henry,jane) $\leftarrow$
mother(jane,john) $\leftarrow$
mother(jane,alice) $\leftarrow$

e_i**:**

gfather(henry,john) $\leftarrow$

Conjunction of ground atoms provable from $B \cup \overline{e_i}$:

$\neg$parent(jane,john) $\wedge$
father(henry,jane) $\wedge$
mother(jane,john) $\wedge$
mother(jane,alice) $\wedge$
$\neg$gfather(henry,john)

$\bot(B, e_i)$**:**

gfather(henry,john) $\vee$ parent(jane,john) $\leftarrow$
father(henry,jane),
mother(jane,john),
mother(jane,alice)

D_i**:**

parent(X,Y) $\leftarrow$ mother(X,Y)

Mode declarations

Finding a clause D_i that subsumes $\bot(\mathcal{B}, e_i)$ is hampered by the fact that $\bot(\mathcal{B}, e_i)$ may be infinite! One workaround is to use a constrained subset of definite clauses to construct finite most-specific clauses. This can be enabled using **mode declarations**. An example set of mode declarations for the problem just considered is:

modeh(,gfather(+person,-person))*

modeh(,parent(+person,-person))*

modeb(,father(+person,-person))*

modeb(,parent(+person,-person))*

modeb(,mother(+person,-person))*

A *definite mode language* is defined as follows.

Definition 2 *Let $C : h \leftarrow b_1, \ldots, b_n$ be a definite clause with an ordering over literals. Let M be a set of mode declarations. C is in the definite mode language $\mathcal{L}(M)$ iff*

1. *h is the atom of a* modeh *declaration in M with every place-marker of $+type$ and $-type$ replaced with variables, and every place marker of $\#type$ replaced by a ground term.*
2. *Every atom b_i in body of C is an atom in a* modeb *declaration in M with $+, -, \#$ places being replaced as above.*
3. *Every variable of $+type$ in b_i is either of $+type$ in h or or $-type$ in a b_j $(1 \leq j < i)$*

Given a set of mode declarations M it is always possible to decide if a clause C is in $\mathcal{L}(M)$. Next, we define depth of variables.

Definition 3 *Let C be a definite clause, v be a variable in an atom in C, and U_v all other variables in body atoms of C that contain v*

$$d(v) = \begin{cases} 0 & \textit{if } v \textit{ in head of } C \\ (\max_{u \in U_v} d(u)) + 1 & \textit{otherwise} \end{cases}$$

For example, if $C : gfather(X, Y) \leftarrow father(X, Z), parent(Z, Y)$

Then $d(X) = d(Y) = 0$, $d(Z) = 1$. Putting together the definitions of mode language and depth, we next define depth bounded definite mode language.

Definition 4 *Let C be a definite clause with an ordering over literals. Let M be a set of mode declarations. C is in the depth-bounded definite mode language $\mathcal{L}_d(M)$ iff all variables in C have depth at most d*

As an example, the clause for $gfather/2$ earlier is in $\mathcal{L}_2(M)$.

We will state some properties of $\perp(\mathcal{B}, e_i)$ without their proofs. For every $\perp(\mathcal{B}, e_i)$ it is the case that:

There is a $\perp_d(\mathcal{B}, e_i)$ in $\mathcal{L}_d(M)$ s.t. $\perp_d(\mathcal{B}, e_i) \succeq \perp(\mathcal{B}, e_i)$

$\perp_d(\mathcal{B}, e_i)$ is finite

If $C \succeq \perp_d(\mathcal{B}, e_i)$ then $C \succeq \perp(\mathcal{B}, e_i)$

$\perp(\mathcal{B}, e_i)$ may not be Horn

$\perp(\mathcal{B}, e_i)$ may not be finite

Least upper bound of Horn clauses e_1, e_2 is:

$$lgg_B(e_1, e_2) = lgg(\perp(B, e_1), \perp(B, e_2))$$

Greatest lower bound of Horn clauses e_1, e_2 is:

$$glb_b(e_1, e_2) = glb(\perp(B, e_1), \perp(B, e_2))$$

Let us take an example of finding $\perp_i$:

$\perp(B,\ e_i)$:

gfather(henry,john) $\vee$ parent(jane,john) $\leftarrow$
 father(henry,jane),
 mother(jane,john),
 mother(jane,alice)

modes:

modeh(,parent(+person,-person))*
modeb(,mother(+person,-person))*
modeb(,father(+person,-person))*

$\perp_0(B, e_i)$**:**

parent(X,Y) $\leftarrow$

$\perp_1(B, e_i)$**:**

parent(X,Y) $\leftarrow$
 mother(X,Y),
 mother(X,Z)

2.5.2 Relative implication ($\models_{\mathcal{B}}$)

Definition 5 *Let C and D be clauses, and $\mathcal{B}$ be a set of clauses. C (logically) implies D relative to $\mathcal{B}$, denoted $C \models_{\mathcal{B}} D$, if $\{C\} \cup \mathcal{B} \models_{\mathcal{B}} D$.*

For example, if $\mathcal{B} = \{P(a)\}$, $C = P(f(x)) \leftarrow P(x)$ and $D = P(f^2(a))$, then $C \models_{\mathcal{B}} D$, because there is a deduction of D from $\{C\} \cup \mathcal{B}$. However, we have $C \not\succeq_{\mathcal{B}} D$ because C has to be used more than once in the deduction of D. Following are some properties of relative subsumption:

1. Relative implication is perhaps the most obvious way to take background knowledge into account.

2. Obviously relative implication is reflexive and transitive so it can serve as a quasi-order on a set of clauses.

3. It is also obvious that if $C \models D$ then $C \models_{\mathcal{B}} D$. The converse need not hold though. Consider $C = P(a) \leftarrow P(b)$, $D = P(a)$, and $\mathcal{B} = \{P(b)\}$: then $C \models_{\mathcal{B}} D$, but $C \not\models D$. '

4. Relative subsumption implies relative implication but not conversely. If $C \succeq_{\mathcal{B}} D$, then $\mathcal{B} \models \forall(C\theta \rightarrow D)$ (for some θ). If $C \models_{\mathcal{B}} D$ then $\{C\} \cup \mathcal{B} \models D$, which is equivalent to $\mathcal{B} \models \forall(C) \rightarrow \forall(D)$ by the deduction theorem.

5. $C \models_{\mathcal{B}} D$ iff there exists a deduction of D from $\{C\} \cup \mathcal{B}$ (contrast this with that for relative subsumption). This is another testimony to the fact that relative implication is a strictly stronger quasi-order than relative subsumption.

6. The negative results for existence of least generalizations under implication carry over to relative implication (LGRI), since ordinary logical implication is just a special case of relative implication. Further, the only positive result for LGI that was stated on page 110 in point (2) is also negative for LGRI. As an example, consider

$$\begin{aligned} &D_1 = P(a) && D_2 = P(b) \\ &\mathcal{B} = \{(P(a) \vee \neg Q(x)), (P(b) \vee \neg Q(x))\} \end{aligned}$$

   It can be shown that the set $\mathcal{S} = \{D_1, D_2\}$ has no LGRI relative to $\mathcal{B}$ in $\mathcal{C}$. Suppose indeed D is an LGRI of $\mathcal{S}$ relative to $\mathcal{B}$. Note that if D contains the literal $P(a)$, then the Herbrand interpretation which makes $P(a)$ true, and which makes all other ground atoms false, would be a model of $\mathcal{B} \cup \{D\}$ but not of D_2, so then we would have $D \not\models_{\mathcal{B}} D_2$. Similarly, if D contains $P(b)$ then $D \not\models_{\mathcal{B}} D_1$. Hence D cannot contain $P(a)$ or $P(b)$. Now let d be a constant not appearing in D. Let $C = P(x) \vee Q(d)$, then $C \models_{\mathcal{B}} \mathcal{S}$. By the definition of an LGRI, we should have $C \models_{\mathcal{B}} D$. Then by the Subsumption Theorem, there must be a derivation from $\mathcal{B} \cup \{C\}$ of a clause E, which subsumes D. The set of all clauses which can be derived (in 0 or more resolution steps) from $\mathcal{B} \cup \{C\}$ is $\mathcal{B} \cup \{C\} \cup \{(P(a) \vee P(x)), (P(b) \vee P(x)\}$. But none of these clauses subsumes D, because D does not contain the constant d, nor the literals $P(a)$ or $P(b)$. Hence $C \not\models_{\mathcal{B}} D$, contradicting the assumption that D is an LGRI of $\mathcal{S}$ relative to $\mathcal{B}$ in C. Thus, in general an LGRI of $\mathcal{S}$ relative to $\mathcal{B}$ need not exist.

7. We can identify a special case in which the existence of an LGRI is guaranteed. Let C and D be clauses, and $\mathcal{B}$ be a finite set of fundion-free ground literals. Then $C \models_{\mathcal{B}} D$ iff $C \models (D \cup \overline{\mathcal{B}})$. Further, if $\mathcal{S} \subseteq \mathcal{C}$ is a finite set of clauses, containing at least one D for which $D \cup \overline{\mathcal{B}}$ is non-tautologous and function-free, then $\mathcal{S}$ has an LGRI in $\mathcal{C}$. As a special case of this, if $\mathcal{C}$ is itself function free clausal language, then for any finite $\mathcal{S} \subseteq \mathcal{C}$, $\mathcal{S}$ has an LGRI in $\mathcal{C}$.

   This can be proved as follows: Suppose $C \models_{\mathcal{B}} D$ *i.e.*, $\{C\} \cup \mathcal{B} \models D$. Let M be a model of C. Then we need to show that M is also a model of

$D \cup \overline{\mathcal{B}}$. If M is not a model of $\mathcal{B}$, then it is a model of at least one literal in $\overline{\mathcal{B}}$, and hence of the clause $D \cup \overline{\mathcal{B}}$. If on the other hand, M is a model of $\mathcal{B}$ then M is also a model of D, because $\{C\} \cup \mathcal{B} \models D$. Then M is also a model of $D \cup \overline{\mathcal{B}}$. This prove the 'only if' part.

Now suppose $C \models (D \cup \overline{\mathcal{B}})$. Let M be a model of $\{C\} \cup \mathcal{B}$. Then we need to show that M is also a model of D. Now, M is a model of C and hence of the clause $D \cup \overline{\mathcal{B}}$. But M is also a model of $\mathcal{B}$, and hence not a model of $\overline{\mathcal{B}}$. Therefore, M must be a model of D. This proves the 'if' part.

Now if $\mathcal{S} = \{D_1, D_2, \ldots, D_n\}$, it follows that an LGI in $\mathcal{C}$ of $T = \{(D_1 \cup \overline{\mathcal{B}}), \ldots, (D_n \cup \overline{\mathcal{B}})\}$ is also an LGRI of $\mathcal{S}$ in $\mathcal{C}$ (which exists according to point 2 on page 110).

2.5.3 Buntine's generalized subsumption ($\geq_{\mathcal{B}}$)

Definition 6 *Let C and D be definite program clauses and $\mathcal{B}$ be a definite program, comprising the background knowledge. We say that $C \geq_{\mathcal{B}} D$ (g-subsumes), if for every Herbrand model M of $\mathcal{B}$ and every ground atom A such that D covers A under M, we have that C covers A under M. Definite clause C is said to 'cover' atom A under M if there exists a ground substitution θ for C (that is, $C\theta$ is ground), such that M is a model for $C^-\theta$ and $C^+\theta = A$.*

Generalized subsumption applies only to definite program clauses. Loosely speaking, generalized subsumption says that for definite clauses C and D, C is more general that D, in any situation consistent with what we already know (through $\mathcal{B}$), C can be used to prove at least as many results as D. For example, if $\mathcal{B}$ consists of the background knowledge on page 116 and C and D are defined somewhat similar to D_1 and D_2 as

$$\begin{aligned} C = &\quad CuddlyPet(x) \leftarrow Small(x), Pet(x) \\ D = &\quad CuddlyPet(x) \leftarrow Cat(x) \end{aligned}$$

then, we can show that $C \geq_{\mathcal{B}} D$. For suppose M is a Herbrand model of $\mathcal{B}$ and D covers some grond atom $A = CuddlyPet(t)$ under M, then for $\theta = \{x/t\}$, $D^-\theta = Cat(t)$ is true under M. Since M is a model of $\mathcal{B}$, in particular of B_1 and B_3, $Pet(t)$ and $Small(t)$ must be true under M as well. Then $C^-\theta = Small(t) \wedge Pet(t)$ is true under M, and $C^+\theta = A$, so C also covers A under M. Hence $C \geq_{\mathcal{B}} D$. In natural language, this can be rephrased as

If small pets are cuddly pets, then cats are cuddly pets, since we already know that cats are small pets.

Some of the properties of $\geq_{\mathcal{B}}$ are:

1. Just as subsumption implies relative subsumption, subsumption also implies g-subsumption. Suppose $C \succeq D$. Then $C\theta \subseteq D$, so $C^+\theta = D^+$ and $C^-\theta \subseteq D^-\theta$ for some θ. If D covers some A under some I, there is a γ such that $D^-\gamma$, is true under I and $D^+\gamma = A$. But then $C^+\theta\gamma = D^+\gamma = A$,

and $C^{-}\gamma \subseteq D^{-}\gamma$, is true under I, so C covers A under I as well. Hence, if $C \succeq D$, then $C \geq_{\mathcal{B}} D$.

The converse need not hold: if $\mathcal{B} = \{P(a)\}$, $C = Q(a) \leftarrow P(a)$ and $D = Q(a)$, then $C \geq_{\mathcal{B}} D$ but $C \not\succeq D$.

2. Generalized subsumption is reflexive and transitive with respect to some definite program $\mathcal{B}$, so it can serve as a quasi-order on a set of definite clauses.

3. $C \geq_{\mathcal{B}} D$ iff there exists an SLD-deduction of D, with C as top clause and members of $\mathcal{B}$ as input clauses. It follows from this that g-subsumption reduces to ordinary subsumption in the presence of empty background knowledge, as was the case for relative subsumption. That is, if C and D are definite program clauses, then $C \geq_{\emptyset} D$ iff $C \succeq D$.

4. From the previous point, it follows that if C and D are definite program clauses, and $\mathcal{B}$ is a definite program, then if $C \geq_{\mathcal{B}} D$, it follows that $C \succeq_{\mathcal{B}} D$. That is, g-subsumption implies relative subsumption. But the converse does not hold. This is sufficient to prove that if an LGG (lub) does not exist under relative subsumption (such as for the example

5. An LGGS of a finite set $\mathcal{S}$ (of definite program clauses which all have the same predicate symbol in their respective heads) always exists either if

   (a) All clauses in $\mathcal{S}$ are atoms, and the background knowledge $\mathcal{B}$ implies only a finite number of ground atoms (i.e., $M_{\mathcal{B}}$ is finite)

   (b) $\mathcal{S}$ and $\mathcal{B}$ are all function-free (see [Bun88]).

   (c) $\mathcal{B}$ is ground. This case differs from the first, because $\mathcal{B}$ may imply only a finite number of ground examples, and still be non-ground itself. For example, $\mathcal{B} = \{P(a), (Q(x) \leftarrow P(x))\}$.

Actually, these three cases are special cases of the following theorem:

Theorem 31 *Let $\mathcal{H}$ be a Horn language and $\mathcal{B}$ be a definite program. Let $\mathcal{S} = \{D_1, \dots, D_n\} \subseteq \mathcal{H}$ be a finite non-empty set of definite program clauses, such that all D_i, have the same predicate symbol in their head. Furthermore, for every $1 \leq i \leq n$, let σ_i be a Skolem substitution for D_i, with respect to $\mathcal{B} \cup \mathcal{S}$, and M_i be the least Herbrand model of $B \cup D_i^{-}\sigma_i$. If every M_i is finite, then there exists an LGGS of $\mathcal{S}$ in $\mathcal{H}$.*

Thus if the least Herbrand models mentioned in the theorem are indeed finite, then we can find an LGGS of a set $\{D_1, \dots, D_n\}$ simply by constructing an LGS of $\{(\{D_1^{+}\} \cup \overline{M_1}), \dots, (\{D_n^{+}\sigma_n\} \cup \overline{M_n})\}$.

2.6 Using Generalization and Specialization

The normal problem of inductive logic programming is to find a correct theory, a set of clauses which implies all given positive examples and which is consistent with respect to the given negative examples. Usually, it is not immediately obvious which set of clauses we should pick as our theory. Rather, we will have to search among the permitted clauses for a set of clauses with the right properties. If a positive example is implied by the theory, we should search for a more general theory. On the other hand, if the theory is not consistent with respect to the negative examples, we should search for a more specific theory - for instance, by replacing a clause in the theory by more specific clauses - such that the theory becomes consistent. Thus, the two most important operations in ILP are *generalization* and *specialization*. Repeated application of such generalization and specialization steps may finally yield a correct theory.

To systematically facilitate this search, it would be very handy if the set of clauses that has to be searched, is somehow structured. Fortunately, this is so as seen in chapter 1. We had seen several alternatives for what it means for some clause to be more general than another clause. Since generalization (or dually, specialization) can proceed along the lines of such a generality order, using such an order can direct the search for a correct theory. For example, least generalizations can be used to generalize given finite sets of examples.

2.7 Using the Cover Structure: Refinement Operators

One way to weaken a existing theory which is too strong is to find a false member of the theory Σ, and delete this clause from the theory. However, deleting a clause might make the theory in turn too weak. A way to strengthen the theory again, is to add weaker versions of previously deleted clauses. For instance, suppose the clause $P(x)$ is false under I, and has been deleted from Σ. It might be that $P(f(x))$, which is a "refinement" of $P(x)$, is true under I. Thus the theory might be strenghened by adding $P(f(x))$ to it.

The finite downward and upward cover chain algorithm provide the general direction of search in any search over the lattice structure over theories (atomic or otherwise). In *top-down search*, we want to find some unknown specialization $\mathbf{l}_2$ of $\mathbf{l}_1$. Then we should use substitutions to try and find a chain of downward covers starting from $\mathbf{l}_1$ as in Algorithm 1.15. Since such finite chains always exist for atoms, we can restrict attention to downward covers of $\mathbf{l}_1$, downward covers of downward covers of $\mathbf{l}_1$, *etc.* Recall from the algorithm in Figure 1.15 that the progress from $\mathbf{l}_i$ to $\mathbf{l}_{i+1}$ is achieved by applying one of the following substitutions:

1. $\{X/f(X_1,\ldots,X_k)\}$ where X is a variable in $\mathbf{l}_i$, $X_1,\ldots,X_k$ are distinct variables that do not appear in $\mathbf{l}_i$, and f is some k-ary function symbol in the language
2. $\{X/c\}$ where X is a variable in $\mathbf{l}_i$, and c is some constant in the language
3. $\{X/Y\}$ where X, Y are distinct variables in $\mathbf{l}_i$

In ILP, these 3 operations define a "downward refinement operator"

On the other hand, in *bottom-up search* we want to find some unknown generalization $\mathbf{l}_1$ of $\mathbf{l}_2$. In that case, we should use inverse substitutions to find a chain of upward covers from $\mathbf{l}_2$ to $\mathbf{l}_1$.

A systematic way to find refinements of clauses, is by using a refinement operator. There are two kinds of refinement operators: upward and downward ones. An upward refinement operator computes a set of generalizations of a given clause, a downward refinement operator computes a set of specializations. What constitutes a 'specialization' or 'generalization' of a clause, is determined by a generality order $\succeq$ (such as subsumption, implication, relative subsumption, relative implication, *etc.*) on clauses. Then we can say that C is a generalization of D (dually: C is a specialization of D), if $C \succeq D$ holds. In each of these orders, the empty clause $\Box$ is the most general clause.

Downward refinement operators compute sets of specializations of a clause, upward ones compute sets of generalizations. Refinement operators are defined for a set of formulae S with a quasi-ordering $\succeq$. There are two refinement operators.

- ρ is a *downward refinement operator* if $\forall C \in S : \rho(C) \subseteq \{D | D \in S \text{ and } C \succeq D\}$
- δ is an *upward refinement operator* if $\forall C \in S : \delta(C) \subseteq \{D | D \in S \text{ and } D \succeq C\}$

For example, with an equality theory $= /2$, $D \in \rho(C)$ (downward refinement operator) if

$$D = \begin{cases} p(X_1, X_2, \ldots, X_{n_p}) & \text{if } C = \Box \text{ and } p/n_p \in \mathcal{L} \\ & \text{and the } X_i \text{ are distinct} \\ C \cup \{\neg l\} & \text{otherwise} \end{cases}$$

where

$$l = \begin{cases} V = W & \text{where } V, W \text{ occur in } C \\ V = f(X_1, X_2, \ldots, X_{n_f}) & \text{where } V \text{ occurs in } C \\ & \text{and } f/n_f \in \mathcal{L} \text{ and} \\ & \text{the } X_i \text{ are distinct} \\ q(X_1, X_2, \ldots, X_{n_q}) & \text{where } q/n_q \in \mathcal{L} \\ & \text{and the } X_i \text{ occur in } C \end{cases}$$

is an example of a refinement operator.

2.7.1 Example

We assume the set of clauses $\mathcal{C}_h$ is ordered by such a generality order $\succeq$. Shapiro's top-down approach only employs a downward refinement operator ρ, so $\rho(C)$ is a set of specializations of a given clause C. We start with $\Sigma = \{\Box\}$. This is clearly too strong, since it implies any clause. Hence we want to find specialisatione of $\Box$, We use the set $\rho(\Box)$ for this. If $\rho(\Box)$ is still too strong, its false members in turn be replaced by their refinements, and so on. Thill allows us to search stepwisely through the generality order. This stepwise approach will

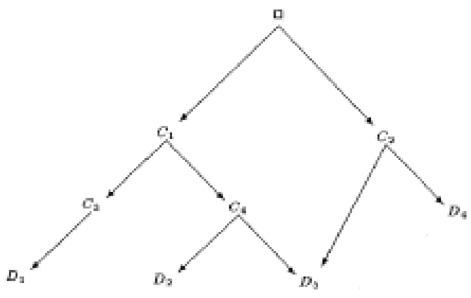


Figure 2.9: Paths through the refinement operator ρ.

only work if there is a path (a number of refinement steps) from *Box* to every clause in atleast one correct theory, For instance, suppose $\Sigma = \{D_2, D_3, D_4\}$ is a correct theory. Let the refinement operator ρ be such, that $\rho(\Box) = \{C_1, C_2\}$, $\rho(C_1) = \{C_3, C_4\}$, $\rho(C_2) = \{D_3, D_4\}$, $\rho(C_3) = \{D_1\}$, and $\rho(C_4) = \{D_2, D_3\}$. Starting from $\Box$, we can reach Σ by considering $\rho(\Box)$, $\rho(C_1)$, $\rho(C_2)$, $\rho(C_4)$. See Figure 2.9

2.7.2 Ideal Refinement Operators

The sets of one-step refinements, n-step refinements, and refinements of some $C \in \mathcal{C}$, are respectively:

$$\rho^1(C) = \rho(C)$$

$$\rho^n(C) = \{D \mid \textit{ there is an } E \in \rho^{n-l}(C) \textit{ such that } D \in \rho(E)\},\ n \geq 2$$

$$\rho^*(C) = \rho^1(C) \cup \rho^2(C) \cup \rho^3(C) \ldots$$

A ρ-chain from C to D is a sequence $C = C_0, C_1, \ldots, C_n = D$, such that $C - i \in \rho(C_{i-1})$ fpr every $1 \leq i \leq n$. A refinement operator induces a refinement graph. This is a directed graph which has the members of $\mathcal{C}$ as nodes (here variant clauses in $\mathcal{C}$ can be viewed as the same node), and which contains an edge from C to D just in case $D \in \rho(C)$. This refinement graph is the space that is searched for candidates to include in the theory (we will later see refinements over theories).

Some desirable properties of ρ (and dually δ) are that they be:

1. **Locally Finite:** $\forall C \in \mathcal{C}$: $\rho(C)$ is finite and computable. Otherwise ρ will be of limited use in practice.

2. **Complete:** $\forall C \succ D$: $\exists E \in \rho^*(C)$ s.t. $E \sim D$. That is, every specialization should be reachable by a finite number of applications of the operator.

3. **Proper:** $\forall C \in \mathcal{C} : \rho(C) \subseteq \{D | D \in S \text{ and } C \succ D\}$. That is, it is better only to compute proper generalizations of a clause, for otherwise repeated

application of the operator might get stuck in a sequence of equivalent clauses (such as the application of inverse reduction in Section 2.3), without ever achieving any real specialization.

A refinement operator is *ideal* if it is locally finite, complete, and proper. An ideal refinement operator induces a refinement graph in which only a finite number of edges start from each node (locally finiteness), in which there exists a path of finite length from C to a member of the equivalence class of D whenever $C \succ D$ (completeness), and which contains no cycles (by properness).

It can be shown that ideal downward $\rho_{\mathcal{A}}$ and upward $\delta_{\mathcal{A}}$ refinement operators exist for the simplest of our quasi-orders: the set of atoms ordered by subsumption.

Definition 7 *Let $\mathcal{A}$ be the set of atoms in a language. The downward refinement operator $\rho_{\mathcal{A}}$ for $\mathcal{A}$ is defined as follows:*

1. *For every variable z in an atom A and every n-ary function symbol f in the language, let $x_1, \ldots, x_n$ be distinct variables not appearing in A. Let $\rho_{\mathcal{A}}(A)$ contain $A\{z/f(x_1, \ldots, x_n)\}$.*

2. *For every variable z in A and every constant a in the language let $\rho_{\mathcal{A}}(A)$ contain $A\{z/a\}$.*

3. *For every two distinct variables x and z in A, let $\rho_{\mathcal{A}}(A)$ contain $A\{z/x\}$.*

Note that $\rho_{\mathcal{A}}(A)$ may still contain variants. For instance, $\rho_{\mathcal{A}}(P(x, y))$ contains both $P(x, x)$ and $P(y, y)$. Clearly, in a practical application we can ignore any redundant variants. The three different kinds of atoms in $\rho(A)$ correspond exactly to the three kinds of downward covers that we discussed in theorem 25. It can be proved easily from the properties of sets of covers of atoms described on page 87 that if $\mathcal{A}$ contains a finite number of constants function and predicate symbols, then $\rho_{\mathcal{A}}$ is ideal.

An ideal upward refinement operator $\delta_{\mathcal{A}}$ can be defined straightforwardly as follows:

Definition 8 *Let $\mathcal{A}$ be the set of atoms in a language. The upward refinement operator $\delta_{\mathcal{A}}$ for $\mathcal{A}$ is defined as follows:*

1. *For every $t = f(x_1, \ldots, x_n)$ in A, for which $x_1, \ldots, x_n$ are distinct variables and each occurrence of some x_i in A is within an occurrence of t, $\delta_{\mathcal{A}}(A)$ contains an atom obtained by replacing all occurrences of t in A by some new variable z not in A.*

2. *For every constant a in A and every non-empty subset of the set of occurrences of a in A, $\delta_{\mathcal{A}}(A)$ contains an atom obtained by replacing those occurrences of a in A by some new variable z not in A.*

3. *For every variable x in A and every non-empty proper subset of the set of occurrences of x in A, $\delta_{\mathcal{A}}(A)$ contains an atom obtained by replacing*

those occurrences of x in A by some new variable z not in A. Note that in this last item, we cannot replace all occurrences of x by a new variable z, for then we would get a variant of A. For instance, $P(z, a, z)$ is a variant of $A = P(x, a, x)$.

As in the case of $\rho_{\mathcal{A}}$, it easily follows that $\delta_{\mathcal{A}}$ is locally finite, complete and proper. We do not even have to presuppose a finite number of constants function and predicate symbol for this, because when constructing JA(A) w only have to deal With the fimte number of symbols in A-there is no need to introduce new constants, functions or predicates.

2.7.3 Refinement Operators on Clauses for Subsumption

Unfortunately, the ideal conditions described for atoms cannot all be met at the same time for more complex orders. Ideal refinement operators do not exist for full clausal languages or Horn languages ordered by subsumption or by the stronger orders. This negative result is a concequence of the fact that finite complete sets of covers do not always exist[8]. We had seen (*c.f.* Section 2.1.2) that the existence of finite chains in lattices of atoms ordered by subsumption does *not* carry over to Horn clauses ordered by subsumption. This had followed from the observation that there are clauses which have no *finite* and complete set of downward covers. This makes it impossible to devise an ILP program that uses a refinement operator that is both complete and non-redundant. Therefore, there are no upward (downward) refinement operators that are locally finite as well as complete as well as proper for sets of clauses That is, for clausal languages ordered by subsumption $\succeq_\theta$ or stronger orders, ideal refinement operators do not exist.

In order to define a refinement operator for full clausal languages, it will be required to drop one of the three properties of idealness. Of the three conditions of locally finiteness, completeness and properness, finiteness seems indispensable: an infinite set $\rho(C)$ of refinements of a clause C cannot be handled well, because it would then be impossible to test all members of $\rho(C)$ in finite time. Furthermore, it is obvious that completeness is also a very valuable property, if you want to be able to guarantee that a solution will always be found whenever one exists. Of the three ideal properties, properness seems the least important and can be compromised. Ideal refinement operators can be approximated by

1. *Dropping the requirement of properness:* Locally finiteness and completeness are considered to be the two most important properties, so dropping the 'properness' is a common practice. We will define downward (ρ_L) and upward (δ_U) refinement operators that are locally finite and complete, but improper. For subsumption, such refinement operators exist, both for the downward and for the upward case. If C subsumes D, then $C\theta \subseteq D$ for some substitution θ. Thus specialization under subsumption can be

[8]In fact, it can be proved that if there exists an ideal downward (upward) refinement operator for $< \mathcal{C}, \succeq >$, then every $C \in \mathcal{C}$ will have a finite complete set of downward (upward) covers

achieved by applying (elementary) substitutions and adding literals. In fact, when adding literals, it is sufficient to add only most general literals, since these can always be instantiated by a substitution later on to get the right literals. A literal $P(x_1, \ldots, x_n)$ or $\neg P(x_1, \ldots, x_n)$ is most general with respect to a clause C, if $x_1, \ldots, x_n$ are distinct variables not appearing in C.

Definition 9 *Let $\mathcal{C}$ be a clausal language. The locally finite and complete downward refinement operator $\rho_{\mathcal{L}}$ for $< \mathcal{C}, \succeq >$ is defined as follows:*

(a) **Apply a substitution to a clause** C *in one of the following ways:*

  i. For every variable z in a clause C and every n-ary function symbol f in the language, let $x_1, \ldots, x_n$ be distinct variables not appearing in C. Let $\rho_{\mathcal{L}}(C)$ contain $C\{z/f(x_1, \ldots, x_n)\}$.

  ii. For every variable z in C and every constant a in the language, let $\rho_{\mathcal{L}}(C)$ contain $C\{z/a\}$.

  iii. For every two distinct variables x and z in C, let $\rho_{\mathcal{L}}(C)$ contain $C\{z/x\}$.

(b) **Add a literal to a clause** C**:** *For every n-ary predicate symbol P in the language, let $x_1, \ldots, x_n$ be distinct variables not appearing in C. Then $\rho_{\mathcal{L}}$ contains both $C \cup \{P(x_1, \ldots, x_n)\}$ and $C \cup \{\neg P(x_1, \ldots, x_n)\}$. Note that the literals $P(x_1, \ldots, x_n)$ and $\neg P(x_1, \ldots, x_n)$ that are added to C by the fourth item in the definition are most general with respect to C.*

The proof of locally finiteness and completeness is straightforward. Sincce we already know that no ideal operators exist for this case, $\rho_{\mathcal{L}}$ cannot be proper. For instance, if $C = \{P(x)\}$ and $D = \{P(x), P(y)\}$, then $D \in \rho_{\mathcal{L}}(C)$ and $C \sim D$. However, this D is needed in a $\rho_{\mathcal{L}}$-chain from C to $\{P(a)P(b)\}$ as follows: $\{P(x)\}, \{P(x), P(y)\}, \{P(a), P(y)\}, \{P(a), P(b)\}$. Notice that for every $\Box \neq C$, we have $\Box \succ C$, so $\rho^*(\Box)$ contains a clause which is subsume-equivalent to C. In other words: If we start with the empty clause (as Shapiro's Model Inference Algorithm does), then for every $C \in \mathcal{C}$, a clause C' such that $C \sim C'$ can be reached by means of $\rho_{\mathcal{L}}$.

Definition 10 *Let $\mathcal{C}$ be a clausal language. The locally finite and complete upward refinement operator δ_u for $< \mathcal{C}, \succeq >$ is defined as follows:*

(a) **Apply one of the following inverse substitution operations:**

  i. For every $t = f(x_1, \ldots, x_n)$ in C, for which all x_i are distinct variables and each occurrence of x_i in a clause C is within an occurrence of t, $\delta_u(C)$ contains the clause obtained by replacing all occurrences of t in C by some new variable z not previously in C.

ii. *For every constant a in C and every non-empty subset of the set of occurrences of a in $\overline{C} = dup(C, a)$, if $\overline{D}$ is the ordered clause obtained by replacing those occurrences of a in $\overline{C}$ by the new variable z, then $\delta_u(C)$ contains D, where D is the set of literals in the ordered clause $\overline{D}$.*

$dup(C, t)$ is defined as follows. If $C = \{L_1, \ldots, L_n\}$ is a clause and t a term occurring in C and suppose t occurs k_1 times in L_1, k_2 times in L_2, etc. Then $dup(C, t) = \overline{C}$ is an ordered clause consisting of 2^{k_1} copies of L_1, 2^{k_2} copies of L_2, ... and 2^{k_n} copies of L_n. Note that if some $L \in C$ does not contain the term t, then $\overline{C}$ contains L $2^0 = 1$ times, as it should.

iii. *For every variable x in C and every non-empty proper subset of the set of occurrences of x in $\overline{C} = dup(C, x)$, if $\overline{D}$ is the ordered clause obtained by replacing those occurrences of x in $\overline{C}$ by the new variable z, then $\delta_u(C)$ contains D.*

(b) **Remove a literal from the body of a clause:** *If $C = D \cup \{L\}$ and L is a most general literal with respect to D, then $\delta_u(C)$ contains D.*

2. *By bounding the language:* If we want to retain all three ideal properties, it seems that the only possibility is to restrict the search space. There exist ideal refinement operators for reduced finite clausal languages, ordered by subsumption. The fact that there always exists an ideal refinement operator for finite sets is mainly of theoretical interest. Thus in practice, we usually prefer more constructive-though possibly improper-refinement operators over such very elaborate ideal operators.

3. *Dropping the requirement of completeness:* Refinement operators are used very often in ILP systems, for instance in MIS [Sha81b], SIM [LD90, Lin92], FOIL [Qui90, QC93], CLAUDIEN [DB93], LINUS [LD94], and Progol [Mug95]. For reasons of efficiency, those operators are usually less general than the ones discussed here, and often incomplete. Nevertheless, the complete operators defined here form a good starting point for the construction of practical refinement operators.

   An approximation to the ideal downward refinement operator, as adopted in Aleph is:

   (a) Adding a literal drawn from $\bot_i$

   $p(X, Y) \leftarrow q(X)$ becomes $p(X, Y) \leftarrow q(X), r(Y)$

   (b) Equating two variables of the same type

   $p(X, Y) \leftarrow q(X)$ becomes $p(X, X) \leftarrow q(X)$

   (c) Instantiate a variable with a general functional term or constant

   $p(X, Y) \leftarrow q(X)$ becomes $p(3, Y) \leftarrow q(3)$

Optimal cover-refinement operators do not exist for clausal languages ordered by subsumption.

2.7.4 Refinement Operators on Theories

Definition 11 *Let $\mathcal{C}$ be a clausal language, containing only a finite number of constants, function symbols, and predicate symbols. Let $\mathcal{S}$ be the set of finite subsets of $\mathcal{C}$. The downward refinement operator $\rho_{\mathcal{I}}$ for $< \mathcal{S}, \models >$ is defined as follows:*

1. $(\Sigma \cup \{R \mid R$ *is a resolvent of* $C_1, C_2 \in \Sigma\}) \in \rho_{\mathcal{I}}(\Sigma)$.
2. *If* $\Sigma = \{C_1, \ldots, C_n\}$, *then* $(\Sigma \cup \rho_{\mathcal{I}}(C_i)) \in \rho_{\mathcal{I}}(\Sigma)$, *for each* $1 \leq i \leq n$.
3. *If* $\Sigma = \{C_1, \ldots, C_n\}$, *then* $(\Sigma \setminus \{C_i\}) \in \rho_{\mathcal{I}}(\Sigma)$, *for each* $1 \leq i \leq n$.

Note that every theory in $\rho_{\mathcal{I}}(\Sigma)$ that is specified by one of the first two items in the definition of $\rho_{\mathcal{I}}$ is logically equivalent to Σ. This shows that $\rho_{\mathcal{I}}$ is not proper. The completeness of $\rho_{\mathcal{I}}$ follows from the Subsumption Theorem, which tells us that logical implication can be implemented by a combination of resolution and subsumption.

2.8 Inverse Resolution

Since induction can be seen as the inverse of deduction, and resolution is our main tool for deduction, using inverse resolution for induction seems a sensible idea. Deduction moves from the general rules to the special case, while induction intends to find the general rules from special cases (examples). Muggleton and Buntine [MB88] introduced inverse resolution as a tool for induction. Their paper was followed by a wave of interest and research into the properties of inverse resolution [Wir89, HS91, Mug91b, Mug92b, Mug92c, RP89, RP90, Rou92, NF91, Ide93c, Ide92, Ide93b, Ide93a, LN92, SADB92, Tay93, SA93, BG93]. Inverting resolution is nowadays still a prominent *generalization operator* for bottom-up approaches to ILP. However, the theoretical foundation of this idea needs much more investigation. Moreover, in the application of inverse resolution, many indeterminacies arise: many different choices of literals, clauses and substitutions lay open. Accordingly, inverse resolution generates a very large search space of possibilities.

Muggleton and Buntine introduced two operators for this: the V-operator and the W-operator.

2.8.1 V-operator

The V-operator, outlined as a non-deterministic[9] algorithm in Figure 2.10, generalizes two given clauses $\{C_1, R\}$ to $\{C_1, C_2\}$, such that R is an instance of a resolvent of C_1 and C_2. The setting for the V-operator is pictorially depicted in Figure 2.11. The goal in inverse resolution is to construct a derivation of a positive example A (usually a ground atom) which hitherto was not implied by

[9]Since, in step 1, the algorithm has to choose one among many different possible θ_1's, which all satisfy $C_1' \subseteq R$.

the theory. Using the algorithm for the V-operator, we can invert one resolution step, for given C_1 and R. By repeatedly applying the V-operator, we are able to invert any SLD-derivation.

```
**INPUT:** Horn clauses $C_1 = L_1 \vee C_1'$ and $R$, where $C_1'\theta_1 \subseteq R$ for some $\theta_1$.
**OUTPUT:** A Horn clause $C_2$, such that $R$ is an instance of a resolvent of $C_1$ and $C_2$.
Choose a substitution $\theta_1$ such that $C_1'\theta_1 \subseteq R$.
Choose an $L_2$ and $C_2'$ such that $L_1\theta_1 = \neg L_2\theta_2$ and $C_2'\theta_2 = R - C_1'\theta_1$, for some $\theta_2$.
**return** $C_2 = L_2 \vee C_2'$.
```

Figure 2.10: The V-operator.

The simplest situation is where $C_1 = L_1$, so where C_1' is empty. Then for a given θ_1, any C_2 and θ_2 with $C_2\theta_2 = \neg L_1\theta_1 \vee R$ will do. Since $L_1\theta_1 \vee R$ is an instance of any of these possible C_2's, it is clear that $C_2 = \neg L_1\theta_1 \vee R$ is the "minimal" of all possible C_2's, for a fixed θ_1.

As an example application of inverse resolution, let $C_1 = P(x) \vee \neg Q(f(x))$, $L_1 = P(x)$, and $R = Q(g(y)) \vee \neg Q(f(g(y)))$. We assume $C_1\theta_1$ and $C_2\theta_2$ do not overlap.

1. Here only one θ_1 is possible, namely $\theta_1 = \{x/g(y)\}$.

2. L_2 and C_2' should be such that, for some θ_2, $L_1\theta_1 = P(g(y)) = \neg L_2\theta_2$ and $R - C_1'\theta_1 = Q(g(y)) = C_2'\theta_2$. Figure 2.12 shows all possible $C_2 = L_2 \vee C_2'$ (unique up to renaming of variables) of two literals, from top to bottom in decreasing order of generality. Small generalization and specialization steps are relevant for the V-operator. In this case, we are often interested in finding a 'minimal' C_2, as in Figure 11.3, where $C_2 = \neg L_1\theta_1 \vee (R - C_1'\theta_1 = \neg P(g(y)) \vee Q(g(y))$ is the minimal choice and can be obtained using the covers relation between clauses in the implication/subsumption

$$\begin{tikzcd}
C_1 = L_1 \vee C_1' \arrow[dr, "\mu"'] & & C_2 = L_2 \vee C_2' \arrow[dl, "\mu"] \\
& R' = C_1'\mu \vee C_2'\mu \arrow[d, "\gamma"] & \\
& R = C_1'\theta_1 \vee C_2'\theta_2 &
\end{tikzcd}$$

Figure 2.11: The setting for the V-operator.

$$\begin{tikzcd}
& \neg P(z) \vee Q(y) \arrow[dl] \arrow[dr] & \\
\neg P(g(z)) \vee Q(y) \arrow[dr] & & \neg P(z) \vee Q(g(y)) \arrow[dl] \\
& \neg P(g(z)) \vee Q(g(y)) \arrow[d] & \\
& \neg P(g(y)) \vee Q(g(y)) &
\end{tikzcd}$$

Figure 2.12: C_2 derived using V-operator.

order studied in Section 2.1.2. The other C_2's can then be found by taking small generalization steps starting from the minimal C_2.

Note that for some C_2, R itself is not a resolvent of C_1 and C_2. For instance, if we let $C_2 = \neg P(z) \vee Q(y)$, then the resolvent of C_1 and C_2 is $\neg Q(f(z)) \vee Q(y)$, of which R is an instance.

We sometimes have to duplicate some literals in R before applying the V-operator, in order to be able to find the desired parent clauses, such that $C_1^{'}\theta_1$ and $C_2^{'}\theta_2$ overlap.

Note that there are many indeterminacies here, which make an unrestricted search through all possible invertible derivations very inefficient. Within the V-operator itself, many different choices for θ_1, L_2 and $C_2^{'}$ are possible. And even before we can use the V-operator, we have to decide which clause from the old theory or the background knowledge to use as C_1, and which literals to duplicate in R. Thus the total number of possibilities may become very large sometimes, which can make application of inverse resolution very inefficient.

In Figures 2.13 and 2.14, we contrast linear derivation against inverse linear derivation.

2.8.2 Predicate Invention: W-operator

One of the problems inductive learning algorithms have to face, is the fact that it is sometimes necessary to invent new predicates. For instanCe; suppose we want our algorithm to induce clauses from examples about family life. It would be very unfortunate if the system did not possess a predicate for the concept of 'parent'. If we have not given such a predicate to the system in advance, the system should be able to invent this predicate for itself. If we examine the V-operator carefully, it is clear that this operator cannot invent new predicates: all predicates appearing in any of the possible C_2 that we might construct already appear in C_1 or R. However, by putting two V-settings side-by-side, we get a W-shape. The W-operator combines two V-operators: it generalizes two given clauses $\{R_1, R_2\}$ to $\{C_1, C_2, C_3\}$, such that R_1 is an instance of a resolvent of

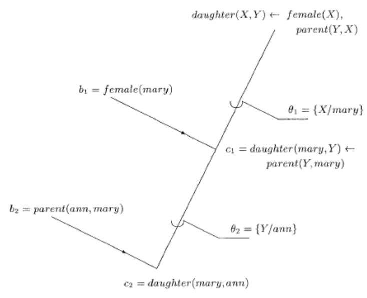

$daughter(X,Y) \leftarrow female(X), parent(Y,X)$
$b_1 = female(mary)$
$\theta_1 = \{X/mary\}$
$c_1 = daughter(mary,Y) \leftarrow parent(Y,mary)$
$b_2 = parent(ann,mary)$
$\theta_2 = \{Y/ann\}$
$c_2 = daughter(mary,ann)$

Figure 2.13: The linear derivation tree for $e_2 = Daughter(mary, ann)$ from background knowledge $\mathcal{B} = \{b_1, b_2\}$ where $b_1 = Female(mary)$ and $b_2 = Parent(ann, mary)$ and from a hypothesis $H = \{c\}$ where $c = Daughter(x, y) \leftarrow Female(x), Parent(y, x)$.

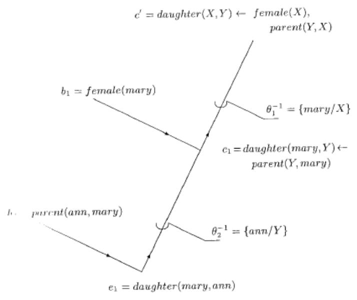

$c' = daughter(X,Y) \leftarrow female(X), parent(Y,X)$
$b_1 = female(mary)$
$\theta_1^{-1} = \{mary/X\}$
$c_1 = daughter(mary,Y) \leftarrow parent(Y,mary)$
$parent(ann,mary)$
$\theta_2^{-1} = \{ann/Y\}$
$e_1 = daughter(mary,ann)$

Figure 2.14: The inverse linear derivation tree for $H = \{c\}$ where $c = Daughter(x, y) \leftarrow Female(x), Parent(y, x)$ from background knowledge $\mathcal{B} = \{b_1, b_2\}$ where $b_1 = Female(mary)$ and $b_2 = Parent(ann, mary)$ and from an example $e_2 = Daughter(mary, ann)$.

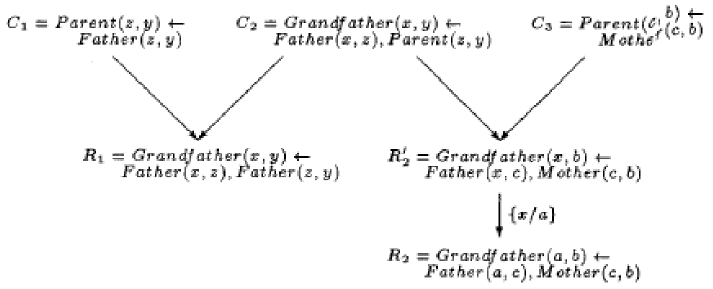

$C_1 = Parent(z,y) \leftarrow Father(z,y)$
$C_2 = Grandfather(x,y) \leftarrow Father(x,z), Parent(z,y)$
$C_3 = Parent(c,b) \leftarrow Mother(c,b)$
$R_1 = Grandfather(x,y) \leftarrow Father(x,z), Father(z,y)$
$R_2' = Grandfather(x,b) \leftarrow Father(x,c), Mother(c,b)$
$\{x/a\}$
$R_2 = Grandfather(a,b) \leftarrow Father(a,c), Mother(c,b)$

Figure 2.15: Generalization of $\{R_1, R_2\}$ to $\{C_1, C_2, C_3\}$ by the W-operator.

C_1 and C_2, and R_2 is an instance of a resolvent of C_2 and C_3. In addition, the W-operator is also able to invent new predicates.

We will present an example which shows the idea behiod the W-operator. Suppose we have two Horn clauses $R_1 = Grandfather(x, y) \leftarrow Father(x, z), Father(z, y)$ and $R_2 = Grandfather(a, b) \leftarrow Father(a, c), Mother(c, b)$, and suppose we want to generalize these clauses. The Woperator constructs clauses C_1, C_2, C_3, such that R_1 is an instance of a resolvent of C_1 and C_2, and R_2 is an instance of a resolvent of C_2 and C_3. Thus, the W-operator generalizes $\{R_1, R_2\}$ to $\{C_1, C_2, C_3\}$. In Figure 2.15, we give possible C_1, C_2, C_3 which can serve this purpose. The important point to notice about the figure is that the predicate Parent, which appears in C_1, C_2 and C_3, did not appear in the clauses R_1 and R_2 we started with. Thus in generalizing $\{R_1, R_2\}$ to $\{C_1, C_2, C_3\}$, the W-operator has itself introduced a new predicate. The invention of this new predicate is quite useful, since it allows us to write out the definition of a 'Grandfather' in a very succint way in C_2: x is the grandfather of y, if x is the father of some z, and z is a parent of y. Note that any predicate name may be assigned the role of *Parent* here, including 'old' names such as Grandfather or Mother, since this predicate is resolved away in the two resolution steps anyway.

The general setting for the W-operator is pictured in Figure 2.16. Given R_1 and R_2, the W-operator constructs C_1, C_2, C_3, with the property that R_1 is an instance of a resolvent of C_1 and C_2, and R_2 is an instance of a resolvent of C_2 and C_3. What we want to find, are $C_1 = L_1 \vee C_1^{'}$, $C_2 = L_2 \vee C - 2^{'}$, $C_3 = L_3 \vee C_3^{'}$, θ_1, θ_2, σ_1 and σ_2, such that $L_1\theta_1 = \neg L_2\theta_2$, $L_2\sigma_1 = \neg L_3\sigma_2$, $R_1 = C_1^{'}\theta_1 \vee C_2^{'}\theta_2$ and $R_2 = C_2^{'}\sigma_1 \vee C_3^{'}\sigma_2$. Thus L_1 and L_2 are resolved upon in deriving R_1, while L_2 and L_3 are resolved upon in deriving R_2. μ is an mgu for L_1 and $\neg L_2$, and ν is an mgu for $\neg L_2$ and L_3. Hence L_1 and L_3 must either be both positive, or both negative. Note that L_1, L_2, L_3 do not appear in R_1 and R_2, which gives the opportunity for inventing a new predicate.

The idea behind the construction of C1, C2 and C3 using the W-operator is sketched below:

1. Given R_1 and R_2, we first try to find a $C_2^{'}$ such that $C_2^{'}\theta_2 \subseteq R_1$ and

$$\begin{tikzcd}
C_1 = L_1 \vee C_1' \arrow[dr, "\mu"'] & & C_2 = L_2 \vee C_2' \arrow[dl, "\mu"] \arrow[dr, "\nu"'] & & C_3 = L_3 \vee C_3' \arrow[dl, "\nu"] \\
& R_1' = C_1'\mu \vee C_2'\mu \arrow[d, "\gamma_1"] & & R_2' = C_2'\nu \vee C_3'\nu \arrow[d, "\gamma_2"] & \\
& R_1 = C_1'\theta_1 \vee C_2'\theta_2 & & R_2 = C_2'\sigma_1 \vee C_3'\sigma_2 &
\end{tikzcd}$$

Figure 2.16: The setting for the W-operator.

$C_2'\sigma_1 \subseteq R_2$, for some θ_2 and σ_1. Let $D_1 = C_2'\theta_2$ and $D_2 = C_2'\sigma_1$. Clearly, many different C_2''s can give the same D_1 and D_2. These C_2''s can be considered as generalizations of $\{D_1, D_2\}$. We would like to begin with a minimal C_2'. This motivates the use of the notion of a 'least generalization' of clauses discussed in Section 2.1.1. If we have found a minimal C_2', the other possible C_2''s can be found by taking small generalization steps, starting from the minimal C_2'. This motivates the use of 'covers' (minimal generalizations or specializations of that clause) and consequently the refinement operator of the clause.

If such a C_2' cannot be found - which means, intuitively, that R_1 and R_2 have 'nothing in common' - we should let C_2' be empty.

2. If we have chosen an appropriate C_2', we can complete C_2 by choosing also L_2. In principle, **any** L_2 will do.

3. Once we have decided which clause to take as C_2, then a C_1 and C_3 can be *found independently*. C_1 can be constructed by the V-operator from C_2 and R_1, and C_3 can be constructed by the V-operator from C_2 and R_2.

Consider Figure 2.15 again. Given R_1 and R_2, how did we find C_1, C_2 and C_3? First we note that $(Grandfather(x,y) \vee \neg Father(x,z))\epsilon \subseteq R_1$, and $(Grandfather(x,y) \vee \neg Father(x,z))\{x/a, y/b, z/c\} \subseteq R_2$. Hence $C_2' = Grandfather(x,y) \vee Father(x,z)$ is an appropriate choice. Secondly, we have to choose L_2. Let us say we take $L_2 = \neg Parent(z,y)$. This gives $C_2 = Grandfather(x,y) \leftarrow Father(x,z), Parent(z,y)$. Thirdly, the V-operator can find $C_1 = Parent(z,y) \leftarrow Father(z,y)$ from C_2 and R_1. Similarly, it can construct the clause $C_3 = Parent(c,b) \leftarrow Mother(c,b)$ from C_2 and R_2. Thus the W-operator generalizes $\{R_1, R_2\}$ to $\{C_1, C_2, C_3\}$.

2.9 Summary

In this chapter, we discussed the following generalization operations:

1. Relative least general generalization (lgg/lub) with respect to subsumption (LGRS) as well as implication order (LGRI). This is utilized by systems such as GOLEM [Muggleton and Feng 92]. The progressive construction of lubs for the subsumption order, starting with terms and culminating in clauses is illustrated in Table 2.1.

   GOLEM is a bottom-up, non-interactive, batch single-predicate learner, which employs LRGS/

2. The upward refinement operator δ_u.

3. Inverse resolution in terms of the V and W operators. This is utilized in ILP systems such as CIGOL (W-operator) [Muggleton and Buntine 88], MARVIN (V-operator) [Sammu and Banerji 86] and ITOU (V-operator) [Rouveirol 92].

   CIGOL is a bottom-up, interactice, incremental, multiple predicate learner that uses the W-operator for predicate invention. The user is asked whether the induced clauses are true in the interpretation intended by the user and how invented predicates should be named.

The following specialization techniques were also discussed:

1. Relative greatest specialization with respect to subsumption as well as implication order. This is rarely used in practice.

2. The downward refinement operator $\rho_{\mathcal{L}}$. This is utilized in ILP systems such as MIS (model inference system) [Shapiro 83], FOIL [Quinlan and Cameron-Jone 93] and its successors mFOIL [Dzeroski 91], CLAUDIEN [L. De Raedt and Bruynooghe 93], PROGOL [Muggleton 95], MOBAL [Morik, Wrobel, Kietz and Emde 93], RDT [J-U Kietz and Wrobel 92], FOCL [Brunk and Pazzani 91], MARKUS [Grobelnik 92] and MPL [De Raedt et al 1993].

   MIS is a top-down, interactive, incremental, multiple-predicate learner, restricted to Horn clauses. FOIL is a top-down, non-interactive, batch single-predicate learner, upgrading Quinlan's earlier decision tree learner ID3 [Quinlan 86] which learns function free normal programs using the (set) 'covering' approach. CLAUDIEN is a top-down, non-interative batch learner which uses Herbrand interpretations as (positive only) examples. CLAUDIEN is one of the very few systems that induces full, rather than definite or normal program clauses. PROGOL is a top-down, non-interactive, batch, multi-predicate learner that learns clauses using an A^*-like heuristic algorithm to search top-down through a refinement graph, while restricting attention to clauses that subsume some bottom-clause $\bot_d(\mathcal{B}, e)$ (*c.f.* page 120). RDT is top-down, non-interactive, batch, multi-predicate learner that learns function-free normal programs, using a set of ground literals as background knowledge (or reduced to that form as in GOLEM) and where language bias is introduced into the refinement operators using a rule schema. A *rule schema* is a clause with predicate-variables instead of ordinary predicate symbols Only clauses that can be obtained by instantiating the predicate-variables in one of the given rule schemas to ordinary predicate symbols may be used in the theory. It also uses a *predicate topology* to further restrict the search for what predicate symbols can be added to the body, given a predicate symbol in the head. MOBAL makes use of RDT as one of its components.

3. There is another specialization technique called *unfolding*, which was not discussed so far. SPECTRE [Bostrom 95], IMPUT [Alexin, Gyimothy and Bostrom 96] and JIGSAW [Ade and Bostron 95] use unfolding. SPECTRE is a top-down, non-interactive, batch, single-predicate learner.

Systems such as CLINT make use of a hybrid of top-down (unfolding) and bottom-up (abduction) approaches.

Lub	Definition	Examples
lub of terms $lub(t_1, t_2)$	1. $lub(t, t) = t$, 2. $lub(f(s1, \ldots, s_n), f(t_1, \ldots, t_n)) = f(lub(s_1, t_1), \ldots, lub(s_n, t_n))$, 3. $lub(f(s_1, \ldots, s_m), g(t_1, \ldots, t_n)) = V$, where $f \neq g$, and V is a variable which represents $lub(f(s_1, \ldots, s_m), g(t_1, \ldots, t_n))$, 4. $lub(s, t) = V$, where $s \neq t$ and at least one of s and t is a variable; in this case, V is a variable which represents $lub(s, t)$.	• $lub([a, b, c], [a, c, d]) = [a, X, Y]$. • $lub(f(a, a), f(b, b)) = f(lub(a, b), lub(a, b)) = f(V, V)$ where V stands for $lub(a, b)$. • When computing lggs one must be careful to use the same variable for multiple occurrences of the lubs of subterms, *i.e.*, $lub(a, b)$ in this example. This holds for lubs of terms, atoms and clauses alike.
lub of atoms $lub(\mathbf{a}_1, \mathbf{a}_2)$	1. $lub(P(s_1, \ldots, s_n), P(t_1, \ldots, t_n)) = P(lub(s_1, t_1), \ldots, lub(s_n, t_n))$, if atoms have the same predicate symbol P, 2. $lub(P(s_1, \ldots, s_m), Q(t_1, \ldots, t_n))$ is undefined if $P \neq Q$.	
lub of literals $lub(\mathbf{l}_1, \mathbf{l}_2)$	1. if $\mathbf{l}_1$ and $\mathbf{l}_2$ are atoms, then $lub(\mathbf{l}_1, \mathbf{l}_2)$ is computed as defined above, 2. if both $\mathbf{l}_1$ and $\mathbf{l}_2$ are negative literals, $\mathbf{l}_1 = \overline{\mathbf{a}_1}$, $\mathbf{l}_2 = \overline{\mathbf{a}_2}$, then $lub(\mathbf{l}_1, \mathbf{l}_2) = lub(\overline{\mathbf{a}_1}, \overline{\mathbf{a}_2}) = \overline{lub(\mathbf{a}_1, \mathbf{a}_2)}$, 3. if $\mathbf{l}_1$ is a positive and $\mathbf{l}_2$ is a negative literal, or vice versa, $lub(\mathbf{l}_1, \mathbf{l}_2)$ is undefined.	• $lub(Parent(ann, mary), Parent(ann, tom)) = Parent(ann, X)$. • $lub(Parent(ann, mary), \overline{Parent(ann, tom)}) = undefined$. • $lub(Parent(ann, X), Daughter(mary, ann)) = undefined$.
lub of clauses $lub(C_1, C_2)$	1. Let $C_1 = \{\mathbf{l}_1, \ldots, \mathbf{l}_n\}$ and $C_2 = \{\mathbf{k}_1, \ldots, \mathbf{k}_m\}$. 2. Then $lub(C_1, C_2) = \{\mathbf{l}_{ij} = lub(\mathbf{l}_i, \mathbf{k}_j) \mid \mathbf{l}_i \in C_1, \mathbf{k}_j \in C_2 \ and \ lub(\mathbf{l}_i, \mathbf{k}_j) \ \ is \ defined \ \}$.	• If $C_1 = Daughter(mary, ann) \leftarrow Female(mary), Parent(ann, mary)$ and $C_2 = Daughter(eve, tom) \leftarrow Female(eve), Parent(tom, eve)$, then $lub(C_1, C_2) = Daughter(X, Y) \leftarrow Female(X), Parent(Y, X)$, where X stands for $lub(mary, eve)$ and Y stands for $lub(ann, tom)$.
$rlgg(\mathbf{a}_1, \mathbf{a}_2)$	$lub(\mathbf{a}_1 \leftarrow \mathcal{B}, \mathbf{a}_2 \leftarrow \mathcal{B})$	$rlgg(Daughter(mary, ann), Daughter(eve, tom) = Daughter(X, Y) \leftarrow Female(X), Parent(Y, X)$ where $\mathcal{B}$ denotes the conjunction of the literals $Parent(ann, mary)$, $Parent(ann, tom)$, $Parent(tom, eve)$, $Parent(tom, ian)$, $Female(ann)$, $Female(mary)$, $Female(eve)$.

Table 2.1: Table showing progressive definitions of lubs, starting with terms and culminating in rlgg between clauses.

Chapter 3

Searching on Graphs

Consider the following problem:

Cab Driver Problem A cab driver needs to find his way in a city from one point (A) to another (B). Some routes are blocked in gray (probably because they are under construction). The task is to find the path(s) from (A) to (B). Figure 3.1 shows an instance of the problem..

Figure 3.2 shows the map of the united states. Suppose you are set to the following task

Map coloring problem Color the map such that states that share a boundary are colored differently..

A simple minded approach to solve this problem is to keep trying color combinations, facing dead ends, back tracking, etc. The program could run for a very very long time (estimated to be a lower bound of 10^{10} years) before it finds a suitable coloring. On the other hand, we could try another approach which will perform the task much faster and which we will discuss subsequently.

The second problem is actually isomorphic to the first problem and to problems of resource allocation in general. So if you want to allocate aeroplanes to routes, people to assignments, scarce resouces to lots of tasks, the approaches we will show in this chapter will find use. We will first address search, then constraints and finally will bring them together.

3.1 Search

Consider the map of routes as in Figure 3.3. The numbers on the edges are distances. Consider the problem of organizing search for finding paths from S (start) to G (goal).

We humans can see the map geometrically and perhaps guess the answer. But the program sees the map only as a bunch of points and their distances. We will use the map as a template for the computer at every instance of time.

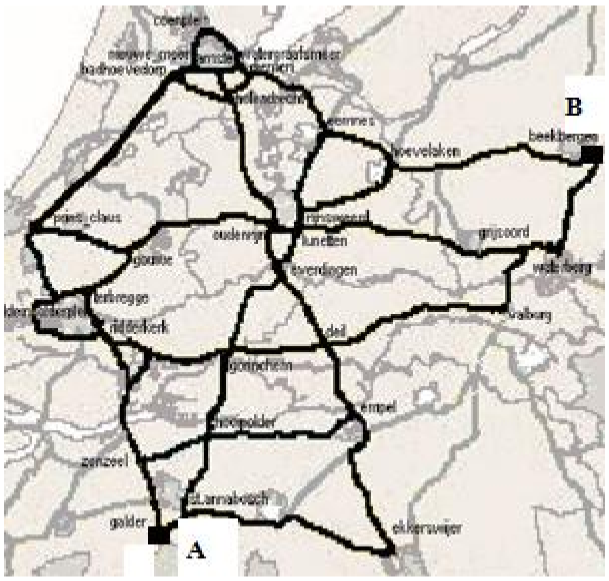


Figure 3.1: Routes in a city. The problem is to find the shortest path from point (A) to point (B).

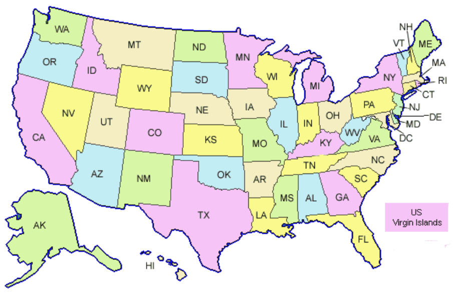


Figure 3.2: The map of the united states of America. The task is to color the map such that adjacent states have different colors.

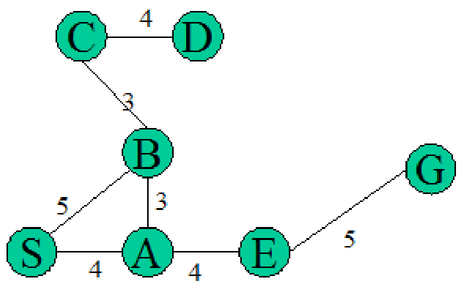


Figure 3.3: Map of routes. The numbers on the edges are distances.

At the beginning, the computer knows only about S. The search algorithm explores the possibilities for the next step originating from s. The possible next steps are A and B (thus excluding G). Further, from A, we could go to B or E. From B, we could go to A or C. From A, we could also go back to S. But we will never bother biting our own tail in this fasion; that is, we will never make a loop. Going ahead, from B, we could only go to C. From C, we can progress only to D and thereafter we are stuck on that path. From A, we could make a move to E and them from E to G. Figure 3.4 shows the exhaustive tree of possible paths (that do not bite their own tail) through this map. The process of finding all the paths is called the *British Museum Algorithm*[1].

But the British Museum algorithm can be very expensive. In fact, some of these searches can be exponential. If you had to look through a tree of chess moves for example, in the beginning it is essentially exponential, which is a bad news since it will imply 10^{10} years or so. We need a more organized approach for searching through these graphs. There exist many such organized methods. Some are better than others, depending on the graph. For example, a depth first search may be good for one problem but horrible for another problem. Words like *depth first search*, *breadth first search*, *beam search*, *A^* search*, *etc.*, form the vocabulary of the search problem in Artificial Intelligence.

The representation we use will define the constraints (for example, the representation of the routes in Figure 3.3 defines the notion of proximity between nodes and also defines constraints on what sequences of vertices correspond to valid paths.

3.1.1 Depth First Search

This algorithm[2] boils down to the following method: Starting at the source, every time you get a choice for the next step, choose a next step and go ahead. We will have the convention that when we forge ahead, we will take the first

[1]The British Museums are considered some of the largest in the world.

[2]Fortunately, the names given to these algorithms are representative of the algorithms actually do.

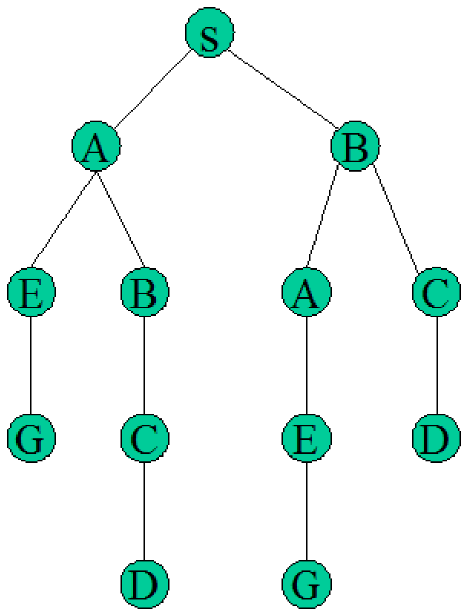


Figure 3.4: The tree of all possible paths from S to G through the map in Figure 3.3.

choice. Thus, a depth first search on the map in Figure 3.3 will tread along the path in Figure 3.5. In practice, what is used is a combination of depth first search and backup. What this means is that when a dead end is hit, the method goes back to the last state. In practice, this method could yield very complicated solutions.

3.1.2 Breadth First Search (BFS)

The way BFS organizes its examination of the tree is layer by layer; the algorithm first explores one layer of solutions (A or B), then forges ahead to another layer (B or E or A or C) and so on. Figure 3.6 illustrates the BFS traversal for the map in Figure 3.3. In practice, this search could expend a lot of time in useless parts of the graph. But it yields the shortest path in terms of number of number of streets covered.

3.1.3 Hill Climbing

Both BFS and DFS are completely uninformed about geography; neither information about distance nor direction is exploited. But often it helps if you have outside information about how good a place is to be in. For instance, in the map in Figure 3.3, between E and D, it is much better to be in E because that gets you closer to the goal. It makes less sense in general to head off to the right

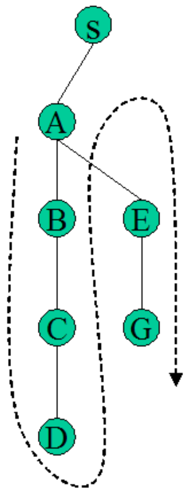


Figure 3.5: The DFS tree from S to G through the map in Figure 3.3.

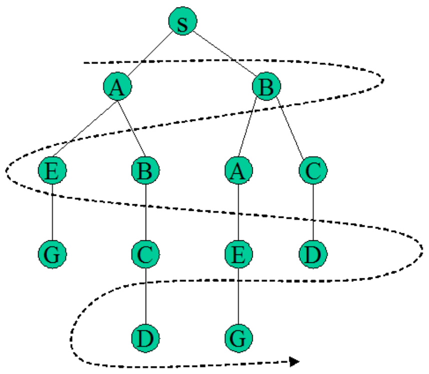


Figure 3.6: The BFS tree from S to G through the map in Figure 3.3.

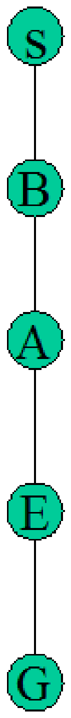


Figure 3.7: The choices based on hill climbing for the route from S to G through the map in Figure 3.3.

if you know that the goal is on the left. It is heuristically good to be closer to the goal, though sometimes it may turn out not to be a good idea. So we could make use of heuristic information of this kind. And this forms the idea behind hill climbing. It is an idea drafted on top of depth first search. When initially at S, you could move to A or B. When you look at A and B, you that one of them is closer to the goal G and you choose that for the next move. And B happens to be closer to G, which you pick for the next move; but this turns out to be a bad idea as we will see. Nevertheless, it looks good from the point of view of the short-sighted hill climbing algorithm. From B, the possible choices are A and C. And A is a natural choice for hill-climbing, owing to its proximity to G. From A onwards, the choices are straightforward - you move to E and then to G. Figure 3.7 shows the route from S to G as chalked out by hill climbing.

Hill climbing always is greedy because it plans only for one step at a time. The method requires a metric such as distance from the goal.

3.1.4 Beam Search

We drafted hill climbing on top of depth first search. Is there a similar thing we could do with breadth first search? And the answer is ‘yes’. And this approach

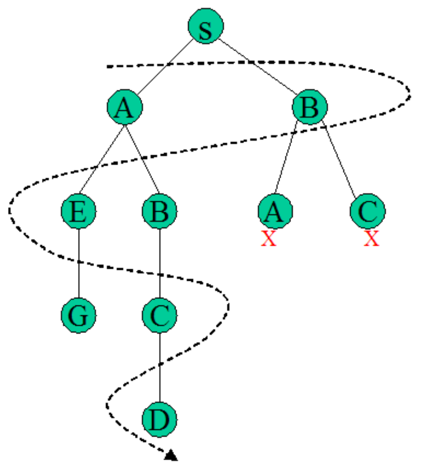


Figure 3.8: The choices based on beam search for the route from S to G through the map in Figure 3.3.

is called *beam search* The problem with breadth first search is that it tends to be exponential. But what we could do to work around is to throw away at every level of BFS, all but the 'most promising' of the paths so far. The most promising step is defined as that step which will get us 'closest' to the goal. The only difference from the hill climbing approach is that beam search does not pick up just one path, it picks up some fixed number of paths to carry down. Let us try beam search on the map in Figure 3.3. For this simple example, let us keep track of two paths at every level. We will refer to the BFS plan in Figure 3.6. From S, we could move to A or B and we keep track of both possibilities. Further, from A, there are two possibilities, *viz.*, B and E, while from B there are two possibilities in the form of A and C. Of the four paths, which two should we retain? The two best (in terms of the heuristic measure of how far we are from the goal) are B and E. Carrying forward from these points, we arrive at G in a straight-forward manner as shown in Figure 3.8.

While the vanila breadth first search is exponential in the number of levels, beam search has a fixed width and is therefore a constant in terms of number of levels. Beam search is however not guranteed to find a solution (though the original BFS is guaranteed to find one). This is a price we pay for saving on time. However, the idea of backing up can be employed here; we could back up to the last unexplored path and try from there.

3.2 Optimal Search

For optimal search, we will start with the brute force method which is exponential and then slap heuristics on top to reduce the amount of work, while still assuring success.

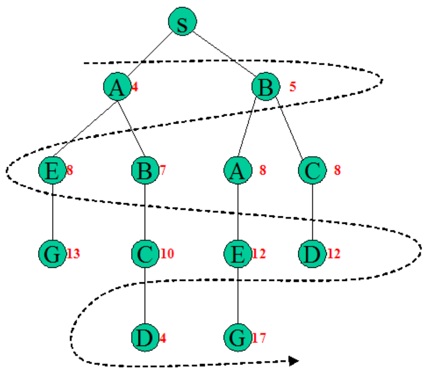


Figure 3.9: The tree of all possible paths from S to G through the map in Figure 3.3, with cumulative path lengths marked at each node.

3.2.1 Branch and Bound

Suppose the problem is to find the best possible path (in terms of distances) between S and G in the map as in Figure 3.3. An oracle suggests that the path $SAEG$ is the shortest one (its length is 13). Can we determine if $SAEG$ is indeed the shortest path? One method to answer this question is to verify if every other path is at least that long.

From S, we could go to A or B. The cumulative path length to B is 5. From B, you could either go to A or C. The cumulative path length upto A or C is 8. At this point, from A, you could go to E while from C you could move to D, with cumulative path lengths of 13 each and so on. Figure 3.9 shows the tree of possible paths from S to G, with cumulative path length (from S) marked at each node. It is evident from the figure that all paths to G have length greater than or equal to 13. We can therefore conclude that the shortest path to G from S is $SAEG$ and has length 13.

Most often we do not have any oracle suggesting the best path. So we have to think how to work without any oracle telling us what the best path is. One way is to find a path to the goal by some search technique (DFS or beam search) and use that as a reference (bound) to check if every other path is longer than that. Of course, our first search may not yield the best path, and hence we might have to change our mind about what the best path (bound) is as we keep trying to verify that the best one we got so far is in fact the best one by extending every other path to be longer than that. The intuition we work with is to always push paths that do not reach the goal until their length is greater than a path that does reach the goal. We might as well work only with shortest paths so far. Eventually, one of those will lead to the goal, with which we will almost be done, because all the other paths will be about that length too, following which we just have to keep pushing all other paths beyond the goal. This method is called branch and bound.

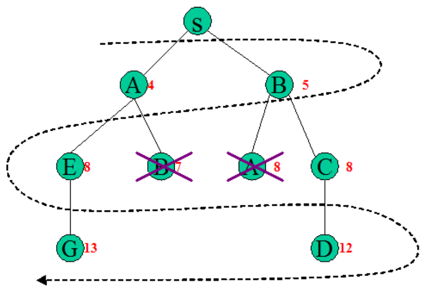


Figure 3.10: The pruned search tree for branch and bound search.

In the example in Figure 3.9, the shortest path upto B from S was of length 5 initially, which used as a bound, could eliminate the path SAB and therefore save the computation of the path $SABCD$. Similarly, the initial path to A, SA of length 4, sets a bound of 4 for the shortest path to A and can thus eliminate the consideration of the path SBA beyond A. Figure 3.10 illustrates the application of branch and bound to prune the BFS search tree of Figure 3.9. Crossed out nodes indicate search paths that are excluded because they yield paths that are longer than bounds (for the corresponding nodes). This crossing out using bounds is a variation on the theme of the dynamic programming principle.

3.2.2 A^* Search

The branch and bound algorithm can also be slapped on top of the hill climbing heuristic or the beam search heuristic. In each of these cases, the bound can be computed as the sum of the accumulated distance and the eucledian distance.

When the branch and bound technique is clubbed with shortest distance heuristic and dynamic programming principle, you get what is traditionally known as A^* search. Understanding A^* search is considered the culmination of optimal search. It is guaranteed to find the best possible path and is generally very fast.

3.3 Constraint Satisfaction

What we have discussed so far is mechanisms for finding the path to the goal (which may not be optimal). Rest of the discussion in this chapter will focus on resource allocation. Research allocation problems involve search. Too often people associate search only with maps. But maps are merely a convenient way[3] to introduce the concept of search and the search problem is not restricted to maps. Very often, search does not involve maps at all. Let us again consider the

[3]Maps involve making a sequence of choices and therefore could involve search amongst the choices.

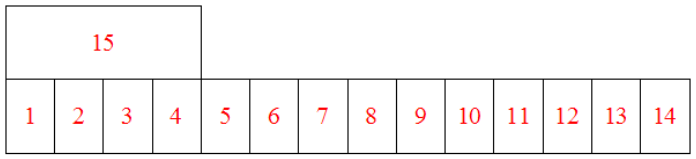


Figure 3.11: The sketch of a simple region for which we try to solve the map coloring problem.

"*Map coloring problem*" (refer to Figure 3.2). It involves searching for sequence of choices. Instead of the large city of USA, let us consider the simpler country of Simplea as in Figure 3.11. Let us say we need to pick a color from the set $\{R, G, B, Y\}$.

Let us say we number the regions arbitrarily. One approach is to try coloring them in the order of increasing numbers, which is a horrible way! This is particularly because the order chosen could be horrible. To color the region, let us start by considering the possible colors that could be assigned to each region. State number 1 could be assigned any of R, B, G or Y. Suppose we take a depth first search approach for this coloring job. Now we need to color the region number 2. Since we adopt depth first search, let us say we pick the color of R for region 1. And as we go from region to region, we keep rotating the set of colors.

The prescription for DFS is that you keep plunging head until you hit a dead end where you cannot do anything. But the problem is that we cannot infer we have hit a dead end till we assign colors to all 15 regions, to realise that the following constraint has been violated: *no two adjacent regions should have the same color*. And even if we backup a step or two, we will have to explore all 15 regions to infer if we have reached a dead end. At this rate, the projected time for successful coloring of the states in a US map is 10^{10} years[4]! That is no way to perform search. The culprit is the particular order in which we chose to color the regions. Figure 3.12 shows an example instance of random assignment of colors that leads to a dead-end (with no suitable color left for region number 15). We introduce the idea of *constraint checking* precisely to address this problem. It is essentially a (DFS) tree trimming method.

First of all, we will need some terminology.

1. *Variables (V):* V is a set of variables. Variables names will be referred to in upper case, whereas their specific instatiations will be referred to in lower case.

2. *Domain (D):* The domain D is the bag of values that the variables can take on. D_i is the domain of the $V_i \in V$. A good example of a domain

[4]The United states has 48 contiguous states. If the number of colors is 4, the branching factor of DFS will be 4 and the height will be 48. Thus, the number of computations will be 4^{48} or 2^{96} which is of the order of 10^{27} and since the length of a year is the order of 10^7, it will take in the order of 10^{11} years assuming that a computation is performed in a nano second.

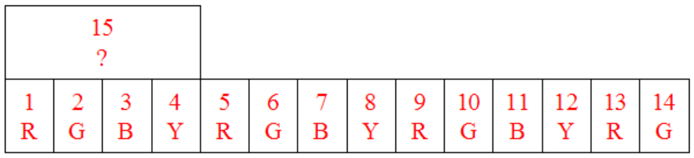


Figure 3.12: A sequence of bad choices for the map coloring problem.

would be $\{R, B, G, Y\}$. In the map coloring problem, all variables have the same domain.

3. *Constraints (C):* A constraint is a boolean function of two variables. C_{ij} represents the constraint between variables X_i and X_j. $C_{ij}(x_i, x_j) = true$ *iff*, the constraint is satisfied for the configuration $X_i = x_i, x_i \in D_i$ and $X_j = x_j, x_j \in D_j$. Else, $C_{ij}(x_i, x_j) = false$. The form of the constraint function could vary from problem to problem. An example of a constraint is: *no two adjacent regions should not have the same color.* Note that there need not be a C_{ij} for all i and j. For instance, in the map coloring problem, the only constraints are those involving adjacent regions, such as region 15 with the regions 1, 2, 3 and 4.

Suppose we are trying to explore if region number $i = 15$ is going to object to any coloring scheme. First we define the constraint $C_j, i = 15, j \in \{1, 2, 3, 4\}$ as follows:

$C_{ij}(x, y), i = 15, j \in \{1, 2, 3, 4\}$ *is true iff* $x \neq y$.

We can state the check as in Figure 3.13.

```
**for** $x \in D_i$ $(i = 15)$ **do**
  **for** All constraints $C_ij$ **do**
    Set $X_i = x$ *iff* $\exists y \in D_j$ such that $C_{ij}(x, y) = true$
  **end for**
**end for**
```

Figure 3.13: The algorithm for constraint checking

3.3.1 Algorithms for map coloring

When assigning a color to each region, we could do different types of checks.

1. *Check all:* At one end of the spectrum, while assigning a color to a region, we check constraint satisfaction with respect to all regions (no matter how far they are from the region under consideration). This is an extreme case and potentially, we could consider all variants of what to check and what not to check. The check is only a speedup mechanism. It neither ensures nor prevents a solution. We could get a solution (if there is one) using depth first search and no search, thought it might take 10^{10} years. If we

want to have minimum number of checks such that they are really helpful, our method for deciding what to check will be checks on regions that are in the immediate vicinity of the last region that had a color assigned.

2. *Check Neighbors:* Check for constraint satisfaction, only all regions in the immediate neigborhood of the last region that had a color assigned. This is at another end of the spectrum.

3. *Check Neighbors of Neighbors:* Check for constraint satisfaction, only all regions in the immediate neigborhood of the last region that had a color assigned as well as well as regions in the immediate neighborhood of the neighbors.

4. *Check Neighbors of 'unique' neighbors:* Check for constraint satisfaction, only all regions in the immediate neigborhood of the last region that had a color assigned as well as well as regions in the immediate neighborhood of the 'unique' neighbors. A neighbor is 'unique', if the constraint set for the region has been reduced to a single element (which means that it has a very tight constraint).

We will evaluate the above approaches on two fronts, *viz.*,

1. *Number of assignments performed:* An *assignment* corresponds to putting a color on a region. If we are very lucky, we might be able to color the map of the United states with just 48 assignments (that is we never had to backup). Constraint checking will enable the algorithm realise that it will soon hit a dead end and will have to back up. In practical situations, we could expect a bit of backup and also expect the constraints to show you some dead ends. So it could happen that the constraint checking reduces the number of assignments to some number slightly above the ideal (say 52).

2. *Number of checks performed:* A *check* corresponds to determining which color could be assigned to a region based on the color assigned so far to another regions.

A simple experiment for coloring the Unites States reveals the statistics in Table 3.1.

The message we can extract from Table 3.1 is that some of the constraint checking is essential, else it takes 10^{10} years. We can conclude that full propogation is probably not a worthy effort. Because full propogation, relative to propogation through unique values yielded the same number of assignments but relatively fewer checks. When we compare heuristic 2 against heuristic 4, we find that there is a tradeoff between number of assignments and number of checks. Checking all is simply a waste of time. The net result is that people tend to invoke either of heuristics 2 and 4. From the point of view of programming simplicity, one might just use heuristic number 2.

SrNo	Constraint Sat Heuristic	Assignments	Checks
1	Check all	54	7324
2	Check neighbors	78	417
3 of neighbors	Check neighbors	54	2806
4 of 'unique' neighbors	Check neighbors	54	894

Table 3.1: Number of assignments and checks for each of the constraint satisfaction heuristics for the problem of coloring the map of the USA using 4 colors.

The computational complexity of the discussed methods is still exponential[5]. But usually, the constraints try to suppress the exponential nature.

3.3.2 Resource Scheduling Problem

Let us forget about maps for a while and talk about airplanes. An upstart airline is trying to figure out how many airplanes they need. They have a schedule outlining when and where they need to fly planes. And the task is to figure out the minimum number of airplanes that need to be purchased. For every airplane saved, let us say the reward is saving on half the cost of the airplane.

$F_1, F_2, \ldots, F_n$ are the flights. Table 3.2 shows the schedule for the flights.

Flight No.	From	To	Dept. Time	Arr. Time
F_1	Boston	LGA	10:30	11:30
F_2	Boston	LGA	11:30	12:30
F_3	Boston	LGA	12:30	13:30
F_4	Boston	LGA	13:30	14:30
F_4	Boston	LAX	14:30	15:30
F_6	...	...	...	
...	...	...	...	
F_15	Boston	LAX	11:00	15:30

Table 3.2: Number of assignments and checks for each of the constraint satisfaction heuristics for the problem of coloring the map of the USA using 4 colors.

To fly this schedule, we have some number $m = 4$ of airplanes: P_1, P_2, P_3, P_4.

[5]Note that these are NP complete problems. Any polynomial time algorithm will fetch field medals.

Of course, we do not want to fly any plane empty (dead head).

The assignment $P_1 \rightarrow F_1$, $P_2 \rightarrow F_2$, $P_3 \rightarrow F_3$ and $P_4 \rightarrow F_4$ will leave no airplane for flight F_5. But if we could fly P_1 back from New York (LGA), the same plane could be used for flight F_4, thus sparing P_4 for F_5. We can draw the correspondence between this problem and the map coloring problem; the 4 airplanes correspond to 4 colors while the flights correspond to regions. The task is to assign planes (colors) to flights (regions), at all times honoring constraints between the flights. The constraints are slightly different from the ones we had in the US map. The constraint is that no airplane can be on two routes at the same time. This implies that there is a constraint between flights F_1 and F_2. Similarly, there is a constraint between the pairs $< F_2, F_3 >$, $< F_3, F_5 >$, $< F_4, F_5 >$, $< F_1, F_5 >$, $< F_2, F_5 >$ and $< F_3, F_5 >$. We assume that the turn around duration for F_1 (which is of a one hour duration) will end by 14:30 hours, which sounds reasonable. Thus, there is no constraint $< F_1, F_4 >$.

You schedule planes the same way you do map coloring. An assignment is tried and constraints for all unassigned flights are checked to ensure that there is at least one airplane that can fly each flight. There is one important difference between the flight scheduling problem and the map coloring problem: we know that we can color maps with 4 colors[6], but we do not know how many airplanes it is going to take to fly a schedule. Hence, we will need to try the flight scheduling problem with different number of airplanes.

Let us say we over-resource the map coloring problem with 7 colors instead of 4. A sample run yields 48 assignments (that is no backup was required) and 274 checks. With 6 colors, you get 48 assignments and 259 checks. If on the extreme end, you used only 3 colors, you could never color Texas.

Frequently the problem is over-constrained and there is no solution with available resources (like say coloring the United States with 2 or 3 colors). In those circumstances, constraints can be turned into preferences so that some regions will not be allowed to be adjacent to regions of the same color. Or we might have to allow some 'dead-hit' flights. And on top of preferences, we could layer beam-search or some other search that tries to minimize the penalty cumulated or maximize the number of constraints that are satisfied.

[6]The 4 color theorem.

Chapter 4

Statistical Relational Learning

One of the central open questions of artificial intelligence is concerned with combining (i) expressive knowledge representation formalisms such as relational and first-order logic with (ii) principled probabilistic and statistical approaches to inference and learning. Why? Here are some reasons:

1. The fields of knowledge representation and inductive logic programming stress the importance of relational and logical representations that provide the flexibility and modularity to model large domains. They also highlight the importance of making general statements, rather than making statements for every single aspect of the world separately.

2. The fields of statistical learning and uncertainty in artificial intelligence emphasize that agents that operate in the real world must deal with uncertainty. An agent typically receives only noisy or limited information about the world; actions are often non-deterministic; and an agent has to take care of unpredictable events. Probability theory provides a sound mathematical foundation for inference and learning under uncertainty.

3. Machine learning, in general, argues that an agent needs to be capable of improving its performance through experience.

Thus, the combination of expressive knowledge representation with probabilistic approaches to inference and learning is needed in order to face the challenges of real-world applications, which are complex and heterogeneous. Most traditional artificial intelligence and machine learning systems, however, are able to handle either uncertainty or rich relational structures but not both.

Statistical learning, reinforcement learning, and data mining methods have traditionally been developed for data in attribute-value form only; data is represented in matrix form: columns represent attributes, and rows represent examples. Indeed, matrices are simple and efficient matrix operations can be used. In turn, a matrix form makes it possible to devise efficient algorithms.

Many, if not most, real-world data sets, however, are not in matrix form. Applications contain several entities and relationships among them. Inductive logic programming and relational learning have been developed for coping with this type of data.They do not, however, handle uncertainty in a principled way.

It is therefore not surprising that there has been a significant interest in integrating statistical learning with first order logic and relational representations. This newly emerging research field is known under the name of statistical relational learning and probabilistic logic learning, and may be briefly defined as follows:

Definition 12 *Statistical relational learning deals with machine learning and data mining in relational domains where observations may be missing, partially observed, and/or noisy.*

Instead of giving a probabilistic characterization of logic programming such as [Ng and Subrahmanian, 1992], this line of research stresses the machine learning aspect.

4.1 Role of Logical Abstraction in SRL

Employing relational and logical abstraction within statistical learning has three advantages:

1. Variables, *i.e.*, placeholders for entities allow one to make abstraction of specific entities.

2. Unification allows one to share information among entities.

3. In many applications, there is a rich background theory available, which can efficiently and elegantly be represented as sets of general regularities.

Thus, instead of learning regularities for each single entity independently, statistical relational learning aims at finding general regularities among groups of entities. The learned knowledge is declarative and compact, which makes it much easier for people to understand and to validate. Although, the learned knowledge must be recombined at run time using some reasoning mechanism such as backward chaining or resolution, which bears additional computational costs, statistical relational models are more flexible, context-aware, and offer, in principle, the full power of logical reasoning. Further, background knowledge often improves the quality of learning as it focuses learning on relevant patterns, *i.e.*, restricts the search space. While learning, relational and logical abstraction allow one to reuse experience: learning about one entity improves the prediction for other entities; it might even generalize to objects, which have never been observed before. Thus, relational and logical abstraction can make statistical learning more robust and efficient. This has been proven beneficial in many fascinating real-world applications in citation analysis, web mining, web navigation, web search, natural language processing, robotics, computer vision, social network analysis, bio-and chemo-informatics, electronic games, and activity recognition. For instance, Liao et al. [2005] applied Taskar et al.s relational Markov networks to a problem related to the ride sharing service, namely learning and inferring transportation routines.

Whereas most of the existing works on statistical relational learning have started from a statistical and probabilistic learning perspective and extended probabilistic formalisms with relational aspects, several pieces of research take a different perspective, starting from inductive logic programming (ILP) and studying how inductive logic programming formalisms, settings and techniques can be extended to deal with probabilities.

4.2 Inductive Logic Programming

ILP is a research field at the intersection of machine learning and logic programming [Muggleton and De Raedt, 1994]. It is often also called multi-relational data mining (MRDM) [D.zeroski and Lavra.c, 2001]. It aims at a formal framework as well as practical algorithms for inductively learning relational descriptions (in the form of logic programs) from examples and background knowledge. However, it does not explicitly deal with uncertainty such as missing or noisy information. Dealing explicitly with uncertainty makes probabilistic ILP more powerful than ILP and, in turn, than traditional attribute-value approaches. Moreover, there are several benefits of an ILP approach to statistical relational learning.

1. Classical ILP learning settings, as we will argue, naturally carry over to the probabilistic case. The probabilistic ILP settings make abstraction of specific probabilistic relational and first order logical representations and inference and learning algorithms yielding general statistical relational learning settings.

2. Many ILP concepts and techniques such as *more-general-than*, *refinement operators*, *least general generalization* (lub), and *greatest lower bound* (glb) can be reused. Therefore, many ILP learning algorithms such as Quinlans FOIL and De Raedt and Dehaspes Claudien can easily be adapted.

3. The ILP perspective highlights the importance of background knowledge within statistical relational learning. The research on ILP and on artificial intelligence in general has shown that background knowledge is the key to success in many applications.

4. An ILP approach should make statistical relational learning more intuitive to those coming from an ILP background and should cross-fertilize ideas developed in ILP and statistical learning.

Formally, ILP is concerned with finding a hypothesis H (a logic program, *i.e.* a definite clause program) from a set of positive and negative examples $\mathcal{E}^+$ and $\mathcal{E}^-$.

Consider learning a definition for the $Daughter/2$ predicate, *i.e.*, a set of clauses with head predicates over $Daughter/2$, given the following facts as learning examples:

$$\begin{array}{ll} \mathcal{E}^+\textbf{:} & Daughter(dorothy, ann). \\ & Daughter(dorothy, brian). \\ \hline \mathcal{E}^-\textbf{:} & Daughter(rex, ann). \\ & Daughter(rex, brian). \end{array}$$

Additionally, we have some general knowledge called background knowledge $\mathcal{B}$, which describes the family relationships and sex of each person:

$$\begin{array}{lll} Mother(ann, dorothy). & Female(dorothy). & Female(ann). \\ Mother(ann, rex). & Father(brian, dorothy). & Father(brian, rex). \end{array}$$

From this information, we could induce the following H:

$$\begin{array}{lll} Daughter(C, P) & :- & Female(C), Mother(P, C). \\ Daughter(C, P) & :- & Female(C), Father(P, C). \end{array}$$

which perfectly explains the examples in terms of the background knowledge, *i.e.*, $\mathcal{E}^+$ are entailed by H together with $\mathcal{B}$, but $\mathcal{E}^-$ are not entailed.

Definition 13 *Given a set of positive and negative examples $\mathcal{E}^+$ and $\mathcal{E}^-$ over some language $\mathcal{L}_E$, a background theory $\mathcal{B}$, in the form of a set of definite clauses, a hypothesis language $\mathcal{L}_H$, which specifies the clauses that are allowed in hypotheses, and a covers relation $covers(e, H, \mathcal{B}) \in \{0, 1\}$, which basically returns the classification of an example e with respect to $\mathcal{H}$ and $\mathcal{B}$, find a hypothesis H in $\mathcal{H}$ that covers (with respect to the background theory $\mathcal{B}$) all positive examples in $\mathcal{E}^+$ (completeness) and none of the negative examples in $\mathcal{E}^-$ (consistency).*

The (i) language $\mathcal{L}_E$ chosen for representing the examples together with the (ii) *covers* relation determines the inductive logic programming setting De Raedt [1997]. There are three broad settings, *viz.*,

1. learning from entailment [Plotkin, 1970]

2. learning interpretations [Helft, 1989, De Raedt and D.zeroski, 1994]

3. an intermediate setting called learning from proofs [Shapiro 1983]

4.3 Probabilistic ILP Settings

The inductive logic programming settings can be extended to the probabilistic case. When working with probabilistic ILP representations, there are essentially two changes:

1. *Probabilistic Clauses:* Clauses in H and $\mathcal{B}$ are annotated with probabilistic information, and Here, we use the following probability notations. With X, we denote a (random) variable. Furthermore, x denotes a state and $\mathbf{X}$ (resp. $\mathbf{x}$) a set of variables (resp. states). We will use Pr to denote a probability distribution, e.g., $\Pr(x)$, and p to denote a probability value, *e.g.*, $p(X = x)$ and $p(X = x)$.

2. *Probabilistic Covers:* The covers relation becomes probabilistic. A probabilistic covers relation softens the hard covers relation employed in traditional ILP and is defined as the probability of an example given the hypothesis and the background theory. A probabilistic covers relation takes as arguments an example e, a hypothesis H and possibly the background theory $\mathcal{B}$, and returns the probability value $\Pr(e|H, \mathcal{B}) \in [0, 1]$ of the example e given H and $\mathcal{B}$, *i.e.*, $covers(e, H, \mathcal{B}) = \Pr(e|H, \mathcal{B})$.

Using the probabilistic covers relation above, our first attempt at a definition of the probabilistic ILP learning problem is as follows:

Definition 14 *Given a probabilistic-logical language $\mathcal{L}_H$ and a set $\mathcal{E}$ of examples over some language $\mathcal{L}_E$, find the hypothesis H^* in $\mathcal{L}_H$ that maximizes $\Pr(\mathcal{E}|H^*, \mathcal{B})$.*

Under the usual *i.i.d.* assumption, *i.e.*, examples are sampled independently from identical distributions, this results in the maximization of

$$\Pr(\mathcal{E}|H^*, \mathcal{B}) = \prod_{e \in \mathcal{E}} P(e|H^*, \mathcal{B}) = \prod_{e \in \mathcal{E}} covers(e, H^*, \mathcal{B})$$

Similar to the ILP learning problem, the language $\mathcal{L}_E$ selected for representing the examples together with the probabilistic covers relation determines different learning setting. Guided by Definition 14, we will introduce several probabilistic ILP settings for statistical relational learning. The main idea is to lift traditional ILP settings by associating probabilistic information with clauses and interpretations and by replacing ILPs deterministic covers relation by a probabilistic one. In the discussion, we will have one trivial but important observation:

Observation: Derivations might fail.

The probability of a failure is zero and, consequently, failures are never observable. Only succeeding derivations are observable, *i.e.*, the probabilities of such derivations are greater than zero. As an extreme case, recall the negative examples $\mathcal{E}^-$ employed in the ILP learning problem on page 4.2. They are supposed to be not covered, *i.e.*,

$$p(\mathcal{E}^-|H,\mathcal{B}) = 0$$

In that example, *Rex* is a male person; he cannot be the daughter of *ann*. Thus, $Daughter(rex, ann)$ was listed as a negative example. **Negative examples conflict with the usual view on learning examples in statistical learning.** In statistical learning, we seek to find that hypothesis H^*, which is most likely given the learning examples:

$$H^* = \underset{\text{argmax}}{H} p(H|\mathcal{E}) = \underset{\text{argmax}}{H} \frac{p(\mathcal{E}|H)p(H)}{p(\mathcal{E})} \quad with \;\; p(\mathcal{E}) > 0$$

Thus, examples $\mathcal{E}$ are observable, *i.e.*, $p(E) > 0$. Therefore, we refine the preliminary probabilistic ILP learning problem definition 14. In contrast to the purely logical case of ILP, we do not speak of positive and negative examples anymore but of possible and impossible ones.

Definition 15 *Given a set $\mathcal{E} = \mathcal{E}_p \cup \mathcal{E}_i$ of possible and impossible examples $\mathcal{E}_p$ and $\mathcal{E}_i$ (with $\mathcal{E}_p \cap \mathcal{E}_i = \emptyset$) over some example language $\mathcal{L}_E$, a probabilistic covers relation $covers(e, H, \mathcal{B}) = P(e|H, \mathcal{B})$, a probabilistic logical language $\mathcal{L}_H$ for hypotheses, and a background theory $\mathcal{B}$, find a hypothesis H^* in $\mathcal{L}_H$ such that $H^* = \underset{\text{argmax}}{H} score(\mathcal{E}, H, \mathcal{B})$ and the following constraints hold:*

$$\forall e_p \in \mathcal{E}_p : \quad covers(e_p, H^*, \mathcal{B}) > 0 \quad and \forall e_i \in \mathcal{E}_i : \; covers(e_i, H^*, \mathcal{B}) = 0$$

The scoring function is some objective score, usually involving the probabilistic covers relation of the possible examples such as the observed likelihood

$$\prod_{e_p \in \mathcal{E}_p} covers(e_p, H^*, \mathcal{B})$$

some penalized variant thereof.

The probabilistic ILP learning problem of Definition 15 unifies ILP and statistical learning in the following sense:

1. Using a deterministic covers relation (which is either 1 or 0) yields the classical ILP learning problem, see Definition 13.

2. On the other hand, sticking to propositional logic and learning from possible examples, i.e., $P(\mathcal{E}) > 0$, only yields traditional statistical learning.

3. Definition 15 makes abstraction of many particular kinds of problems.

    - In density estimation, the joint probability distribution of some random variables is estimated

- Whereas, in classification and regression the dependency of a discrete respectively continuous target variable given the value of some other variables is estimated.

4. Furthermore, several types of learning can be distinguished.

    - In supervised learning, the training examples contain information about all variables including the target variable.
    - In reinforcement learning, the training examples contain only indirect target information such as the classifier did well or not (in the form of some reward).
    - Finally, in unsupervised learning, no values of the target variable are observed.

5. Another important distinction is whether all the random variables variables are observed, or whether some of them are hidden, *e.g.*, they are specified in the background knowledge and never observed.

6. We can also formulate the learning problem as either of the following:

    - As a 'point estimation problem, *i.e.*, the goal is to find a single best hypothesis H^*.
    - As a 'Bayesian learning problem', where the goal is to return a posterior distribution over hypotheses.

7. Finally, learning might refer to the structure, *i.e.*, the underlying logic program of the hypothesis, the parameters, or both. To come up with algorithms solving probabilistic ILP learning problems, say for density estimation, one typically distinguishes two subtasks because $H = (L, \lambda)$ is essentially a logic program L annotated with probabilistic parameters λ:

    - Parameter estimation where it is assumed that the underlying logic program L is fixed, and the learning task consists of estimating the parameters λ that maximize the likelihood.
    - Structure learning where both L and λ have to be learned from the data.

Similar to that for traditional ILP, there are three probabilistic ILP settings, which extend the purely logical ones. The three ILP settings and their probabilistic extensions are outlined in the following sections.

4.3.1 Probabilistic setting for justification

There are two important components in SRL:

Abduction. Process of hypothesis formation. The logical setting for abduction is what traditional ILP is mostly about.

Justification. The degree of belief assigned to an hypothesis given a certain amount of evidence.

Recall the Bayes' Theorem

$$p(h|E) = \frac{p(h).p(E|h)}{p(E)}$$

The best hypothesis in a set $\mathcal{H}$ (ignoring ties)

$$H \ = \ argmax_{h \in \mathcal{H}} \ p(h|E)$$

We will consider the following learning framework.

Let X be a countable set of instances (encodings of all objects of interest) and D_X be a probability measure on X

Let $\mathcal{C} \subseteq 2^X$ be a countable set of concepts and $D_{\mathcal{C}}$ be a probablity measure on 2^X

Let $\mathcal{H}$ be a countable set of hypotheses[1] and $D_{\mathcal{H}}$ be a probability measure (prior) over $\mathcal{H}$

Let the concept represented by $h \in \mathcal{H}$ be $c(h) \in \mathcal{C}$

Let $\mathcal{C}$ and $\mathcal{H}$ be such that

- for each $C \in \mathcal{C}$, there is an $h \in \mathcal{H}$ s.t. $C = c(h)$
- for each $C \in \mathcal{C}$, $D_{\mathcal{C}}(C) \ = \ \sum_{\{h \in \mathcal{H} | C = c(h)\}} P(h)$

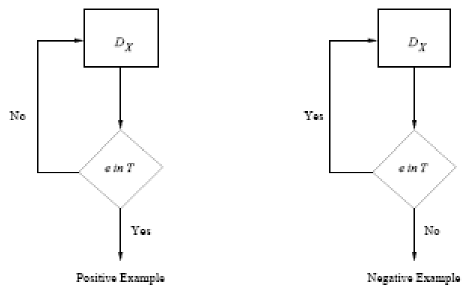
Target concept T is chosen using the distribution $D_{\mathcal{C}}$

Let $g(h)$ denote the proportion (w.r.t. the instance space) of the concept represented by a hypothesis $h \in \mathcal{H}$

- That is, $g(h) = \sum_{x \in c(h)} D_X(x)$
- $g(h)$ is a measure of the "generality" of h

Noise Free Data

Following is a model for noise free data.



[1]Note that $\mathcal{H}$ also been used earlier to denote the space of horn clauses. This time, it is used to denote the space of hypothesis, which anyways happens to be within the space of horn clauses.

Given $\mathcal{E} = \mathcal{E}^+ \cup \mathcal{E}^-$

$$p(h|E) \propto D_{\mathcal{H}}(h) \prod_{e \in E^+} p(e|h) \prod_{e \in E^-} p(e|h)$$

Or

$$P(h|E) \propto D_{\mathcal{H}}(h) \prod_{e \in E^+} \frac{D_X(e)}{g(h)} \prod_{e \in E^-} \frac{D_X(e)}{1-g(h)}$$

Assuming p positive and n negative examples

$$P(h|E) \propto D_{\mathcal{H}}(h) \left(\prod_{e \in E} D_X(e) \right) \left(\frac{1}{g(h)} \right)^p \left(\frac{1}{1-g(h)} \right)^n$$

Maximal $P(h|E)$ means finding the hypothesis that maximises

$$logD_{\mathcal{H}}(h) + p \ log \frac{1}{g(h)} + n \ log \frac{1}{1-g(h)}$$

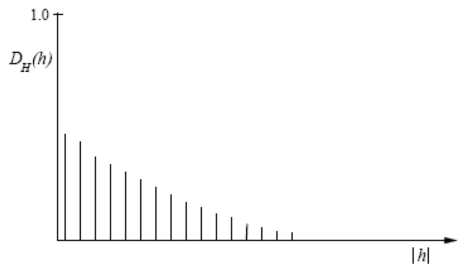
If there are no negative examples, then this becomes

$$logD_{\mathcal{H}}(h) + p \ log \frac{1}{g(h)}$$

Let us answer some questions at this point:

1. *What is the distribution $D_{\mathcal{H}}(h)$:*

   A common assumption is that "larger" programs are less likely (in coding terminology, require more bits to encode)



As an example

$$D_{\mathcal{H}}(h) = 2^{-|h|}$$

That is

$$logD_{\mathcal{H}}(h) = -|h|$$

2. What is generality function $g(h)$?

   Recall that $g(h) = \sum_{x \in c(h)} D_X(x)$

   – $c(h)$ may be infinite

   – D_X is usually unknown (and is a mapping to the reals)

   Have to be satisfied with approximate estimates of $g(h)$

   Estimation procedure

   (a) Randomly generate a finite sample of n instances using a known distribution (for eg. uniform)

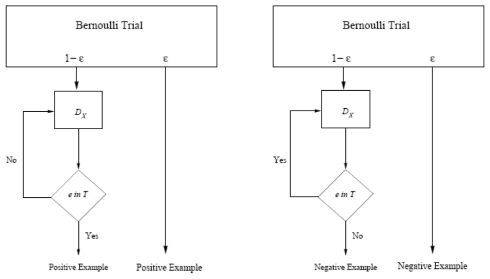
   (b) Determine the number of these instances (say c) entailed by h

   (c) $g(h) \approx \frac{c+1}{n+2}$

3. What about noisy data? This will be addressed in the next subsection.

A Model for Noisy Data

Following is a model for justification in the presence of noisy data.



For any hypothesis h the examples $\mathcal{E} = \mathcal{E}^+ \cup \mathcal{E}^-$ can now be partitioned as follows

1. $TP = \{e | e \in E^+ \; and \; e \in c(h)\}$ (true positives)
2. $FN = \{e | e \in E^+ \; and \; e \notin c(h)\}$ (false negatives)
3. $FP = \{e | e \in E^- \; and \; e \in c(h)\}$ (false positives)
4. $TN = \{e | e \in E^- \; and \; e \notin c(h)\}$ (true negatives)

Recall

$$p(h|E) \propto D_{\mathcal{H}}(h) \prod_{e \in E^+} p(e|h) \prod_{e \in E^-} p(e|h)$$

Now

$$\prod_{e \in E^+} p(e|h) = \prod_{e \in TP} \left(\frac{D_X(e)(1-\epsilon)}{g(h)} + D_X(e)\epsilon \right) \prod_{e \in FN} D_X(e)\epsilon$$

$$\prod_{e \in E^-} p(e|h) = \prod_{e \in TN} \left(\frac{D_X(e)(1-\epsilon)}{1-g(h)} + D_X(e)\epsilon \right) \prod_{e \in FP} D_X(e)\epsilon$$

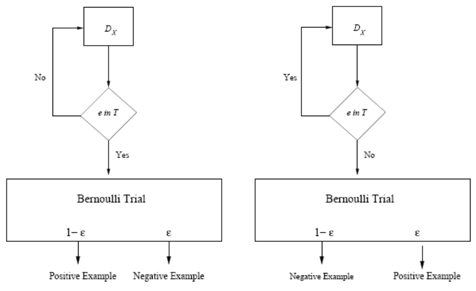
So, with $FPN = FP \cup FN$

$$p(h|E) \propto D_{\mathcal{H}}(h) \left(\prod_{e \in E} D_X(e) \right) \left(\frac{1-\epsilon}{g(h)} \right)^{|TP|} \left(\frac{1-\epsilon}{1-g(h)} \right)^{|TN|} \epsilon^{|FPN|}$$

Maximal $P(h|E)$ means finding the hypothesis that maximises

$$log D_{\mathcal{H}}(h) \ + \ |TP| \ log \frac{1-\epsilon}{g(h)} \ + \ |TN| \ log \frac{1-\epsilon}{1-g(h)} \ + \ |FPN| log \epsilon$$

Another model for noisy data is outlined below:



4.4 Learning from Entailment

Learning from entailment is by far the most popular ILP setting and it is addressed by a wide variety of well-known ILP systems such as FOIL [Quinlan and Cameron-Jones, 1995], Progol [Muggleton, 1995], and Aleph [Srinivasan, 1999]. When learning from entailment, the examples are definite clauses and a hypothesis H covers an example e with respect to the background theory $\mathcal{B}$ if and only if $\mathcal{B} \cup H \models e$, *i.e.*, each model of $B \cup H$ is also a model of e.

In many well-known systems, such as FOIL, one requires that the examples are ground facts. To illustrate the above setting, consider the following example inspired on the well-known mutagenicity application [Srinivasan et al., 1996].

Consider the following facts in the background theory $\mathcal{B}$, which describe part of molecule 225.

$Molecule(225).$	$Logmutag(225, 0.64).$
$Lumo(225, -1.785).$	$Logp(225, 1.01).$
$Nitro(225, [f1\_4, f1\_8, f1\_10, f1\_9]).$	$Atom(225, f1\_1, c, 21, 0.187).$
$Atom(225, f1\_2, c, 21, -0.143).$	$Atom(225, f1\_3, c, 21, -0.143).$
$Atom(225, f1\_4, c, 21, -0.013).$	$Atom(225, f1\_5, o, 52, -0.043).$
$Bond(225, f1\_1, f1\_2, 7).$	$Bond(225, f1\_2, f1\_3, 7).$
$Bond(225, f1\_3, f1\_4, 7).$	$Bond(225, f1\_4, f1\_5, 7).$
$Bond(225, f1\_5, f1\_1, 7).$	$Bond(225, f1\_8, f1\_9, 2).$
$Bond(225, f1\_8, f1\_10, 2).$	$Bond(225, f1\_1, f1\_11, 1).$
$Ring\_size\_5(225, [f1\_5, f1\_1, f1\_2, f1\_3, f1\_4]).$	$Bond(225, f1\_11, f1\_12, 2).$
$Hetero\_aromatic\_5\_ring(225, [f1\_5, f1\_1, f1\_2, f1\_3, f1\_4]).$	$Bond(225, f1\_11, f1\_13, 1).$
...	...

Consider now the positive example $Mutagenic(225)$. It is covered by the following H:

$$Mutagenic(M) \quad \text{:-} \quad Nitro(M, R1), Logp(M, C), C > 1.$$

together with the background knowledge $\mathcal{B}$, because $H \cup \mathcal{B}$ entails the example. To see this, we unify $Mutagenic(225)$ with the clauses (H) head. This yields

$$Mutagenic(225) \quad :- \quad Nitro(225, R1), Logp(225, C), C > 1.$$

Now, $Nitro(225, R1)$ unifies with the third ground atom (left-hand side column) in $\mathcal{B}$, and $Logp(225, C)$ with the second one on the right. Because $1.01 > 1$, we found a proof of mutagenic(225).

There are, broadly speaking, three types of ILP approaches for learning from entailment: *top-down approaches*, *bottom-up approaches* and *hybrid approaches*.

1. *Top-down approaches* start from short clauses, iteratively adding literals to their bodies as long as they do not become to overly general. Basically, in top-down approaches, hypotheses are generated in a pre-determined order, and then tested against the examples. More precisely:

   (a) They start with the most general hypothesis, *i.e.*, clauses of the form $Daugther(C, P) : -True$, where all arguments are distinct variables.

   (b) After seeing the first example that contradicts the hypothesis, *i.e.*, after seeing the first negative example, the hypothesis is specialized by specializing the general clause.

   (c) The clause is specialized typically in two ways:

      - by applying a substitution,
      - by adding a literal, *i.e.*, an atom or its negation to the body, and

For instance, we can specialize $Daughter(C, P) : -True$ and consider $Daughter(C, P) : -Female(C)$ and $Daughter(C, P) : -Mother(P, C)$ for further investigations. Several possibilities (and successive specializations) have to be tried before one finds a clause that covers some positive examples but no negative ones such as $Daugther(C, P) : -Female(C), Mother(P, C)$.

(d) If some positive examples are still not covered, these techniques typically add a new, maximally general clause to the hypothesis and essentially iterate the process in step 1c as before, until all positive examples are covered and no negative example is covered.

(e) The background knowledge $\mathcal{B}$ is typically viewed as a logic program ($i.e.$ a definite clause program) that is provided to the inductive logic programming system and fixed during the learning process. The hypothesis H together with the background theory $\mathcal{B}$ should cover all positive and none of the negative examples.

Top-down approaches are often employed by ILP systems that learn from entailment. More precisely, these systems often employ a separate-and-conquer rule-learning strategy [Furnkranz, 1999]. In an outer loop of the algorithm, they follow a set-covering approach [Mitchell, 1997] in which they repeatedly search for a rule covering many positive examples and none of the negative examples. They then delete the positive examples covered by the current clause and repeat this process until all positive examples have been covered. In the inner loop of the algorithm, they typically refine a clause by unifying variables, by instantiating variables to constants, and/or by adding literals to the clause.

Figure 4.1 presents the generic framework for top-down ILP systems while Figure 4.2 outlines one of the original ILP systems, MIS.

2. *Bottom-up approaches* start from long clauses, iteratively removing literals until they would become overly general. While top-down approaches successively specialize a very general starting hypothesis, bottom-up approaches successively generalize a very specific hypothesis. This is basically done

   (a) by deleting literals (or clauses),
   (b) by turning constants into variables and/or
   (c) by turning bounded variables into new variables.

   At each step, the theory is generalized by taking the least general generalization (under .-subsumption) of pairwise clauses. Of course, care must be taken that the generalized theory does not cover

   negative examples.

3. *Hybrid approaches* that mix top-down and bottom-up searches. Hybrid approaches are usually employed for multiple predicate learning [De Raedt et al., 1993] and theory revision [Wrobel, 1996].

```
**Initialize** $\mathcal{E}_{cur} = \mathcal{E}$;
**Initialize** $\mathcal{H} = \emptyset$;
//Start covering
**repeat**
  **Initialize** $C = T \leftarrow$;
  //Start specialization
  **repeat**
    Find the best refinement $C_{best} \in \rho_{\mathcal{L}}(C)$;
    Assign $C = C_{best}$;
  **until** Necessity stopping criterion is satisfied;
  Add $C$ to $\mathcal{H}$ to get new hypothesis $\mathcal{H}' = \mathcal{H} \cup C$;
  Remove positive examples covered by $C$ from $\mathcal{E}_{cur}$ to get new training set
  $\mathcal{E}^{'}_{cur} = \mathcal{E}_{cur} - \bigcup_{\{e \in \mathcal{E}^{+}_{cur} | covers(\mathcal{B},\mathcal{H}',e)=1\}} \{e\}$;
  Assign $\mathcal{E}_{cur} = \mathcal{E}^{'}_{cur}$, $\mathcal{H} = \mathcal{H}'$;
**until** Sufficiency stopping criterion is satisified;
```

Figure 4.1: A general strategy for top-down ILP approaches.

4.4.1 Constraining the ILP Search

It should be stressed that ILP is a difficult problem. Practical ILP systems fight the inherent complexity of the problem by imposing all sorts of constraints, mostly syntactic in nature. Such constraints include

1. *Language and search biases:* These are sometimes summarized as declarative biases, (see [Nedellec et al., 1996] for an overview). Essentially, the main source of complexity in ILP steams from the variables in the clauses. In top-down systems, the branching factor of the specialization operator increases with the number of variables in the clauses. Following are frequently adopted techniques for reducing this branching factor by introducing language and search bias.

   (a) Introducing types for predicates can rule out main potential substitutions and unifications. As an example, the type definition $type(Father(person, person))$ specifies that both argument of atoms over $Father/2$ have to be persons.

   (b) Refinement operators can also be used to encode a language bias, since they can be restricted to generate only a subset of the language $\mathcal{L}_H$. For instance, refinement operators can easily be modified to generate only constant-free and function-free clauses.

   (c) Other methods use a kind of grammar construction to explicitly declare the range of acceptable clauses, see e.g. Cohen [1994].

   (d) Lookaheads are an example of a search bias. In some cases, an atom might never be chosen by our algorithm because it will not, in itself,

```
**Initialize** hypothesis $H$ to a (possibly empty) set of clauses in $\mathcal{L}_H$;
**loop**
  Read the next (positive or negative) example;
  **repeat**
    **if** There exists an $e \in \mathcal{E}^-$ covered by $H$ **then**
      Delete incorrect clauses from $H$.
    **end if**
    **if** There exists an $e \in \mathcal{E}^+$ not covered by $H$ **then**
      With breadth-first search of the refinement graph develop a clause $C$
      which covers $e$ and add it to $H$;
    **end if**
  **until** $H$ is complete and consistent;
  **return** Hypothesis $H$;
**end loop**
```

Figure 4.2: A simplified version of the MIS algorithm by [Shapiro 1983].which is the general strategy of top-down approaches.

result in a better score. However, such an atom, while not useful in itself, might introduce new variables that make a better coverage possible by adding another atoms later on [Quinlan, 1991] 7. It is usually solved by allowing the algorithm to look ahead in the search space. Instead of considering refinements with a single atom, one considers larger refinements consisting of multiple atoms [Blockeel and De Raedt, 1997].

2. *Bound on number of distinct variables:* One can also put a bound in the number of distinct variables that can occur in clauses.

3. *Mode declarations:* Mode declarations are another well-known ILP devise. They are used to describe input-output behaviour of predicate definitions. For example, we might specify $mode(Daugther(+,-))$ and $mode(Father(-,+))$, meaning that the $+$ arguments must be instantiated, whereas the $-$ arguments will be bounded to the answer.

In general, a model can suffer from either underfitting or overfitting. A model that is not sufficiently complex can fail to fully detect the underlying rule of a complicated data set, leading to underfitting. A model that is too complex may fit the noise, not just the underlying rule, leading to overfitting and, for instance, wild predictions.

4.4.2 Example: Aleph

As an example for the learning from entailments setting, we will discuss Aleph. Given background knowledge $\mathcal{B}$ and positive examples $\mathcal{E}^+ = e_1 \wedge e_2 \ldots$, negative examples $E^- = \overline{f_1} \wedge \overline{f_2} \wedge \ldots$, the generic ILP system Aleph, is concerned with finding a hypothesis $H = D_1 \wedge \ldots$ that satisfies (note: $\cup$ and $\wedge$ used interchangeably)

Prior Satisfiability. $B \wedge \mathcal{E}^- \not\models \Box$

Prior Necessity. $B \not\models \mathcal{E}^+$

Posterior Sufficiency. $\mathcal{B} \wedge H \models \mathcal{E}^+$ and $\mathcal{B} \wedge D_j \models e_1 \vee e_2 \vee \ldots$

Posterior Satisfiability. $\mathcal{B} \wedge H \wedge \mathcal{E}^- \not\models \Box$

If more than one H satisfies this, the one with highest posterior probability is chosen. The D_i can be found by examining clauses that "relatively subsume" at least one example, making use of plotkin's relative subsumption discussed elaborately in Section 2.5.1. This gives the following sufficient implementation of Aleph, given $\mathcal{B}, \mathcal{E}$.

```
1. $H_0 = \mathcal{B}, i = 0, \mathcal{E}^+ = \{e_1, \ldots, e_n\}$
2. repeat
    (a) increment $i$
    (b) Obtain the most specific clause $\perp(\mathcal{B}, e_i)$
    (c) Find the clause $D_i$ that: subsumes $\perp(\mathcal{B}, e_i)$; and is consistent with
        the negative examples;
    (d) $H_i = H_{i-1} \cup \{D_i\}$
3. until $i > n$
4. return $H_n$
```

As discussed in Section 2.5.1, there are problems with this implementation. Particularly,

- $\perp(\mathcal{B}, e_i)$ may be infinite
- This implementation may perform a lot of redundant computation ($D_i \in H_{i-1}$)
- This implementation need not return the hypothesis with maximum posterior probability

So we next present a "Greedy" implementation for Aleph, given $\mathcal{B}, \mathcal{E}$.

```
1. $H_0 = \mathcal{B}, \mathcal{E}_0^+ = \mathcal{E}^+, i = 0.$
2. Repeat
    (a) Increment $i$
    (b) Randomly choose a positive example $e_i$ from $\mathcal{E}_{i-1}^+$
    (c) Obtain the most specific clause $\perp(\mathcal{B}, e_i)$
```

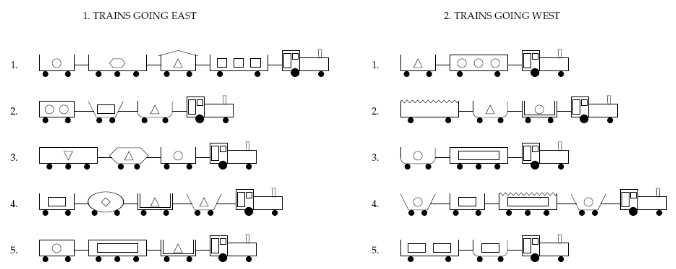


Figure 4.3: The train-spotting example for illustrating Aleph.

```
        (d) Find the clause $D_i$ that: subsumes $\perp(\mathcal{B}, e_i)$; and is consistent with the
            negative examples; and maximises $p(H_{i-1} \cup \{D_i\}|e_i^+ \cup \mathcal{E}^-)$ where $e_i^+$ are
            the examples in $\mathcal{E}^+$ made redundant by $H_{i-1} \cup \{D_i\}$
        (e) $H_i = H_{i-1} \cup \{D_i\}$
        (f) $\mathcal{E}_i^+ = \mathcal{E}_{i-1}^+ \backslash e_i^+$
    3. until $\mathcal{E}_i^+ = \emptyset$
    4. return $H_i$
```

However, this implementation does not address the problem that $\perp(\mathcal{B}, e_i)$ may be infinite and that it need not return in the hypothesis with maximum posterior probability. The problem of infinite $\perp(\mathcal{B}, e_i)$ may be addressed by making use of mode declarations introduced on page 120. This gives a revised "Greedy" implementation for Aleph, given $\mathcal{B}, \mathcal{E}, d$.

```
    1. $H_0 = \mathcal{B}, \mathcal{E}_0^+ = \mathcal{E}^+, i = 0$
    2. Repeat
        (a) Increment $i$
        (b) Randomly choose a positive example $e_i$ from $\mathcal{E}_{i-1}^+$
        (c) Obtain the most specific clause $\perp_d(\mathcal{B}, e_i)$
        (d) Find the clause $D_i$ that: subsumes $\perp(\mathcal{B}, e_i)$; and is consistent with the
            negative examples; and maximises $p(H_{i-1} \cup \{D_i\}|e_i^+ \cup \mathcal{E}^-)$ where $e_i^+$ are
            the examples in $\mathcal{E}^+$ made redundant by $H_{i-1} \cup \{D_i\}$
        (e) $H_i = H_{i-1} \cup \{D_i\}$
        (f) $\mathcal{E}_i^+ = \mathcal{E}_{i-1}^+ \backslash e_i^+$
    3. until $\mathcal{E}_i^+ = \emptyset$
    4. return $H_i$
```

As an example, let us consider the trainspotting problem (*c.f.* Figure 4.3) to illustrate the discussion thus far.

We will use the following mode declarations for the trainspotting example.

```
:- modeh(1,eastbound(+train)).
:- modeb(1,short(+car)).
:- modeb(1,closed(+car)).
:- modeb(1,long(+car)).
:- modeb(1,open_car(+car)).
:- modeb(1,double(+car)).
:- modeb(1,jagged(+car)).
:- modeb(1,shape(+car,#shape)).
:- modeb(1,load(+car,#shape,#int)).
:- modeb(1,wheels(+car,#int)).
:- modeb(*,has_car(+train,-car)).
```

Further, the examples $\mathcal{E}^+$ and $\mathcal{E}^-$ for this example will be:

Positive	**Negative**
`eastbound(east1).`	`eastbound(west6).`
`eastbound(east2).`	`eastbound(west7).`
`eastbound(east3).`	`eastbound(west8).`
`eastbound(east4).`	`eastbound(west9).`
`eastbound(east5).`	`eastbound(west10).`

whereas, the background knowledge $\mathcal{B}$ will comprise:

```
% type definitions
car(car_11).  car(car_12). ...
car(car_21).  car(car_22).  ...
...

shape(elipse).  shape(hexagon). ...
...

% eastbound train 1
has_car(east1,car_11).  has_car(east1,car_12). ...
shape(car_11,rectangle). shape(car_12,rectangle). ...
open_car(car_11). closed(car_12).
long(car_11).  short(car_12). ...
...

% westbound train 6
has_car(west6,car_61).  has_car(west6,car_62). ...
long(car_61).  short(car_62).
shape(car_61,rectangle).  shape(car_62,rectangle).
...
```

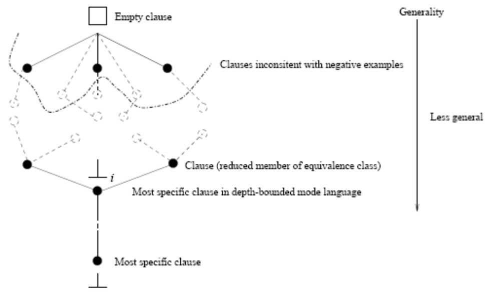
The outcome of a simple search in Aleph, for the trainspotting setting is outlined next:

```
eastbound(A) :-
   has_car(A,B).
```

```
[5/5]
eastbound(A) :-
   has_car(A,B), short(B).
[5/5]
eastbound(A) :-
   has_car(A,B), open_car(B).
[5/5]
eastbound(A) :-
   has_car(A,B), shape(B,rectangle).
[5/5]

...

[theory]

[Rule 1] [Pos cover = 5 Neg cover = 0]

eastbound(A) :-
       has_car(A,B), short(B), closed(B).

[pos-neg] [5]
```

We will next discuss the search and redundancy aspects of Aleph. There are two stages in the clause-by-clause construction of hypothesis, which are summarized below:

1. Search


Empty clause
Generality
Clauses inconsitent with negative examples
Less general
Clause (reduced member of equivalence class)
$\perp_i$
Most specific clause in depth-bounded mode language
Most specific clause
$\perp$

2. Remove redundant clauses once best clause is found

Aleph has provision for bottom-up as well as top-down search for moving about in the lattice. The following refinement steps can be used in Aleph.

General-to-specific search: start at $\square$, and move by

1. Adding a literal drawn from $\perp_i$

   $p(X,Y) \leftarrow q(X)$ becomes $p(X,Y) \leftarrow q(X), r(Y)$

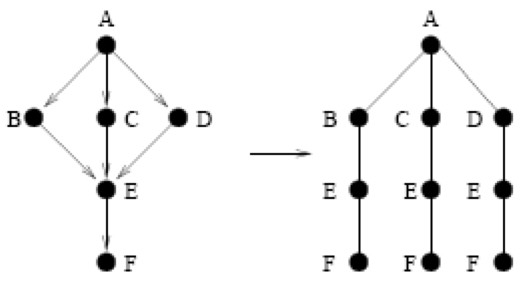


Figure 4.4: The conversion of (relative) subsumption lattice to tree.

2. Equating two variables of the same type

   $p(X,Y) \leftarrow q(X)$ becomes $p(X,X) \leftarrow q(X)$

3. Instantiate a variable with a general functional term or constant

   $p(X,Y) \leftarrow q(X)$ becomes $p(3,Y) \leftarrow q(3)$

Specific-to-general search: start at $\perp_i$

Dual operations to above (ie. remove literal etc.)

Progol does a general-to-specific search

Next, we discuss some search methods. The subsumption lattice can be represented as a directed acyclic graph. However, the DAG can be converted to a tree such that root is the first node ($\Box$ or $\perp_i$) and children of a node are refinements. The conversion of a lattice to a search tree is illustrated in Figure 4.4

Searching the lattice is therefore equivalent to searching a tree. Out goal is to find the node (goal node) that has greatest "compression". There are two basic types of tree search: depth-first (DF) and breadth-first (BF) as discussed in chapter 3. DF and BF are "blind". More guidance is desirable at any node s. The guidance can be in one or more of the following forms:

- g_s: cost of optimal path from root to s
- h_s: estimated cost of optimal path to goal from s

The different kinds of guided search can be summarized as follows:

Hill-climbing: DF with h_s

Best-first: BF with h_s

Best-cost: BF with g_s

A^*: BF with g_s and h_s

Progol does an A^*-like search – uses utilit instead of costs. For a clause C at a node

$$\begin{aligned}
p_s &= |\{e : e \in E^+ \text{ and } B \wedge C \wedge \overline{e} \models \Box\}| \\
n_s &= |\{e : e \in E^- \text{ and } B \wedge C \wedge e \models \Box\}| \\
c_s &= |C| - 1 \\
l_s &= \text{l.b. on literals needed} \\
h_s &= -(n_s + c_s + l_s) \\
g_s &= p_s \\
f_s &= g_s + h_s
\end{aligned}$$

Compression of a clause at node $s = p_s - n_s - c_s$ Progol seeks clauses with $c_s < c$ and $n_s = 0$ and guaranteed to have maximum $p_s - c_s - h_s$. Progol has several admissable pruning strategies to reduce search.

E.g. Let $children(s) = S$. If $f(s) > p_{s'} - c_{s'}$ for $s' \in S$ then prune(s')

We next outline an optimal search algorithm that uses branch-and-bound.

$bb(i, \rho, f)$: Given an initial element i from a discrete set S; a successor function $\rho : S \rightarrow 2^S$; and a cost function $f : S \rightarrow \Re$, return $H \subseteq S$ such that H contains the set of cost-minimal models. That is for all $h_{i,j} \in H, f(h_i) = f(h_j) = f_{min}$ and for all $s' \in S \backslash H$ $f(s') > f_{min}$.

```
1. $Active := \langle i \rangle$.
2. $best := \infty$
3. $selected := \emptyset$
4. while $Active \neq \langle\rangle$
5. begin
   (a) remove element $k$ from $Active$
   (b) $cost := f(k)$
   (c) if $cost < best$
   (d) begin
       i. $best := cost$
       ii. $selected := \{k\}$
       iii. let $Prune_1 \subseteq Active$ s.t. for each $j \in Prune_1$, $\underline{f}(j) > best$ where
            $\underline{f}(j)$ is the lowest cost possible from $j$ or its successors
       iv. remove elements of $Prune_1$ from $Active$
   (e) end
   (f) elseif $cost = best$
       i. $selected := selected \cup \{k\}$
   (g) $Branch := \rho(k)$
   (h) let $Prune_2 \subseteq Branch$ s.t. for each $j \in Prune_2$, $\underline{f}(j) > best$ where
       $\underline{f}(j)$ is the lowest cost possible from $j$ or its successors
   (i) $Bound := Branch \backslash Prune_2$
   (j) add elements of $Bound$ to $Active$
```

```
6. end
7. return selected
```

Different search methods result from specific implementations of *Active*:

– Stack: depth-first search

– Queue: breadth-first search

– Prioritised Queue: best-first search

There are two types of redundancies in the hypothesis H:

1. **Redundancy 1: Literal Redundancy:**

   Literal l is redundant in clause $C \vee l$ relative to background B iff

   $$B \wedge (C \vee l) \equiv B \wedge C$$

   It can be shown that the literal l is redundant in clause $C \vee l$ relative to the background B iff

   $$\mathcal{B} \wedge (C \vee l) \models C$$

   The clause C is said to be reduced (*c.f.* Section 2.1, page 102) with respect to background knowledge $\mathcal{B}$ iff no literal in C is redundant.

2. **Redundancy 2: Clause redundancy:**

   Clause C is redundant in the $\mathcal{B} \wedge C$ iff $\mathcal{B} \wedge C \equiv \mathcal{B}$.

   It can be shown that clause C is redundant in $\mathcal{B} \wedge C$ iff

   $$\mathcal{B} \models C \equiv \mathcal{B} \wedge \overline{C} \models \square$$

   A set of clauses S is said to be reduced iff no clause in S is redundant

   Progol uses this procedure to determine (and remove) examples made redundant by clause found in the search.

   Example

   $$\begin{array}{rc} e_j: & gfather(henry, john) \leftarrow \\ \\ B: & father(henry, jane) \leftarrow \\ & father(henry, joe) \leftarrow \\ & parent(jane, john) \leftarrow \\ & parent(joe, robert) \leftarrow \\ \\ D_j: & gfather(X, Y) \leftarrow father(X, Z), parent(Z, Y) \end{array}$$

e_j is redundant in $B \wedge D_j \wedge e_j$ since $B \wedge D_j \wedge \overline{e_j} \models \Box$

There are several implementation issues that need further thought.

Question. Will the clause-by-clause search method yield the best set of clauses? If no, why not?

Question. Is it possible to do a theory-by-theory search?

Question. Is it possible devise a complete search that is non-redundant? If no, why not?

4.5 Probabilistic Learning from Entailment

Probabilistic learning from entailment has been investigated for learning stochastic logic programs [Muggleton, 2000a,b, Cussens, 2001, Muggleton, 2002] and for parameter estimation of PRISM programs [Sato and Kameya, 2001, Kameya et al., 2004] from possible examples only.

In order to integrate probabilities in the entailment setting, we need to find a way to assign probabilities to clauses that are entailed by an annotated logic program. Since most ILP systems working under entailment employ ground facts for a single predicate as examples, we will restrict our attention to assign probabilities to facts for a single predicate[2].

More formally, let us annotate a logic program H consisting of a set of clauses of the form $q \leftarrow b_i$, where p is an atom of the form $p(V_1, \ldots, V_n)$ with the V_i different variables, and the b_i are different bodies of clauses. Furthermore, we associate to each clause in H the probability values $p(b_i|q)$; they constitute the conditional probability distribution that for a random substitution θ for which $q\theta$ is ground and true (resp. false), the query $b_i\theta$ succeeds (resp. fails) in the knowledge base $\mathcal{B}$. Recall that the query q succeeds in $\mathcal{B}$ if there is a substitution σ such that $\mathcal{B} \models q\sigma$. Furthermore, we assume the prior probability of q is given as $p(q)$, which denotes the probability that for a random substitution θ, $p\theta$ is true (resp. false). This can then be used to define the covers relation $p(q\theta \mid H, \mathcal{B})$ as follows (we delete the $\mathcal{B}$ as it is fixed):

$$p(q\theta|H) = p(q\theta \mid b_1\theta, \ldots, b_k\theta) = \frac{p(b_1\theta, \ldots, b_k\theta|q\theta) \times p(q\theta)}{p(b_1\theta, \ldots, b_k\theta)}$$

For instance, applying the naive Bayes assumption yields

$$p(q\theta|H) = \frac{\prod_i p(b_i\theta|q\theta) \times p(q\theta)}{p(b_1\theta, \ldots, b_k\theta)} \tag{4.1}$$

[2]It remains an open question as how to formulate more general frameworks for working with entailment.

Finally, since $p(q\theta \mid H) + p(\neg q\theta \mid H) = 1$, we can compute $p(q\theta|H)$ without $p(b_1\theta, \ldots, b_k\theta)$ through normalization.

As an example, consider again the mutagenicity domain and the following annotated logic program:

$$\begin{aligned}
(0.01, 0.21): &\quad Mutagenetic(M) \leftarrow Atom(M, \_, \_, 8, \_)\\
(0.38, 0.99): &\quad Mutagenetic(M) \leftarrow Bond(M, \_, A, 1), Atom(M, A, c, 22, \_), Bond(M, A, 2)
\end{aligned}$$

We denote the first clause by b_1 and the second one by b_2. The vectors on the left-hand side of the clauses specify $p(b_i\theta = true \mid q\theta = true)$ and $p(b_i\theta = true \mid q\theta = false)$ respectively. The covers relation (assuming the Naive Bayes assumption) assigns probability 0.97 to example 225 because both features fail for $\theta = \{M/225\}$. Hence

$$\begin{aligned}
p(Mutagenetic(225) = true, b_1\theta = false, b_2\theta = false) &= p(b_1\theta = false|Mutagenetic(225) = true)\\
&\times\ p(b_2\theta = false|Mutagenetic(225) = true)\\
&\times\ p(Mutagenetic(225) = true)\\
&= 0.99 \times 0.62 \times 0.31 \approx 0.19
\end{aligned}$$

and

$$p(Mutagenetic(225) = false, b_1\theta = false, b_2\theta = false) = 0.79{\times}0.01{\times}0.68 \approx 0.005$$

This yields

$$P(Mutagenetic(225) = true \mid b_1\theta = false, b_2\theta = false) = \frac{0.19}{0.19 + 0.005} \approx 0.97$$

4.5.1 Structure and Parameter Learning

Next, we outline typical solutions to the problems of structure and parameter learning in the setting of probabilistic learning from entailment (as well as interpretation).

1. The problem of parameter estimation is concerned with estimating the values of the parameters λ of a fixed probabilistic program $H = (L, \lambda)$ that best explains the examples $\mathcal{E}$. So, λ is a set of parameters and can be represented as a vector. To measure the extent to which a model fits the data, one usually employs the likelihood of the data, i.e. $P(\mathcal{E}|L, \lambda)$, though other scores or variants could be used as well.

   When all examples are fully observable, maximum likelihood reduces to frequency counting. In the presence of missing data, however, the maximum likelihood estimate typically cannot be written in closed form. It is a numerical optimization problem, and all known algorithms involve nonlinear optimization The most commonly adapted technique for probabilistic logic learning is the Expectation-Maximization (EM) algorithm [Dempster et al., 1977, McLachlan and Krishnan, 1997]. EM is based on the observation that learning would be easy (*i.e.*, correspond to frequency counting), if the values of all the random variables would be known. Therefore, it estimates these values, maximizes the likelihood based on the estimates, and then iterates. More specifically, EM assumes that the parameters have been initialized (*e.g.*, at random) and then iteratively performs the following two steps until convergence:

(a) **(E-Step):** On the basis of the observed data and the present parameters of the model, it computes a distribution over all possible completions of each partially observed data case.

(b) **(M-Step):** Treating each completion as a fully observed data case weighted by its probability, it computes the improved parameter values using (weighted) frequency counting. The frequencies over the completions are called the *expected counts*.

2. What if we need to learn both the structure L and the parameters λ of the probabilistic program $H = (L, \lambda)$ from the data? Often, further information is given as well. As in ILP, the additional knowledge can take various different forms, including a language bias that imposes restrictions on the syntax of L, and an initial hypothesis (L, λ) from which the learning process can start.

   Nearly all (score-based) approaches to structure learning perform a heuristic search through the space of possible hypotheses. Typically, hill-climbing or beam-search is applied until the hypothesis satisfies the logical constraints and the $score(H, \mathcal{E})$ is no longer improving. The steps in the search-space are typically made using refinement operators, see Section 2.7.

   Logical constraints often require that the possible examples are covered in the logical sense. For instance, when learning stochastic logic programs from entailment, the possible example clauses must be entailed by the logic program.

4.5.2 Example: Extending FOIL

Building on [Landwehr et al., 2005], we will next illustrate a promising, alternative approach with less computational complexity, which adapts FOIL [Quinlan and Cameron-Jones, 1995] with the conditional likelihood as described in Equation (4.1) as the scoring function $score(L, \lambda, \mathcal{E})$. This idea has been followed with nFOIL, see [Landwehr et al., 2005] for more details.

Given a training set $\mathcal{E}$ containing positive and negative examples (i.e. true and false ground facts), this algorithm stays in the learning from possible examples only to induce a probabilistic logical model to distinguish between the positive and negative examples. It computes Horn clause features $b_1, b_2, \ldots$ in an outer loop. It terminates when no further improvements in the score are obtained, i.e, when $score(\{b_1, \ldots, b_i\}, \lambda_i, \mathcal{E}) < score(\{b_i, \ldots, b_{i+1}\}, \lambda_{i+1}, \mathcal{E})$, where

λ denotes the maximum likelihood parameters. A major difference with FOIL is, however, that the covered positive examples are not removed. The inner loop is concerned with inducing the next feature b_{i+1} top-down, *i.e.*, from general to specific. To this aim it starts with a clause with an empty body, e.g., $Mutagenic(M)$. This clause is then specialized by repeatedly adding atoms to the body, *e.g.*, $Mutagenic(M) \leftarrow Bond(M, A, 1)$, $Muta(M) \leftarrow Bond(M, A, 1), Atom(M, A, c, 22, \_)$, *etc.* For each refinement $b^{'}_{i+1}$ we then compute the maximum-likelihood parameters $\lambda^{'}_{i+1}$ and $score(\{b_1, \ldots, b^{'}_{i+1}\}, \ldots, \mathcal{E})$. The refinement that scores best, say b_{i+1}, is then considered for further refinement and the refinement process terminates when

$$score(\{b_1, \ldots, b_{i+1}\}, \lambda_{i+1}, \mathcal{E}) < score(\{b_1, \ldots, b^{''}_{i+1}\}, \lambda^{''}_{i+1}, \mathcal{E})$$

It has been experimentally found that nFOIL performs well compared to other ILP systems on traditional ILP benchmark data sets. mFOIL and Aleph, two standard ILP systems, are never significantly better than nFOIL (paired sampled t-test, $p = 0.05$). nFOIL achieves significantly higher predictive accuracies than mFOIL on Alzheimer amine, toxic, and acetyl. Compared to Aleph, nFOIL achieves significantly higher accuracies on Alzheimer amine and acetyl (paired sampled t-test, $p = 0.05$). For more details, we refer to [Landwehr et al., 2005].

4.5.3 Example: Stochastic Logic Programming

Cussens [2001] and Sato and Kameya [2001], solve the parameter estimation problem for stochastic logic programs respectively PRISM programs, and Muggleton [2000a, 2002] presents an approach to structure learning of stochastic logic programs: adding one clause at a time to an existing stochastic logic program. The essentially use equation (4.4) as covers relation and, hence, employ the learning from entailment setting while making use of the semantics of probabilistic proofs. Here, the examples are ground atoms entailed by the target stochastic logic program. However, the Naive Bayes framework studied prior to this example has a much lower computational complexity. Also, learning stochastic logic programs from atoms only is much harder than learning them from proofs because atoms carry much less information than proofs.

4.6 Learning from Interpretations

The learning from interpretations setting [De Raedt and Dzeroski, 1994] upgrades boolean concept-learning in computational learning theory [Valiant, 1984]. When learning from interpretations, the examples are Herbrand interpretations[3] and a hypothesis H covers an example e with respect to the background theory $\mathcal{B}$ if and only if $\mathcal{B} \cup e$ is a model of H.

As an example, consider the interpretation I, which is the union of B and e:

[3]Recall that Herbrand interpretations are sets of true ground facts and they completely describe a possible situation.

$$B = \{Father(henry, bill), Father(alan, betsy), Father(alan, benny),\ Father(brian, bonnie), Father(bill, carl),$$
$$e = \{Carrier(alan), Carrier(ann), Carrier(betsy).\}$$

The interpretation I is covered by the clause H:

$$Carrier(X) \quad :- \quad Mother(M, X), Carrier(M), Father(F, X), Carrier(F).$$

because I is a model of C, *i.e.*, for all substitutions θ such that $body(C)\theta \subseteq I$, it holds that $head(C)\theta \in I$.

The key difference between learning from interpretations and learning from entailment is that interpretations carry much more, even complete information. Indeed, when learning from entailment, an example can consist of a single fact, whereas when learning from interpretations, all facts that hold in the example are known. Therefore, learning from interpretations is typically easier and computationally more tractable than learning from entailment, *c.f.*, [De Raedt, 1997].

ILP systems that learn from interpretations work in a similar fashion as those that learn from entailment. There is, however, one crucial difference and it concerns the generality relationship: When learning from entailment, G is more general than S if and only if $G \models S$,whereas when learning from interpretations, when $S \models G$. Another difference is that learning from interpretations is well suited for learning from positive examples only. For this case, a complete search of the space ordered by θ-subsumption is performed until all clauses cover all examples [De Raedt and Dehaspe, 1997].

4.7 Learning from Probabilistic Interpretations

The large majority of statistical relational learning techniques proposed so far fall into the learning from interpretations setting including parameter estimation of probabilistic logic programs [Koller and Pfeffer, 1997], learning of probabilistic relational models [Getoor et al., 2002], parameter estimation of relational Markov models [Taskar et al. 2002], learning of object-oriented Bayesian networks [Bangs et al., 2001], learning relational dependency networks [Neville and Jensen, 2004], learning logic programs with annotated disjunctions [Vennekens et al., 2004, Riguzzi, 2004] and most recently, learning Bayesian logic programs [Kersting et. al. 2000,2005].

In order to integrate probabilities in the learning from interpretations setting, we need to find a way to assign probabilities to interpretations covered by an annotated logic program. In the past few years, this issue has received a lot of attention and various different approaches have been developed such as probabilistic-logic programs [Ngo and Haddawy, 1997], probabilistic relational models [Pfeffer, 2000], relational Baysian networks Jager [1997], and Bayesian logic programs [Kersting, 2000, Kersting and De Raedt, 2001b]. Here, we focus on two methods:

1. *Undirected Graphical Model:* Domingos and Richardsons [2004] Markov logic networks (MLNs) and
2. *Directed Graphical Model:* Kersting and De Raedt's [2001b] Bayesian logic programs.

4.7.1 Example: Markov Logic Networks

Markov logic networks combine Markov networks [Pearl, 1991], which represent probability distributions over propositional interpretations, with first order logic. The idea underlying Markov logic networks is to view logical formulas as soft constraints on the set of possible worlds, *i.e.*, as *soft constraints on interpretations*: if a world violates one formula, it is less probable but not necessarily impossible as in classical logic. The fewer formulas a world violates, the more probable it is. In a Markov logic network, this is realized by associating a weight with each formula that reflects how strong the constraint is. More precisely, a Markov logic network consists of weighted first-order predicate logic formulae $\Sigma = \{C_1, C_2, \ldots, C_m\}$. The weights w_C of a formula C specify a bias for *ground instances to be true in a logical model.*

Consider the following example taken from [Richardson and Domingos, 2005]. Friends-smokers is a small Markov logic network that calculates the probability of a person P having lung cancer $Ca(P)$ based whether or not a person or her friends $Fr(P, P')$ smokes $Sm(P)$ respectively $Sm(P')$. This can be encoded using the following Markov logic formulas:

$$\begin{array}{lll} 1.5: & \forall X: & Sm(X) \Rightarrow Ca(X) \\ 1.1: & \forall X, Y: & Fr(X, Y) \Rightarrow (Sm(X) \iff Sm(Y)) \end{array}$$

For a given finite domain (roughly speaking a finite set of constants) $D = \{d_1, d_2, \ldots, d_n\}$, the Markov logic network defines a probability distribution over interpretations I over domain D and the relations occurring in the Markov logic network via

$$P(I|H, \mathcal{B}) = \frac{1}{Z(I)} \prod_{C \in H \cup \mathcal{B}} e^{n_C(I).w_C} = \frac{1}{Z(I)} \prod_{C \in H \cup \mathcal{B}} \phi_C(I)^{n_C(I)} \tag{4.2}$$

where $n_C(I)$ is the number of true groundings of C in I, $\phi_C(I) = e^{w_C}$, and $\mathcal{B}$ is a possible background theory.

Markov logic networks can be viewed as proving templates for constructing Markov networks. Given a set D constants,

- the nodes correspond to the ground atoms in the Herbrand base of the corresponding set of formulas Σ
- and there is an edge between two nodes if and only if the corresponding ground atoms appear together in at least one grounding of one formula $C_i \in \Sigma$.

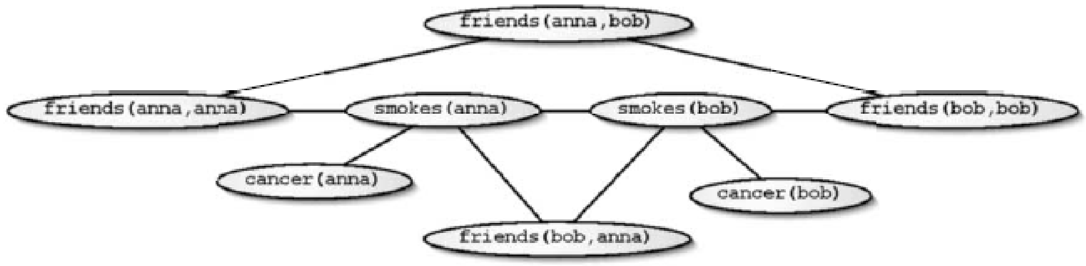


Figure 4.5: The Markov network induced by the friends-smoker Markov logic network assuming anna and bob as constants.

Assuming *anna* and *bob* as constants, the friends-smoker Markov logic network induces the Markov network in Figure 4.5.

Note that given different sets of constants, the Markov logic network will produce different Markov networks. From Equation (4.2), we can see that an example e consists of

- *a logical part*, which is a Herbrand interpretation of the annotated logic program, and
- *a probabilistic part*, which is a partial state assignment of the random variables occurring in the logical part.

To see this, consider that a possible example I in the friends-smokers domain is

$$\begin{array}{lll} Friends(anna, bob) = true, & Friends(bob, anna) = true, & Friends(anna, anna) = ?, \\ Friends(bob, bob) = true, & Smokes(anna) = false, & Smokes(bob) = ?, \\ Cancer(anna) = ?, & Cancer(bob) = false & \end{array}$$

where ? denotes an unobserved state. The covers relation for e can now be computed using any Markov network inference engine based on Equation (4.2).

Nearly all (score-based) approaches to structure learning perform a heuristic search through the space of possible hypotheses. Typically, hill-climbing or beam-search is applied until the hypothesis satisfies the logical constraints and the $score(H, \mathcal{E})$ is no longer improving. The steps in the search-space are typically made using refinement operators, see Section 2.7.

As described in Section 4.5, logical constraints often require that the possible examples are covered in the logical sense. For instance, when learning Markov logic networks, the possible interpretations must be models of the underlying logic program. Thus, for a probabilistic program $H = (\mathcal{L}_H, \lambda_H)$ and a background theory $\mathcal{B} = (\mathcal{B}, \lambda_B)$ it holds that $\forall e_p \in \mathcal{E}_p : \ p(e|H, \mathcal{B}) > 0$ if and only if $covers(e, \mathcal{L}_H, \mathcal{B}) = 1$, where $\mathcal{L}_H$ (respectively $\mathcal{B}$) is the underlying logic program (logical background theory) and $covers(e, \mathcal{L}_H, \mathcal{B})$ is the purely logical covers relation, which is either 0 or 1.

Kok and Domingos [2005] proposed a beam-search based approach for learning clausal Markov logic networks from possible examples only. Recall that a clausal Markov logic program consists of weighted clauses, *i.e.*, disjunction of literals. The clauses without associated weights constitute a clausal program L, and the weights the parameters λ. Starting with some initial clausal Markov logic network $H = (L, \lambda)$, the parameters maximizing $score(L, \lambda, \mathcal{E})$ are computed. Then, refinement operators (Section 2.7) generalizing respectively specializing L are used to to compute all neighbours of L in the hypothesis space. Literals are added and deleted, and signs of literals are flipped. Each neighbour is scored, yielding new hypotheses (L', λ'). To speed-up scoring, Kok and Domingos employ a variant of the pseudo-log-likelihood

$$\sum i = 1^n \log p(X_i = x_i | MB_x(X_i)$$

where x is the Herbrand base, x_i is the i^{th} ground atoms truth value, and $MB_x(X_i)$ is the state of $X_i s$ Markov blanket in the data, where, the Markov blanket of a node is its set of neighbouring nodes. The b best ones with $score(L', \lambda', \mathcal{E}) > score(L, \lambda, \mathcal{E})$ are kept. On these b best ones, the refining and scoring process is iteratively applied again until no new clauses improve the score or a maximal number of literals is reached. The clause with highest score in all iterations is added to H, and the process is continued until no improvement in score of the current best hypothesis is obtained.

4.7.2 Example: Bayesian Logic Programs

Recall from Chapter **??** that a Bayesian network specifies a joint probability distribution over a finite set of random variables and consists of two components: (1) a qualitative or logical one that encodes the local influences among the random variables using a directed acyclic graph, and (2) a quantitative one that encodes the probability densities over these local influences.

Despite these interesting properties, Bayesian networks also have a major limitation: they are essentially propositional representations.

For an illustrative example, imagine the task of modeling the localization of genes/proteins. When using a Bayesian network, every gene is a single random variable. There is no way of formulating general probabilistic regularities among the localizations of the genes such as the localization L of gene G is influenced by the localization L' of another gene G' that interacts with G. The propositional nature and limitation of Bayesian networks are similar to those of traditional attribute-value learning techniques, which have motivated work on upgrading these techniques within ILP. This in turn also explains the interest in upgrading Bayesian networks towards using first order logical representations.

Bayesian logic programs (BLPs) unify Bayesian networks with (definite clause) logic programming, which allows one to overcome the propositional character of Bayesian networks and the purely logical nature of logic programs. From a knowledge representation point of view, BLPs can be distinguished from alternative frameworks by having both

1. *Logic programs* (i.e. definite clause programs, which are sometimes called pure Prolog programs) as well as

2. it Bayesian networks as an immediate special case.

This is realized through the use of a small but powerful set of primitives. Indeed, similar to that for markov logic networks, the underlying idea of BLPs is to establish a one-to-one mapping between

1. ground atoms in the least Herbrand model ($MM(\Sigma)$ and random variables, and

2. between the immediate consequence operator and the direct influence relation.

Therefore, BLPs can also handle domains involving structured terms as well as continuous random variables. We will briefly describe BLPs, their representation language, their semantics, and a query-answering process, and present the learning of BLPs from data based on [Kersting 200, Kersting and De Raedt 2005].

The least Herbrand model of a BLP together with its direct influence relation is viewed as a (possibly infinite) Bayesian network. BLPs inherit the advantages of both Bayesian networks and definite clause logic, including

1. the strict separation of qualitative and quantitative aspects and consequently,

2. the introduction of a graphical representation, which stays close to the graphical representation of Bayesian networks.

Indeed, BLPs can naturally model any type of Bayesian network (including those involving continuous variables) as well as any type of 'pure' Prolog program (including those involving functors). In fact, BLPs can model hidden Markov models and stochastic grammars, and cal also be related to other first order extensions of Bayesian networks.

The framework for learning BLPs is an instance of the probabilistic learning from interpretations setting as described in Section 2.4.3. It is unifying

as it combines traditional Bayesian network learning and ILP principles. Therefore, the BLP framework builds upon

- Many of the results for Bayesian network learning from the Uncertainty in AI community, see e.g. [Heckerman, 1995],

- Many of the results from the ILP community,

- The EM and gradient ascent algorithms for parameter estimation,

- The general structure learning mechanisms from the field of Bayesian networks, and
- The clausal discovery and learning from interpretations settings from ILP for probabilistic learning from interpretations.

Consider the family disease example from page **??**. Now, imagine another totally separated family, which could be described by a similar Bayesian network. The graphical structure and associated conditional probability distribution for the two families are controlled by the same intensional regularities. But these overall regularities cannot be captured by a traditional Bayesian network. So, we need BLPs to represent these overall regularities. The central notion of BLPs is that of a Bayesian clause.

Definition 16 *A Bayesian (definite) clause C is an expression of the form $A|A_1, \ldots, A_n$ where $n \geq 0$, the $A, A_1, \leftarrow, A_n$ are Bayesian atoms and all Bayesian atoms are (implicitly) universally quantified. When $n = 0$, C is called a Bayesian fact and expressed as A. The differences between a Bayesian clause and a logical clause are:*

1. *the atoms $P(t_1, \ldots, t_k)$ and predicates P/k are Bayesian, which means that they have an associated finite set $S(P/k)$ of possible states (the ideas easily generalize to discrete and continuous random variables, modulo the well-known restrictions for Bayesian networks)*
2. *'— is used instead of ':- to highlight the conditional probability distribution.*

For instance, consider the Bayesian clause $C : BT(X) \mid MC(X), PC(X)$ where $S(BT/1) = \{a, b, ab, 0\}$ and $S(MC/1) = S(PC/1) = \{a, b, 0\}$. Intuitively, a Bayesian predicate P/k generically represents a set of random variables. More precisely, each Bayesian ground atom $\mathbf{g}$ over P/k represents a random variable over the states $S(\mathbf{g}) = S(P/k)$. For example, $BT(ann)$ represents the blood type of a person named *Ann* as a random variable over the states $\{a, b, ab, 0\}$. Apart from that, most logical notions carry over to BLPs. So, we can speak of Bayesian predicates, terms, constants, substitutions, propositions, ground Bayesian clauses, Bayesian Herbrand interpretations *etc.* For the sake of simplicity we will sometimes omit the term Bayesian as long as no ambiguities arise. We will assume that all Bayesian clauses are range-restricted, i.e., $Var(head(C)) \subseteq Var(body(C))$. Range restriction is often imposed in the database literature; it allows one to avoid the derivation of non-ground true facts. As already indicated while discussing Figure **??**, a set of Bayesian clauses encodes the qualitative or structural component of the BLPs. More precisely, ground atoms correspond to random variables, and the set of random variables encoded by a particular BLP corresponds to its *least Herbrand domain.* In addition, the direct influence relation corresponds to the immediate consequence.

Consider again, the following

$M(ann, dorothy).$

$PC(ann).$

$MC(ann).$

$MC(X) \mid M(Y, X), MC(Y), PC(Y).$

$BT(X) \mid MC(X), PC(X).$

$Ff(brian, dorothy).$

$PC(brian).$

$MC(brian).$

$PC(X) | F(Y, X), MC(Y), PC(Y).$

For each Bayesian predicate, the identity function is the combining rule. The conditional probability distributions associated with the Bayesian clauses $BT(X)|MC(X), PC(X)$ and $MC(X)|M(Y, X), MC(X), PC(Y)$ are represented as tables. The other distributions are correspondingly defined. The Bayesian predicates $M/2$ and $F/2$ have as possible states $\{true, false\}$.

$MC(X)$	$PC(X)$	$\Pr(BT(X))$
a	a	$(0.97, 0.01, 0.01, 0.01)$
b	a	$(0.01, 0.01, 0.97, 0.01)$
...	...	...
0	0	$(0.01, 0.01, 0.01, 0.97)$

$M(Y, X)$	$MC(Y)$	$PC(Y)$	$\Pr(MC(X))$
$true$	a	a	$(0.98, 0.01, 0.01)$
$true$	b	a	$(0.01, 0.98, 0.01)$
...	...	...	...
$false$	a	a	$(0.33, 0.33, 0.33)$
...	...	...	...

To keep the exposition simple, we will assume that $cpd(C)$ is represented as a table. More elaborate representations such as decision trees or rules would be possible too. The distribution $CPD(c)$ generically represents the conditional probability distributions associated with each ground instance $C\theta$ of the clause C.

In general, one has several clauses that may even make conflicting statements on conditional probability distributions. As another example, consider clauses $C_1 = BT(X)|MC(X)$ and $C_2 = BT(X)|PC(X)$ and assume corresponding substitutions $theta_i$ that ground the clauses C_i such that $head(C_1\theta_1) = head(C_2\theta_2)$. In contrast to $BT(X)|MC(X), PC(X)$, they specify $cpd(C_1\theta_1)$ and $cpd(c_2\theta_2)$, but not the desired distribution $\Pr(head(C_1\theta_1)|body(C_1) \cup body(C_2))$.

So called *combining rules* are the standard solution to obtain the distribution required.

Definition 17 *A combining rule is a function that maps finite sets of conditional probability distributions* $\{\Pr(A|A_{i_1}, ..., A_{i_{n_i}})|i = 1, ..., m\}$ *onto one (combined) conditional probability distribution* $\Pr(A|B_1, \ldots, B_k)$ *with* $\{B_1, \ldots, B_k\} \subseteq \cup_{i=1}^{m} A_{i_1}, \ldots, A_{i_{n_i}}$.

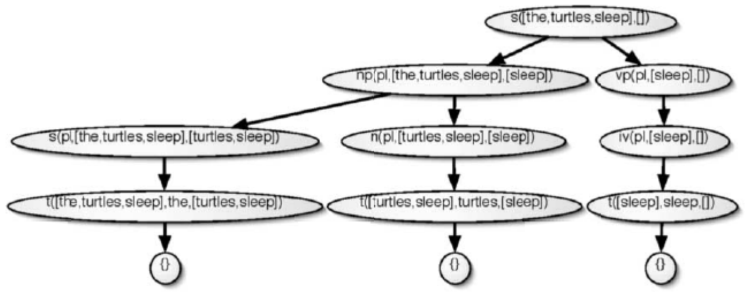

s([the,turtles,sleep],[])
np(pl,[the,turtles,sleep],[sleep])
vp(pl,[sleep],[])
s(pl,[the,turtles,sleep],[turtles,sleep])
n(pl,[turtles,sleep],[sleep])
iv(pl,[sleep],[])
t([the,turtles,sleep],the,[turtles,sleep])
t([turtles,sleep],turtles,[sleep])
t([sleep],sleep,[])
{}
{}
{}

Figure 4.6: An example proof tree (which is covered by the definite clause grammar on page 4.8).

We assume that for each Bayesian predicate P/k there is a corresponding combining rule $cr(P/k)$, such as noisy or (see e.g.[Jensen, 2001]) or average. The latter assumes $n_1 = \ldots = n_m$ and $S(A_{i_j}) = S(A_{k_j})$, and computes the average of the distributions over $S(A)$ for each joint state over $j \in S(A_{i_j})$.

4.8 Learning from Proofs

Because learning from entailment (with ground facts as examples) and interpretations occupy extreme positions with respect to the information the examples carry, it is interesting to investigate intermediate positions. Ehud Shapiros [1983] Model Inference System (MIS) fits nicely within the learning from entailment setting where examples are facts. However, to deal with missing information, Shapiro employs a clever strategy: MIS queries the users for missing information by asking them for the truth-value of facts. The answers to these queries allow MIS to reconstruct the trace or the proof of the positive examples.

When learning from proofs, the examples are ground proof-trees and an example e is covered by a hypothesis H with respect to the background theory $\mathcal{B}$ if and only if e is a proof-tree for $H \cup B$. At this point, there exist various possible forms of proof-trees. Here, we will, assume that the proof-tree is given in the form of a ground and-tree where the nodes contain ground atoms. More formally, a tree t is a proof-tree for a logic program Σ if and only if t is a rooted tree where for every node $n \in t$ with $children(n)$ satisfies the property that there exists a substitution θ and a clause $C \in \Sigma$ such that $n = head(C)\theta$. and $children(n) = body(C)$.

Consider the following definite clause grammar.

$$\begin{array}{lll}
Sentence(A, B) & :- & Noun\_phrase(C, A, D), Verb\_phrase(C, D, B). \\
Noun\_phrase(A, B, C) & :- & Article(A, B, D), Noun(A, D, C). \\
Verb\_phrase(A, B, C) & :- & Intransitive\_verb(A, B, C). \\
Article(singular, A, B) & :- & Terminal(A, a, B). \\
Article(singular, A, B) & :- & Terminal(A, the, B). \\
Article(plural, A, B) & :- & Terminal(A, the, B). \\
Noun(singular, A, B) & :- & Terminal(A, turtle, B). \\
Noun(plural, A, B) & :- & Terminal(A, turtles, B). \\
Intransitive\_verb(singular, A, B) & :- & Terminal(A, sleeps, B). \\
Intransitive\_verb(plural, A, B) & :- & Terminal(A, sleep, B). \\
Terminal([A|B], A, B). & &
\end{array}$$

It covers the proof tree shown in Figure 4.6. Proof-trees contain, as interpretations, a lot of information. Indeed, they contain instances of the clauses that were used in the proofs. Therefore, it may be hard for the user to provide these type of examples. Even though this is generally true, there exist specific situations for which this is feasible. Indeed, consider tree banks such as the UPenn Wall Street Journal corpus [Marcus et al., 1994], which contain parse trees. These trees directly correspond to the proof-trees we talk about. Another example is explanation-based learning (EBL) [Ellman, 1989, Mooney and Zelle, 1994]. It uses an existing domain theory to deductively explain an example (explanation step) in terms of a proof-tree and variablizes the explanation, *i.e.*, generalizes the proof as far as possible while maintaining its correctness (generalization step).

In the learning from proofs setting, one could turn all the proof-trees (corresponding to positive examples) into a set of ground clauses, which would constitute the initial theory. This theory can then be generalized by taking the least general generalization (under θ-subsumption) of pairwise clauses. Of course, care must be taken that the generalized theory does not cover negative examples.

4.9 Probabilistic Proofs

To define probabilities on proofs, ICL [Poole, 1993], PRISMs [Sato, 1995, Sato and Kameya, 2001], and stochastic logic programs [Eisele, 1994, Muggleton, 1996, Cussens, 2001] attach probabilities to facts (respectively clauses) and treat them as stochastic choices within resolution. Relational Markov models [Anderson et al., 2002] and logical hidden Markov models [Kersting 2000], can be viewed as a simple fragment of them, where heads and bodies of clauses are singletons only, so-called iterative clauses. We will illustrate probabilistic learning from proofs using stochastic logic programs. For a discussion of the close relationship among stochastic logic programs, ICL, and PRISM, we refer to [Cussens, 2005].

Stochastic logic programs are inspired on stochastic context free grammars [Abney, 1997, Manning and Schutze, 1999]. The analogy between context free grammars and logic programs is that

1. grammar rules correspond to definite clauses,
2. sentences (or strings) correspond to atoms, and
3. productions correspond to derivations.

Furthermore, in stochastic context-free grammars, the rules are annotated with probability labels in such a way that the sum of the probabilities associated to the rules defining a non-terminal is 1.0 (see this relation with the closed world assumption discussed in Section 1.3.7).

Eisele and Muggleton have exploited this analogy to define stochastic logic programs. These are essentially definite clause programs, where each clause C has an associated probability label p_C such that the sum of the probabilities associated to the rules defining any predicate is 1.0 (though Cussens [1999] considered less restricted versions as well). This framework allows ones to assign probabilities to proofs for a given predicate q given a stochastic logic program $H \cup B$ in the following manner. Let D_q denote the set of all possible ground proofs for atoms over the predicate q. For simplicity reasons, it will be assumed that there is a finite number of such proofs and that all proofs are finite (but again see [Cussens, 1999] for the more general case). Now associate to each proof $t_q \in D_q$ the probability

$$v_t = \prod_C p_C^{n_{C,t}}$$

where the product ranges over all clauses C and $n_{C,t}$ denotes the number of times clause C has been used in the proof t_q. For stochastic context free grammars, the values v_t correspond to the probabilities of the production.

However, the **difference** between context free grammars and logic programs is that in grammars two rules of the form $n \leftarrow q, n_1, \ldots, n_m$ and $q \leftarrow q_1, \ldots, q_k$ always resolve to give $n \leftarrow q_1, \ldots, q_k, n_1, \ldots, n_m$, **whereas resolution may fail due to unification**. Therefore, the probability of a proof tree t in D_q, *i.e.*, a successful derivation is

$$p(t|H, \mathcal{B}) = \frac{v_t}{\sum\limits_{s \in D_q} v_t} \tag{4.3}$$

The probability of a ground atom a is then defined as the sum of all the probabilities of all the proofs for that ground atom.

$$p(a|H, \mathcal{B}) = \sum_{s\ is\ a\ proof\ for\ a}^{s \in D_q} v_s \tag{4.4}$$

As an example, consider a stochastic variant of the definite clause grammar in the example on page 191 with uniform probability values for each predicate. The value v_u of the proof (tree) u in the example on the page 191 is $p(v_u) = \frac{1}{3} \times \frac{1}{2} \times \frac{1}{2} = \frac{1}{12}$. The only other ground proofs s_1, s_2 of atoms over the predicte $Sentence$ are those of $Sentence([a, turtle, sleeps], [])$ and $Sentence([the, turtle, sleeps], [])$. Both get the value $p(v_{s_1}) = p(v_{s_2}) = 1$. Because there is only one proof for each of the sentences,

$$p(Sentence([the, turtles, sleep], [])) = v_u = \frac{1}{3}$$

For stochastic logic programs, there are at least two natural learning settings.

1. Motivated by Equation (4.3), we can learn them from proofs. This makes structure learning for stochastic logic programs relatively easy, because proofs carry a lot of information about the structure of the underlying stochastic logic program. Furthermore, the learning setting can be considered as an extension of the work on learning stochastic grammars from proof-banks. It should therefore also be applicable to learning unification based grammars. We will present a probabilistic ILP approach within the learning from proofs setting subsequently in this section.

2. On the other hand, we can use Equation (4.4) as covers relation and, hence, employ the learning from entailment setting as mentioned in Section 4.5.3. Here, the examples are ground atoms entailed by the target stochastic logic program. Learning stochastic logic programs from atoms only is much harder than learning them from proofs because atoms carry much less information than proofs.

4.9.1 Example: Extending GOLEM

Given a training set $\mathcal{E}$ containing ground proofs as examples, one possible approach to learning from possible proofs only combines ideas from the early ILP system Golem [Muggleton and Feng, 1992] that employs Plotkins [1970] least general generalization(LGG) with bottom-up generalization of grammars and hidden Markov models [Stolcke and Omohundro, 1993]. The resulting algorithm employs the likelihood of the proofs $score(L, \lambda, \mathcal{E})$ as the scoring function. It starts by taking as L_0 the set of ground clauses that have been used in the proofs in the training set and scores it to obtain λ_0. After initialization, the algorithm will then repeatedly select a pair of clauses in L_i, and replace the pair by their LGG (lub) to yield a candidate L'. The candidate that scores best is then taken as $H_{i+1} = (L_{i+1}, \lambda_{i+1})$, and the process iterates until the score no longer improves.

One interesting issue is that strong logical constraints can be imposed on the LGG. These logical constraints directly follow from the fact that the example proofs should still be valid proofs for the logical component L of all hypotheses considered. Therefore, it makes sense to apply the LGG only to clauses that define the same predicate, that contain the same predicates, and whose (reduced)

LGG also has the same length as the original clauses. In general, the length (number of literals) of the LGG of m (ground) clauses of length at most n is n^m, see [Muggleton and Feng, 1992].

As an example, consider the following target stochastic logic program:

$$\begin{array}{ll}
1: & S(A,B) \leftarrow NP(Number,A,C), VP(Number,C,B).\\
\frac{1}{2}: & NP(Number,A,B) \leftarrow det(A,C), n(Number,C,B).\\
\frac{1}{2}: & NP(Number,A,B) \leftarrow pronom(Number,A,B).\\
\frac{1}{2}: & VP(Number,A,B) \leftarrow V(Number,A,B).\\
\frac{1}{2}: & VP(Number,A,B) \leftarrow V(Number,A,C), NP(D,C,B).\\
1: & Det(A,B) \leftarrow Term(A,the,B).\\
\frac{1}{4}: & N(s,A,B) \leftarrow Term(A,man,B).\\
\frac{1}{4}: & N(s,A,B) \leftarrow Term(A,apple,B).\\
\frac{1}{4}: & N(pl,A,B) \leftarrow Term(A,men,B).\\
\frac{1}{4}: & N(pl,A,B) \leftarrow Term(A,apples,B).\\
\frac{1}{4}: & V(s,A,B) \leftarrow Term(A,eats,B).\\
\frac{1}{4}: & V(s,A,B) \leftarrow Term(A,sings,B).\\
\frac{1}{4}: & V(pl,A,B) \leftarrow Term(A,eat,B).\\
\frac{1}{4}: & V(pl,A,B) \leftarrow Term(A,sing,B).\\
1: & Pronom(pl,A,B) \leftarrow Term(A,you,B).\\
1: & Term([A|B],A,B).
\end{array}$$

From this program, (independent) training sets of 50, 100, 200, and 500 proofs were generated. For each training set, 4 different random initial sets of parameters were tried. The learning algorithm was run on each data set starting from each of the initial sets of parameters. The algorithm stopped when a limit of 200 iterations was exceeded or a change in log-likelihood between two successive iterations was smaller than 0.0001. In all runs, the original structure was induced from the proof-trees. Moreover, already 50 proof-trees suffice to rediscover the structure of the original stochastic logic program. Further experiments with 20 and 10 samples respectively show that even 20 samples suffice to learn the given structure. Sampling 10 proofs, the original structure is rediscovered in one of five experiments. This supports that the learning from proof trees setting carries a lot information. Furthermore, the method scales well. Runs on two independently sampled sets of 1000 training proofs yield similar results: the original structure was learned in both cases. More details can be found in [De Raedt et al., 2005].

Other statistical relational learning frameworks that have been developed within the learning from proofs setting are relational Markov models [Anderson et al., 2002] and logical hidden Markov models [Kersting 2000?].

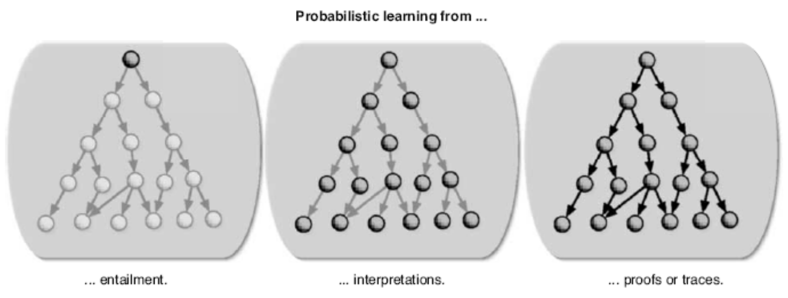


Figure 4.7: The level of information on the target probabilistic program provided by probabilistic ILP settings: shaded parts denote unobserved information. Learning from entailment provides the least information. Only roots of proof tree are observed. In contrast, learning from proofs or traces provides the most information. All ground clauses and atoms used in proofs are observed. Learning from interpretations provides an intermediate level of information. All ground atoms but not the clauses are observed.

4.10 Summary

The three learning settings are graphically compared in Figure 4.7.

References

[CKT91] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the Really Hard Problems Are. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence, IJCAI-91, Sidney, Australia*, pages 331–337, 1991.

[CM97] Stephen A. Cook and David G Mitchell. Finding hard instances of the satisfiability problem: A survey. In Du, Gu, and Pardalos, editors, *Satisfiability Problem: Theory and Applications*, volume 35, pages 1–17. American Mathematical Society, 1997.

[Gol79] A. Goldberg. Average case complexity of the satisfiability problem. In *Proceedings of the 4th Workshop on Automated Deduction*, pages 1–6, Austin, Texas, USA, 1979.

[NCdW97] Shan Wei Nienhuys-Cheng and Ronald de Wolf. *Foundations of Inductive Logic Programming.* Springer Verlag, Germany, 1997.

[SKC93] Bart Selman, Henry A. Kautz, and Bram Cohen. Local search strategies for satisfiability testing. In Michael Trick and David Stifler Johnson, editors, *Proceedings of the Second DIMACS Challange on Cliques, Coloring, and Satisfiability*, Providence RI, 1993.