Inference 12

Classical and Extended Inference

This chapter explores *inference,* the process of deducing facts from other known facts. Inference is useful for belief maintenance and is a cornerstone of rational thought. We start with *predicate logic,* and then explore *extended inference* systems—production systems, relaxation labeling, and active knowledge (procedures).

Predicate logic (Section 12.1) is a system for expressing propositions and for deriving consequences of facts. It has evolved over centuries, and many clear accounts describe predicate logic in its various forms [Mendelson 1964; Robinson 1965]. It has good formal properties, a nontrivial but automatable inference procedure, and a history of study in artificial intelligence. There are several "classical" extensions (modal logics, higher-order logics) which are studied in well-settled academic disciplines of metamathematics and philosophy. *Extended inference* (Section 12.2) is possible in automated systems, and is interesting technically and from an implementational standpoint.

A *production system* (Section 12.3) is a general rewriting system consisting of a set of *rewriting rules* ($A \rightarrow BC$ could mean "rewrite A as BC") and an executive program to apply rewrites. More generally, the rules can be considered "situation–action" pairs ("in situation A, do B and C"). Thus production systems can be used to control computational activities. Production systems, like semantic nets, embody powerful notions that can be used for extended inference.

Labeling schemes (Section 12.4) are unlike most inference mechanisms in that they often involve mathematical optimization in continuous spaces and can be implemented with parallel computation. Labeling is like inference because it establishes consistent "probability-like" values for "hypotheses" about the interpretation of entities.

Active knowledge (Section 12.5) is an implementation of inference in which each chunk of knowledge is a program. This technique goes far in the direction of "proceduralizing" the implementation of propositions. The design issues for such a system include the vocabulary of system primitives and their actions, mechanisms for implementing the flow of control, and overall control of the action of the system.

12.1 FIRST ORDER PREDICATE CALCULUS

Predicate logic is in many ways an attractive knowledge representation and inference system. However, despite its historical stature, important technical results in automated inference, and much research on inference techniques, logic has not dominated all aspects of mechanized inference. Some reasons for this are presented in Sections 12.1.6 and 12.2. The logical system that has received the most study is *first order predicate logic.* General theorem provers in this calculus are cumbersome for reasons which we shall explore. Furthermore, there is some controversy as to whether this logical system is adequate to express the reasoning processes used by human beings [Hayes 1977; Collins 1978; Winograd 1978; McCarthy and Hayes 1969]. We briefly describe some aspects of this controversy in Section 12.1.6. Our main purpose is to give the flavor of predicate calculus-based methods by describing briefly how automated inference can proceed with the formulae of predicate calculus expressed in the convenient *clause form.* Clause form is appealing for two reasons. First, it can be represented usefully in relational n-tuple or semantic network notation (Section 12.1.5). Second, the predicate calculus clause and inference system may be easily compared to production systems (Section 12.3).

12.1.1 Clause-Form Syntax (Informal)

In this section we describe the syntax of clause-form predicate calculus sentences. In the next, a more standard nonclausal syntax is described, together with a method for assigning meaning to grammatical logical expressions. Next, we show briefly how to convert from nonclausal to clausal syntax.

A *sentence* is a set of *clauses.* A clause is an ordered pair of sets of *atomic formulae*, or *atoms.* Clauses are written as two (possibly null) sets separated by an arrow, pointing from the *hypotheses* or *conditions* of the clause to its *conclusion.* The *null clause,* whose hypotheses and conclusion are both null, is written $\square$. For example, a clause could appear as

$$A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m$$

where the A's and B's are atoms. An atom is an expression

$$P(t_1, \ldots, t_j),$$

where P is a predicate symbol which "expects j arguments," each of which must be a *variable*, *constant symbol*, or a *term.* A term is an expression

$$f(t_1, \ldots, t_k)$$

where f is a *function symbol* which "expects k arguments," each of which may be a term. It is convenient to treat constant symbols alone as terms.

A careful (formal) treatment of the syntax of logic must deal with technical issues such as keeping constant and term symbols straight, associating the number of expected arguments with a predicate or function symbol, and assuring an infinite supply of symbols.

For example, the following are sentences of logic.

$\rightarrow$ Obscured(Backface(Block1))
Visible(Kidney) $\rightarrow$
Road(x), Unpaved(x) $\rightarrow$ Narrow(x)

12.1.2 Nonclausal Syntax and Logic Semantics (Informal)

Nonclausal Syntax

Clause form is a simplified but logically equivalent form of logic expressions which are perhaps more familiar. A brief review of non-clausal syntax follows.

The concepts of constant symbols, variables, terms, and atoms are still basic. A set of *logical connectives* provides unary and binary operators to combine atoms to form *well-formed formulae* (wffs). If A and B are atoms, then A is a wff, as is $\tilde{}A$ ("not A") $A \Longrightarrow B$ ("A implies B," or "if A then B"), $A \vee B$ ("A or B"), $A \wedge B$ ("A and B"), $A \Longleftrightarrow B$ ("A is equivalent to B," or "A if and only if B"). Thus an example of a wff is

Back(Face) $\vee$ (Obscured(Face)) $\Longrightarrow\tilde{}$ (Visible(Face))

The last concept is that of *universal* and *existential quantifiers*, the use of which is illustrated as follows.

($\forall$ x) (wff using "x" as a variable).
($\exists$ thing) (wff using "thing" as a variable).

A universal quantifier $\forall$ is interpreted as a conjunction over all domain elements, and an existential quantifier $\exists$ as a disjunction over all domain elements. Hence their usual interpretation as "for each element . . ." and "there exists an element. . . ."

Since a quantified wff is also a wff, quantifiers may be iterated and nested. A quantifier quantifies the "dummy" variable associated with it (x and thing in the examples above). The wff within the *scope* of a quantifier is said to have this quantified variable *bound* by the quantifier. Typically only wffs or clauses all of whose variables are bound are allowed.

Semantics

How does one assign meaning to grammatical clauses and formulae? The semantics of logic formulae (clauses and wffs alike) depends on an *interpretation* and on the meaning of connectives and quantifiers. An interpretation specifies the following.

1. A *domain* of individuals
2. A particular domain element is associated with each constant symbol
3. A function over the domain (mapping k individuals to individuals) is associated with each function symbol.
4. A relation over the domain (a set of ordered k-tuples of individuals) is associated with each predicate symbol.

The interpretation establishes a connection between the symbols in the representation and a domain of discourse (such as the entities one might see in an office or chest x-ray). To establish the truth or falsity of a clause or wff, a value of TRUE or FALSE must be assigned to each atom. This is done by checking in the world of the domain to see if the terms in the atom satisfy the relation specified by the predicate of the atom. If so, the atom is TRUE; if not, it is FALSE. (Of course, the terms, after evaluating their associated functions, ultimately specify individuals). For example, the atom

GreaterThan$(5,\pi)$

is true under the obvious interpretation and false with domain assignments such that

GreaterThan means "Is the author of"
5 means the book *Gone With the Wind*
π means Rin-Tin-Tin.

After determining the truth values of atoms, wffs with connectives are given truth values by using the *truth tables* of Table 12.1, which specify the semantics of the logical connectives. The relation of this formal semantics of connectives with the usual connectives used in language (especially "*implies*") is interesting, and one must be careful when translating natural language statements into predicate calculus.

The semantics of clause form expressions is now easy to explain. A sentence is the *conjunction* of its clauses. A clause

$$A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m$$

with variables $x_1, \ldots, x_k$ is to be understood

Table 12.1

A	B	$\tilde{}A$	$A \wedge B$	$A \vee B$	$A \Longrightarrow B$	$A \Longleftrightarrow B$
T	T	F	T	T	T	T
T	F	F	F	T	F	F
F	T	T	F	T	T	F
F	F	T	F	F	T	T

$$\forall\, x_1, \ldots, x_k,\ (A_1 \wedge \ldots \wedge A_n) \Longrightarrow (B_1 \vee \ldots \vee B_m).$$

The null clause is to be understood as a contradiction. A clause with no conditions is an assertion that at least one of the conclusions is true. A clause with null conclusion is a denial that the conditions (hypotheses) are true.

12.1.3 Converting Nonclausal Form to Clauses

The conversion of nonclausal to clausal form is done by applying straightforward rewriting rules, based on logic identities (ultimately the truth tables). There is one trick necessary, however, to remove existential quantifiers. *Skolem functions* are used to replace existentially quantified variables, according to the following reasoning.

Consider the wff

$(\forall x)((\exists y)(\text{Behind}(y, x)))$.

With the proper interpretation, this wff might correspond to saying "For any object x we consider, there is another object y which is behind x." Since the $\exists$ is within the scope of the $\forall$, the particular y might depend on the choice of x. The Skolem function trick is to remove the existential quantifier and use a function to make explicit the dependence on the bound universally quantified variable. The resulting wff could be

$(\forall x)\ (\text{Behind}(\text{SomethingBehind}(x), x))$

which might be rendered in English: "Any object x has another object behind it; furthermore, some Skolem function we choose to call SomethingBehind determines which object is behind its argument." This is a notational trick only; the existence of the new function is guaranteed by the existential quantification; both notations are equally vague as to the entity the function actually produces.

In general, one must replace each occurrence of an existentially quantified variable in a wff by a (newly created Skolem) function of all the universally quantified variables whose scope includes the existential quantifier being eliminated. If there is no universal quantifier, the result is a new function of no arguments, or a new constant.

$(\exists x)(\text{Red}(x))$,

which may be interpreted "Something is red," is rewritten as something like

Red(RedThing)

or

"Something is red, and furthermore let's call it RedThing."

The conversion from nonclausal to clausal form proceeds as follows (for more details, see [Nilsson 1971]). Remove all implication signs with the identity $(A \Longrightarrow B) \Longleftrightarrow ((\tilde{}\,A) \vee B)$. Use DeMorgan's laws (such as $\tilde{}(A \vee B) \Longleftrightarrow ((\tilde{}\,A) \wedge (\tilde{}\,B))$, and the extension to quantifiers, together with cancellation of double negations, to force negations to refer only to single predicate letters. Rewrite variables to give each quantifier its own unique dummy variable. Use Skolem functions to remove existential quantifiers. Variables are all now universally quantified, so eliminate the quantifier symbols (which remain implicitly), and rearrange the expression into conjunctive normal form (a conjunction of disjunctions.) The $\wedge$'s now connect disjunctive *clauses* (at last!). Eliminate the $\wedge$'s, obtaining from the original expression possibly several clauses.

At this point, the original expression has yielded multiple disjunctive clauses. Clauses in this form may be used directly in automatic theorem provers [Nilsson 1971]. The disjunctive clauses are not quite in the clause form as defined earlier, however; to get clauses into the final form, convert them into implications. Group negated atoms, reexpanding the scope of negation to include them all and converting the $\vee$ of $\tilde{}$'s into a $\tilde{}$ of $\wedge$'s. Reintroduce one implication to go from

$$B_1 \vee B_2 \ldots \vee B_m \vee (\tilde{}(A_1 \wedge A_2 \ldots \wedge A_n))$$

to

$$A_1 \wedge \ldots \wedge A_n \rightarrow B_1 \vee B_2 \ldots \vee B_m$$

To obtain the final form, replace the connectives (which remain implicitly) with commas.

12.1.4 Theorem Proving

Good accounts of the basic issues of automated theorem proving are given in [Nilsson 1971; Kowalski 1979; Loveland 1978]. The basic ideas are as follows. A sentence is *inconsistent*, or *unsatisfiable*, if it is false in every interpretation. Some trivially inconsistent sentences are those containing the null clause, or simple contradictions such as the same clause being both unconditionally asserted and denied. A sentence that is true in all interpretations is *valid.* Validity of individual clauses may be checked by applying the truth tables unless quantifiers are present, in which case an infinite number of formulae are being specified, and the truth status of such a clause is not algorithmically decidable. Thus it is said that first order predicate calculus is *undecidable.* More accurately, it is *semidecidable*, because any valid wff can be established as such in some (generally unpredictable) finite time. The validation procedure will run forever on invalid formulae; the rub is that one can never be sure whether it is running uselessly, or about to terminate in the next instant.

The notion of a *proof* is bound up with the notion of logical entailment. A clause C *logically follows* from a set of clauses S (we take S to *prove* C) if every interpretation that makes S true also makes C true. A formal proof is a sequence of inferences which establishes that C logically follows from S. In nonclausal predicate logic, inferences are rewritings of axioms and previously established formulae in accordance with *rules of inference* such as

Modus Ponens: From (A) and $(A \Longrightarrow B)$ infer (B)
Modus Tollens: From $(\tilde{}B)$ and $(A \Longrightarrow B)$ infer (A)
Substitution: e.g. From $(\forall x)$(Convex(x)) infer (Convex(Region31))
Syllogisms,

and so forth.

Automatic clausal theorem provers usually try to establish that a clause C logically follows from the set of clauses S. This is accomplished by showing the *unsatisfiability* of S and (C) taken together. This rather backward approach is a technical effect of the way that theorem provers usually work, which is to derive a contradiction.

The fundamental and surprising result that all true theorems are provable in finite time, and an algorithmic (but inefficient) way to find the proof, is due to Herbrand [Herbrand 1930]. The crux of the result is that although the domain of individuals who might participate in an interpretation may be infinite, only a finite number of interpretations need be investigated to establish unsatisfiability of a set of clauses, and in each only a finite number of individuals must be considered. A computationally efficient way to perform automatic inference was discovered by Robinson [Robinson 1965]. In it, a single rule of inference called *resolution* is used. This single rule preserves the *completeness* of the system (all true theorems are provable) and its *correctness* (no false theorems are provable).

The rule of resolution is very simple. Resolution involves matching a condition of one clause A with a conclusion of another clause B. The derived clause, called the *resolvent*, consists of the unmatched conditions and conclusions of A and B instantiated by the matching substitution. *Matching* two atoms amounts to finding a substitution of terms for variables which if applied to the atoms would make them identical.

Theorem proving now means resolving clauses with the hope of producing the empty clause, a contradiction.

As an example, a simple resolution proof goes as follows. Say it is desired to prove that a particular wastebasket is invisible. We know that the wastebasket is behind Brian's desk and that anything behind something else is invisible (we have a simpleminded view of the world in this little example). The givens are the wastebasket location and our naive belief about visibility:

$$\rightarrow \text{Behind}(\text{WasteBasket}, \text{DeskOf}(\text{Brian})) \tag{12.1}$$
$$\text{Behind}(\text{object},\text{obscurer}) \rightarrow \text{Invisible}(\text{object}) \tag{12.2}$$

Here Behind and Invisible are predicates, DeskOf is a function, Brian and WasteBasket are constants (denote particular specific objects), and object and obscurer are (universally quantified) variables. The negation of the conclusion we wish to prove is

$$\text{Invisible}(\text{WasteBasket}) \rightarrow \tag{12.3}$$

or, "Asserting the wastebasket is invisible is contradictory." Our task is to show this set of clauses is inconsistent, so that the invisibility of the wastebasket is proved. The resolution rule consists of matching clauses on opposite sides of the arrow which can be unified by a substitution of terms for variables. A substitution that works is:

Substitute WasteBasket for object and DeskOf(Brian) for obscurer in (12.2).

Then a cancellation can occur between the right side of (12.1) and the left side of (12.2). Another cancellation can then occur between the right side of (12.2) and the left side of (12.3), deriving the empty clause (a contradiction), Quod Erat Demonstrandum.

Anyone who has ever tried to do a nontrivial logic proof knows that there is searching involved in finding which inference to apply to make the proof terminate. Usually human beings have an idea of "what they are trying to prove," and can occasionally call upon some domain semantics to guide which inferences make sense. Notice that at no time in a resolution proof or other formal proof of logic is a specific interpretation singled out; the proof is about all possible interpretations. If deductions are made by appealing to intuitive, domain-dependent, semantic considerations (instead of purely syntactic rewritings), the deduction system is *informal*. Almost all of mathematics is informal by this definition, since normal proofs are not pure rewritings.

Many nonsemantic heuristics are also possible to guide search, such as trying to reduce the differences between the current formulae and the goal formula to be proved. People use such heuristics, as does the Logic Theorist, an early nonclausal, nonresolution theorem prover [Newell et al. 1963].

A basic resolution theorem prover *is* guaranteed to terminate with a proof if one exists, but usually resource limitations such as time or memory place an upper limit on the amount of effort one can afford to let the prover spend. As all the resolvents are added to the set of clauses from which further conclusions may be derived, the question of selecting which clauses to resolve becomes quite a vital one. Much research in automatic theorem proving has been devoted to reducing the search space of derivations for proofs [Nilsson 1980; Loveland 1970]. This has usually been done through heuristics based on formal aspects of the deductions (such as: make deductions that will not increase drastically the number of active clauses). Guidance from domain-dependent knowledge is not only hard to implement, it is directly against the spirit of resolution theorem proving, which attempts to do all the work with a uniform inference mechanism working on uninterpreted symbol strings. A moderation of this view allows the "intent" of a clause to guide its application in the proof. This can result in substantial savings of effort; an example is the treatment of "frame axioms" recommended by Kowalski (Section 13.1.4). Ad hoc, nonformalizable, domain-dependent methods are not usually welcome in automatic theorem-proving circles; however, such heuristics only guide the activity of a formal system; they do not render it informal.

12.1.5 Predicate Calculus and Semantic Networks

Predicate calculus theorem proving may be assisted by the addition of more relational structure to the set of clauses. The structure in a semantic net comes from *links* which connect *nodes*; nodes are accessed by following links, so the availability of information in nodes is determined by the link structure. Links can thus help by providing quick access to relevant information, given that one is "at" a particular node.

Although there are several ways of representing predicate calculus formulae in networks, we adopt here that of [Kowalski 1979; Deliyanni and Kowalski 1979]. The steps are simple:

1. Use a partition to represent the clause.
2. Convert all atoms to binary predicate atoms.
3. Distinguish between conditions and conclusions.

Recall that in Chapter 10, a partition is defined as a set of nodes and arcs in a graph. The internal structure of the partition cannot be determined from outside it. Partitioning extends the structure of a semantic net enough to allow unambiguous representations of all of first order predicate calculus.

The first step in developing the network representation for clauses is to convert each relation to a binary one. We distinguish between conditions and conclusions by using an additional bit of information for each arc. Diagrammatically, an arc is drawn with a double line if it is a condition and a single line if it is a conclusion. Thus the earlier example $S = \{(12.1), (12.2), (12.3)\}$ can be transformed into the network shown in Fig. 12.1.

This figure hints at the advantages of the network embedding for clauses: It is an indexing scheme. This scheme does not indicate which clauses to resolve next but can help reduce the possibilities enormously. If the most recent resolution involved a given clause with a given set of terms, other clauses which also have those terms will be represented by explicit arcs nearby in the network (this would *not* be true if the clauses were represented as a set). Similarly, other clauses involving the same predicate symbols are also nearby being indexed by those symbols. Again, this would not be true in the set representation. Thus the embedded network

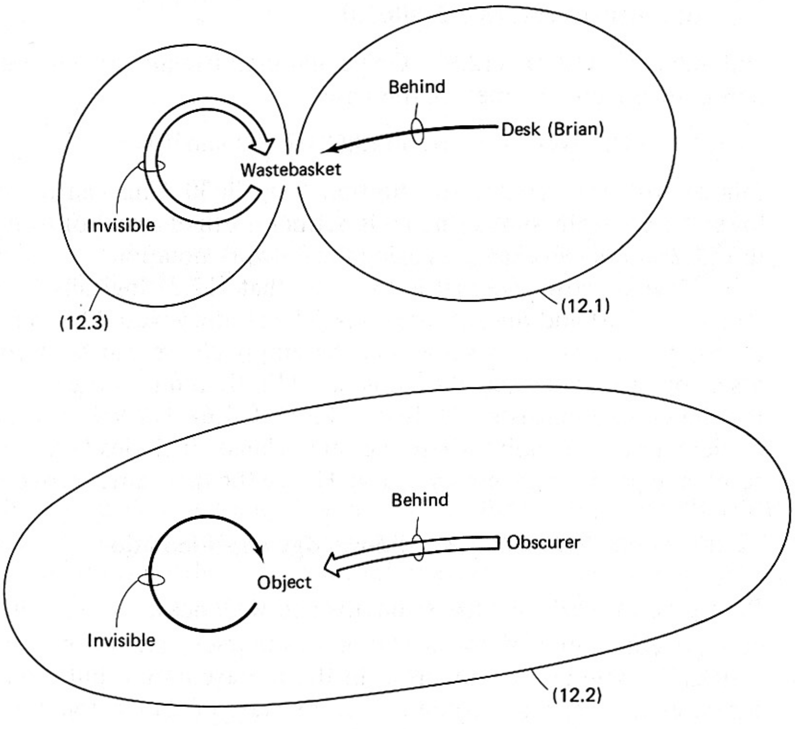

Fig. 12.1 Converting clauses to networks.

representation contains argument indices and predicate indices which can be extremely helpful in the inference process.

A very simple example illustrates the foregoing points. Suppose that S consists of the set of clauses

$$\text{SouthOf(river2},x\text{), NorthOf(river1},x\text{)} \rightarrow \text{Between(river1, river2, }x\text{)} \tag{12.4}$$
$$\rightarrow \text{SouthOf (}u\text{, silo30)} \tag{12.5}$$
$$\rightarrow \text{NorthOf (river1, silo30)} \tag{12.6}$$

Clause (12.5) might arise when it is determined that "silo30" is south of some feature in the image whose identity is not known. *Bottom up inference* derives new assertions from old ones. Thus in the example above the variable substitutions

$$u = \text{river2} \qquad x = \text{silo30}$$

match assertion (12.5) with the general clause (12.4) and allow the inference

$$\begin{aligned}&\text{NorthOf(river1, silo30)}\\ &\rightarrow \text{Between(river1, river2, silo30)}\end{aligned} \tag{12.7}$$

Consequently, use (12.6) and (12.7) to assert

$$\rightarrow \text{Between(river1, river2, silo 30)} \tag{12.8}$$

Suppose that this was not the case: that is, that

$$\text{Between(river1, river2, silo30)} \rightarrow \tag{12.9}$$

and that $S = \{(12.4), (12.9)\}$. One could then use *top-down inference*, which infers new denials from old ones. In this case

$$\text{NorthOf(river1,silo30), SouthOf(river2,silo30)} \rightarrow \tag{12.10}$$

follows with the variable substitution x = silo30. This can be interpreted as follows: "If x is really silo30, then it is neither north of river1 or south of river2." Figure 12.2 shows two examples using the network notation.

Now suppose the goal is to prove that (12.8) logically follows from (12.4) through (12.6) and the substitutions. The strategy would be to negate (12.8), add it to the data base, and show that the empty clause can be derived. Negating an assertion produces a denial, in this case (12.9), and now the set of axioms (including the denial) consists of $\{(12.4), (12.5), (12.6), (12.9)\}$. It is easy to repeat the earlier steps to the point where the set of clauses includes (12.8) and (12.9), which resolve to produce the empty clause. Hence the theorem is proved.

12.1.6 Predicate Calculus And Knowledge Representation

Pure predicate calculus has strengths and weaknesses as a knowledge representation system. Some of the seeming weaknesses can be overcome by technical "tricks." Some are not inherent in the representation but are a property of the common interpreters used on it (i.e., on state-of-the-art theorem provers). Some problems are relatively basic, and the majority opinion seems to be that first order

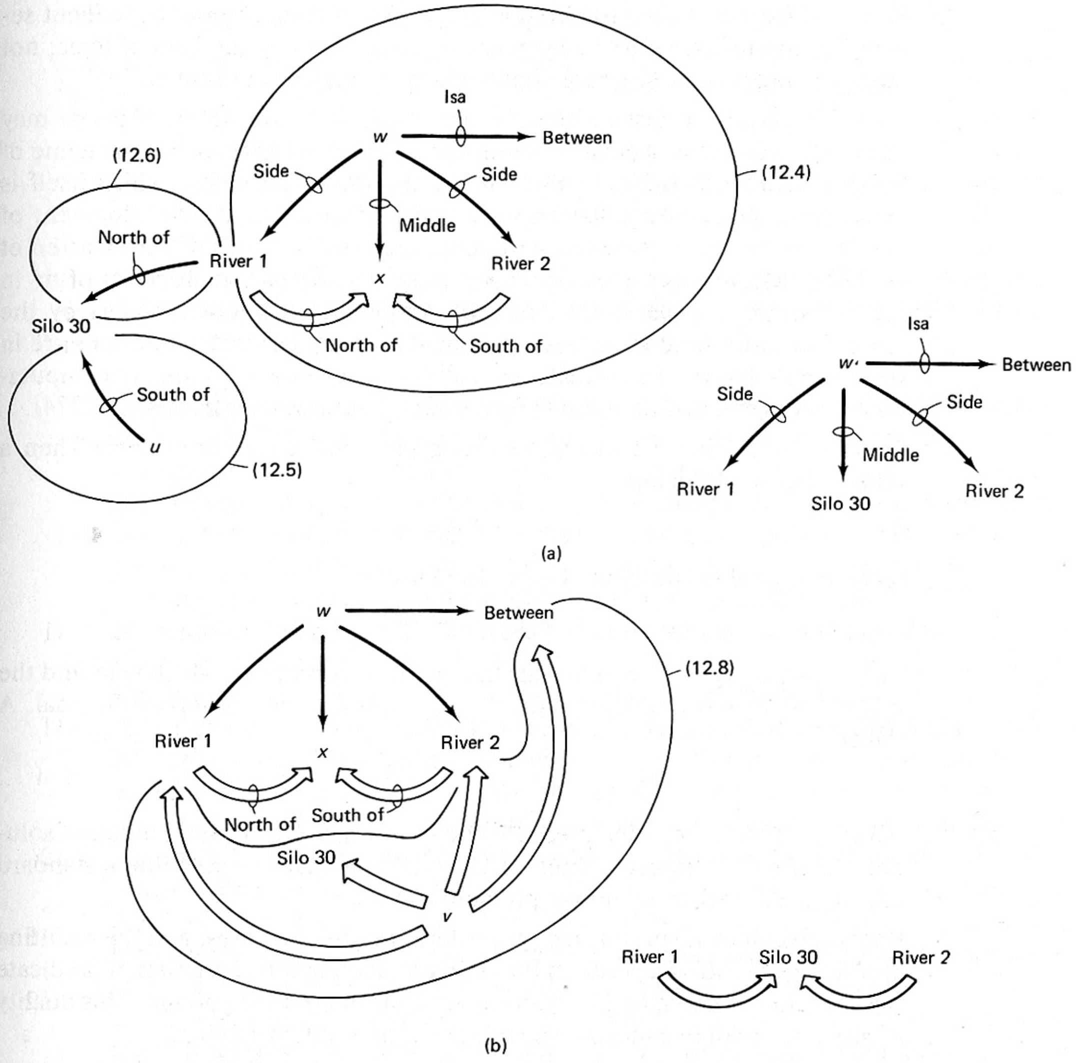

Fig. 12.2 Resolution using networks. (a) Bottom-up inference as a result of substitutions u = river2, x = silo30. (b) Top-down inference as a result of substitutions $w = v$, x = silo30.

predicate logic must be extended in order to become a representation scheme that is satisfactorily matched to the power of the deductive methods applied by human beings. Opinion is divided on the technical aspects of such enhancements. Predicate calculus has several strengths, some of which we list below.

1. Predicate logic is a well-polished gem, having been refined and studied for several generations. It waş designed to represent knowledge and inference. One knows what it means. Its model theory and proof theory are explicit and lucid [Hayes 1977; 1980].

2. Predicate logic can be considered a language with a machine-independent semantics; the meaning of the language is determined by the laws of logic, not the actual programming system upon which the logic is "executed."
3. Predicate calculus clauses with only one conclusion atom (Horn clauses) may be considered as "procedures," with the single conclusion being the name of the procedure and the conditions being the procedure body, which itself is made up of procedure calls. This view of logic leads to the development of predicate logic-based programming languages (such as PROLOG [Warren et al. 1977; McDermott 1980]). These programs exhibit nondeterminism in several interesting ways; the order of computations is not specified by the proof procedure (and is not restricted by it, either). Several resolutions are in general possible for any clause; the combinations determine many computations and several distinguishable forms of nondeterminism [Kowalski 1974].
4. Predicate logic may be interpreted as a problem-reduction system. Then a (Horn) clause of the form

   $$\rightarrow B$$

   represents a solved problem. One of the form

   $$A_1, \ldots, A_n \rightarrow$$

   with variables $x_1, \ldots, x_k$ is a goal statement, or command, which is to find the x's that solve the problems $A_1, \ldots, A_n$. Finding the x's solves the goal. A clause

   $$A_1, \ldots, A_n \rightarrow B$$

   is a solution method, which reduces the solution of B to a combination of solutions of A's. This interpretation of Horn clauses maps cleanly into a standard and–or goal tree formulation of problem solving.
5. Resolutions may be performed on the left or right of clauses, and the resulting derivation trees correspond, in the problem-solving interpretation of predicate calculus, to top-down and bottom-up versions of problem solving. This duality is very important in conceptualizing aspects of problem solving.
6. There is a uniform proof procedure for logic which is guaranteed to prove in finite time any true theorem (logic is semidecidable and complete). No false theorems are provable (logic is correct). These and other good formal properties are important when establishing formally the properties of a knowledge representation system.

Predicate calculus is not a favorite of everyone, however; some of the (perceived) disadvantages are given below, together with ways they might be countered.

1. Sometimes the axioms necessary to implement relatively common concepts are not immediately obvious. A standard example is "equality." These largely technical problems are annoying but not basic.

2. The "first order" in first order predicate calculus means that the system

does not allow clauses with variables ranging over an infinite number of predicates, functions, assertions and sentences (e.g., "All unary functions are boring" cannot be stated directly). This problem may be ameliorated by a notational trick; the situations under which predicates are true are indicated with a Holds predicate. Thus instead of writing On(block1, surface, situation1), write Holds (On(block1,surface), situation1). This notation allows inferences about many situations with only one added axiom. The "situational calculus" reappears in Section 12.3.1. Another useful notational trick is a Diff relation, which holds between two terms if they are syntactically different. There are infinitely many axioms asserting that terms are different; the actual system can be made to incorporate them implicitly in a well-defined way. The Diff relation is also used in Section 12.3.1.

3. The *frame problem* (so called for historical reasons and not related to the frames described in Section 10.3.1) is a classic bugbear of problem-solving methods including predicate logic. One aspect of this problem is that for technical reasons, it must be explicitly stated in axioms that describe actions (in a general sense a visual test is an action) that almost all assertions were true in a world state *remain* true in the new world state after the action is performed. The addition of these new axioms causes a huge increase in the "bureaucratic overhead" necessary to maintain the state of the world. Currently, no really satisfactory way of handling this problem has been devised. The most common way to attack this aspect of the frame problem is to use explicit "add lists" and "delete lists" ([Fikes 1977], Chapter 13) which attempt to specify exactly what changes when an action occurs. New true assertions are added and those that are false after an action must be deleted. This device is useful, but examples demonstrating its inadequacy are readily constructed. More aspects of the frame problem are given in Chapter 13.

4. There are several sorts of reasoning performed by human beings that predicate logic does not pretend to address. It does not include the ability to describe its own formulae (a form of "quotation"), the notion of defaults, or a mechanism for plausible reasoning. Extensions to predicate logic, such as modal logic, are classically motivated. More recently, work on extensions addressing the topics above have begun to receive attention [McCarthy 1978; Reiter 1978; Hayes 1977]. There is still active debate as to whether such extensions can capture many important aspects of human reasoning and knowledge within the model-theoretic system. The contrary view is that in some reasoning, the very *process* of reasoning itself is an important part of the semantics of the representation. Examples of such extended inference systems appear in the remainder of this chapter, and the issues are addressed in more detail in the next section.

12.2 COMPUTER REASONING

Artificial intelligence in general and computer vision in particular must be concerned with *efficiency* and *plausibility* in inference [Winograd 1978]. Computer-based knowledge representations and their accompanying inference processes often sacrifice classical formal properties for gains in control of the inference process and for flexibility in the sorts of "truth" which may be inferred.