Chapter 2

Logic and Resolution

One of the earliest formalisms for the representation of knowledge is *logic*. The formalism is characterized by a well-defined syntax and semantics, and provides a number of *inference rules* to manipulate logical formulas on the basis of their form in order to derive new knowledge. Logic has a very long and rich tradition, going back to the ancient Greeks: its roots can be traced to Aristotle. However, it took until the present century before the mathematical foundations of modern logic were laid, amongst others by T. Skolem, J. Herbrand, K. Gödel, and G. Gentzen. The work of these great and influential mathematicians rendered logic firmly established before the area of computer science came into being.

Already from the early 1950s, as soon as the first digital computers became available, research was initiated on using logic for problem solving by means of the computer. This research was undertaken from different points of view. Several researchers were primarily interested in the mechanization of mathematical proofs: the efficient automated generation of such proofs was their main objective. One of them was M. Davis who, already in 1954, developed a computer program which was capable of proving several theorems from number theory. The greatest triumph of the program was its proof that the sum of two even numbers is even. Other researchers, however, were more interested in the study of human problem solving, more in particular in heuristics. For these researchers, mathematical reasoning served as a point of departure for the study of heuristics, and logic seemed to capture the essence of mathematics; they used logic merely as a convenient language for the formal representation of human reasoning. The classical example of this approach to the area of theorem proving is a program developed by A. Newell, J.C. Shaw and H.A. Simon in 1955, called the *Logic Theory Machine*. This program was capable of proving several theorems from the Principia Mathematica of A.N. Whitehead and B. Russell. As early as 1961, J. McCarthy, amongst others, pointed out that theorem proving could also be used for solving non-mathematical problems. This idea was elaborated by many authors. Well known is the early work on so-called *question-answering systems* by J.R. Slagle and the later work in this field by C.C. Green and B. Raphael.

After some initial success, it soon became apparent that the inference rules known at that time were not as suitable for application in digital computers as hoped for. Many AI researchers lost interest in applying logic, and shifted their attention towards the development of other formalisms for a more efficient representation and manipulation of information. The breakthrough came thanks to the development of an efficient and flexible inference rule in 1965, named *resolution*, that allowed applying logic for automated problem solving by the

computer, and theorem proving finally gained an established position in artificial intelligence and, more recently, in the computer science as a whole as well.

Logic can directly be used as a knowledge-representation formalism for building expert systems; currently however, this is done only on a small scale. But then, the clear semantics of logic makes the formalism eminently suitable as a point of departure for understanding what the other knowledge-representation formalisms are all about. In this chapter, we first discuss the subject of how knowledge can be represented in logic, departing from propositional logic, which although having a rather limited expressiveness, is very useful for introducing several important notions. First-order predicate logic, which offers a much richer language for knowledge representation, is treated in Section 2.2. The major part of this chapter however will be devoted to the algorithmic aspects of applying logic in an automated reasoning system, and resolution in particular will be the subject of study.

2.1 Propositional logic

Propositional logic may be viewed as a representation language which allows us to express and reason with statements that are either *true* or *false*. Examples of such statements are:

‘A full-adder is a logical circuit’
‘10 is greater than 90’

Clearly, such statement need not be true. Statements like these are called *propositions* and are usually denoted in propositional logic by uppercase letters. Simple propositions such as P and Q are called *atomic propositions* or *atoms* for short. Atoms can be combined with so-called *logical connectives* to yield *composite propositions*. In the language of propositional logic, we have the following five connectives at our disposal:

negation:	$\neg$	(not)
conjunction:	$\wedge$	(and)
disjunction:	$\vee$	(or)
implication:	$\rightarrow$	(if then)
bi-implication:	$\leftrightarrow$	(if and only if)

For example, when we assume that the propositions G and D have the following meaning

$G =$ ‘A Bugatti is a car’
$D =$ ‘A Bugatti has 5 wheels’

then the composite proposition

$G \wedge D$

has the meaning:

‘A Bugatti is a car *and* a Bugatti has 5 wheels’

However, not all formulas consisting of atoms and connectives are (composite) propositions. In order to distinguish syntactically correct formulas that do represent propositions from those that do not, the notion of a well-formed formula is introduced in the following definition.

Definition 2.1 *A* well-formed formula *in propositional logic is an expression having one of the following forms:*

(1) An atom is a well-formed formula.

(2) If F is a well-formed formula, then $(\neg F)$ is a well-formed formula.

(3) If F and G are well-formed formulas, then $(F \wedge G)$, $(F \vee G)$, $(F \rightarrow G)$ and $(F \leftrightarrow G)$ are well-formed formulas.

(4) No other formula is well-formed.

EXAMPLE 2.1

Both formulas $(F \wedge (G \rightarrow H))$ and $(F \vee (\neg G))$ are well-formed according to the previous definition, but the formula $(\rightarrow H)$ is not.

In well-formed formulas, parentheses may be omitted as long as no ambiguity can occur; the adopted priority of the connectives is, in decreasing order, as follows:

$\neg \ \wedge \ \vee \ \rightarrow \ \leftrightarrow$

In the following, the term formula is used as an abbreviation when a well-formed formula is meant.

EXAMPLE 2.2

The formula $P \rightarrow Q \wedge R$ is the same as the formula $(P \rightarrow (Q \wedge R))$.

The notion of well-formedness of formulas only concerns the syntax of formulas in propositional logic: it does not express the formulas to be either *true* or *false*. In other words, it tells us nothing with respect to the semantics or meaning of formulas in propositional logic. The truth or falsity of a formula is called its *truth value*. The meaning of a formula in propositional logic is defined by means of a function $w : \mathrm{PROP} \rightarrow \{true, false\}$ which assigns to each proposition in the set of propositions PROP either the truth value *true* or *false*. Consequently, the information that the atom P has the truth value *true*, is now denoted by $w(P) = true$, and the information that the atom P has the truth value *false*, is denoted by $w(P) = false$. Such a function w is called an interpretation function, or an *interpretation* for short, if it satisfies the following properties (we assume F and G to be arbitrary well-formed formulas):

(1) $w(\neg F) = true$ if $w(F) = false$, and $w(\neg F) = false$ if $w(F) = true$.

(2) $w(F \wedge G) = true$ if $w(F) = true$ and $w(G) = true$; otherwise $w(F \wedge G) = false$.

(3) $w(F \vee G) = false$ if $w(F) = false$ and $w(G) = false$; in all other cases, that is, if at least one of the function values $w(F)$ and $w(G)$ equals *true*, we have $w(F \vee G) = true$.

(4) $w(F \rightarrow G) = false$ if $w(F) = true$ and $w(G) = false$; in all other cases we have $w(F \rightarrow G) = true$.

(5) $w(F \leftrightarrow G) = true$ if $w(F) = w(G)$; otherwise $w(F \leftrightarrow G) = false$.

Table 2.1: The meanings of the connectives.

F	G	$\neg F$	$F \wedge G$	$F \vee G$	$F \rightarrow G$	$F \leftrightarrow G$
true	*true*	*false*	*true*	*true*	*true*	*true*
true	*false*	*false*	*false*	*true*	*false*	*false*
-false	*true*	*true*	*false*	*true*	*true*	*false*
false	*false*	*true*	*false*	*false*	*true*	*true*

Table 2.2: Truth table for $P \rightarrow (\neg Q \wedge R)$.

P	Q	R	$\neg Q$	$\neg Q \wedge R$	$P \rightarrow (\neg Q \wedge R)$
true	*true*	*true*	*false*	*false*	*false*
true	*true*	*false*	*false*	*false*	*false*
true	*false*	*true*	*true*	*true*	*true*
true	*false*	*false*	*true*	*false*	*false*
false	*true*	*true*	*false*	*false*	*true*
false	*true*	*false*	*false*	*false*	*true*
false	*false*	*true*	*true*	*true*	*true*
false	*false*	*false*	*true*	*false*	*true*

These rules are summarized in Table 2.1. The first two columns in this table list all possible combinations of truth values for the atomic propositions F and G; the remaining columns define the meanings of the respective connectives. If w is an interpretation which assigns to a given formula F the truth value *true*, then w is called a *model* for F.

By repeated applications of the rules listed in table 2.1, it is possible to express the truth value of an arbitrary formula in terms of the truth values of the atoms the formula is composed of. In a formula containing n different atoms, there are 2^n possible ways of assigning truth values to the atoms in the formula.

EXAMPLE 2.3

Table 2.2 lists all possible combinations of truth values for the atoms in the formula $P \rightarrow (\neg Q \wedge R)$; for each combination, the resulting truth value for this formula is determined. Such a table where all possible truth values for the atoms in a formula F are entered together with the corresponding truth value for the whole formula F, is called a *truth table*.

Definition 2.2 *A formula is called a* valid *formula if it is true under all interpretations. A valid formula is often called a* tautology. *A formula is called* invalid *if it is not valid.*

So, a valid formula is true regardless of the truth or falsity of its constituent atoms.

EXAMPLE 2.4

The formula $((P \rightarrow Q) \wedge P) \rightarrow Q$ is an example of a valid formula. In the previous example we dealt with an invalid formula.

Definition 2.3 *A formula is called* unsatisfiable *or* inconsistent *if the formula is false under all interpretations. An unsatisfiable formula is also called a* contradiction. *A formula is called* satisfiable *or* consistent *if it is not unsatisfiable.*

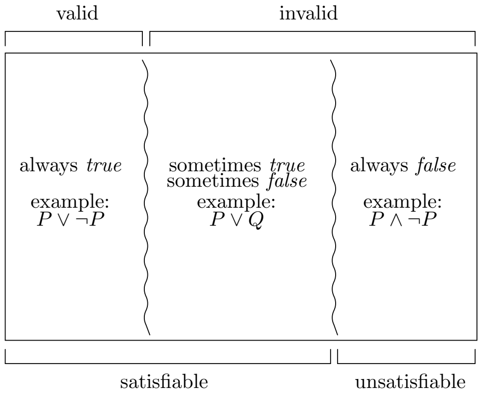


Figure 2.1: Relationship between validity and satisfiability.

Table 2.3: Truth table of $\neg(P \land Q)$ and $\neg P \lor \neg Q$.

P	Q	$\neg(P \land Q)$	$\neg P \lor \neg Q$
true	*true*	*false*	*false*
true	*false*	*true*	*true*
false	*true*	*true*	*true*
false	*false*	*true*	*true*

Note that a formula is valid precisely when its negation is unsatisfiable and vice versa.

EXAMPLE 2.5

The formulas $P \land \neg P$ and $(P \to Q) \land (P \land \neg Q)$ are both unsatisfiable.

Figure 2.1 depicts the relationships between the notions of valid, invalid, and satisfiable, and unsatisfiable formulas.

Definition 2.4 *Two formulas F and G are called* equivalent*, written as $F \equiv G$, if the truth values of F and G are the same under all possible interpretations.*

Two formulas can be shown to be equivalent by demonstrating that their truth tables are identical.

EXAMPLE 2.6

Table 2.3 shows that $\neg(P \land Q) \equiv \neg P \lor \neg Q$.

Using truth tables the logical equivalences listed in Table 2.4 can easily be proven. These equivalences are called *laws of equivalence*. Law (a) is called the *law of double negation*; the laws (b) and (c) are called the *commutative laws*; (d) and (e) are the so-called *associative*

Table 2.4: Laws of equivalence.

$\neg(\neg F) \equiv F$	(a)
$F \lor G \equiv G \lor F$	(b)
$F \land G \equiv G \land F$	(c)
$(F \land G) \land H \equiv F \land (G \land H)$	(d)
$(F \lor G) \lor H \equiv F \lor (G \lor H)$	(e)
$F \lor (G \land H) \equiv (F \lor G) \land (F \lor H)$	(f)
$F \land (G \lor H) \equiv (F \land G) \lor (F \land H)$	(g)
$F \leftrightarrow G \equiv (F \to G) \land (G \to F)$	(h)
$F \to G \equiv \neg F \lor G$	(i)
$\neg(F \land G) \equiv \neg F \lor \neg G$	(j)
$\neg(F \lor G) \equiv \neg F \land \neg G$	(k)

laws, and (f) and (g) are the *distributive laws*. The laws (j) and (k) are known as the *laws of De Morgan*. These laws often are used to transform a given well-formed formula into a logically equivalent but syntactically different formula.

In the following, a conjunction of formulas is often written as a set of formulas, where the elements of the set are taken as the conjunctive subformulas of the given formula.

EXAMPLE 2.7

The set $S = \{F \lor G, H\}$ represents the following formula: $(F \lor G) \land H$.

Truth tables can be applied to determine whether or not a given formula follows logically from a given set of formulas. Informally speaking, a formula logically follows from a set of formulas if it is satisfied by all interpretations satisfying the given set of formulas; we say that the formula is a logical consequence of the formulas in the given set. The following is a formal definition of this notion.

Definition 2.5 *A formula G is said to be a* logical consequence *of the set of formulas $F = \{F_1, \ldots, F_n\}$, $n \geq 1$, denoted by $F \vDash G$, if for each interpretation w for which $w(F_1 \land \cdots \land F_n) =$ true, we have $w(G) =$ true.*

EXAMPLE 2.8

The formula R is a logical consequence of the set of formulas $\{P \land \neg Q, P \to R\}$. Thus we can write $\{P \land \neg Q, P \to R\} \vDash R$.

Note that another way of stating that two formulas F and G are logically equivalent, that is, $F \equiv G$, is to say that both $\{F\} \vDash G$ and $\{G\} \vDash F$ hold. This tells us that the truth value of F and G are explicitly related to each other, which can also be expressed as $\vDash (F \leftrightarrow G)$.

Satisfiability, validity, equivalence and logical consequence are *semantic* notions; these properties are generally established using truth tables. However, for deriving logical consequences from of a set of formulas for example, propositional logic provides other techniques than using truth tables as well. It is possible to derive logical consequences by *syntactic* operations only. A formula which is derived from a given set of formulas then is guaranteed to be a logical consequence of that set if the syntactic operations employed meet certain conditions. Systems in which such syntactic operations are defined, are called (*formal*) *deduction systems*. Various sorts of deduction systems are known. An example of a deduction system is an *axiomatic system*, consisting of a formal language, such as the language of propositional logic described above, a set of *inference rules* (the syntactic operations) and a set of *axioms*. In Section 2.4 we shall return to the subject of logical deduction.

2.2 First-order predicate logic

In propositional logic, atoms are the basic constituents of formulas which are either *true* or *false*. A limitation of propositional logic is the impossibility to express general statements concerning similar cases. *First-order predicate logic* is more expressive than propositional logic, and such general statements can be specified in its language. Let us first introduce the language of first-order predicate logic. The following symbols are used:

- *Predicate symbols*, usually denoted by uppercase letters. Each predicate symbol has associated a natural number n, $n \geq 0$, indicating the number of arguments the predicate symbol has; the predicate symbol is called an *n-place* predicate symbol. 0-place or *nullary* predicate symbols are also called (*atomic*) *propositions*. One-place, two-place and three-place predicate symbols are also called *unary*, *binary* and *ternary* predicate symbols, respectively.

- *Variables*, usually denoted by lowercase letters from the end of the alphabet, such as x, y, z, possibly indexed with a natural number.

- *Function symbols*, usually denoted by lowercase letters halfway the alphabet. Each function symbol has associated a natural number n, $n \geq 0$, indicating its number of arguments; the function symbol is called *n-place*. Nullary function symbols are usually called *constants*.

- The *logical connectives* which have already been discussed in the previous section.

- Two *quantifiers*: the *universal quantifier* $\forall$, and the *existential quantifier* $\exists$. The quantifiers should be read as follows: if x is a variable, then $\forall x$ means 'for each x' or 'for all x', and $\exists x$ means 'there exists an x'.

- A number of *auxiliary symbols* such as parentheses and commas.

Variables and functions in logic are more or less similar to variables and functions in for instance algebra or calculus.

Before we define the notion of an atomic formula in predicate logic, we first introduce the notion of a term.

Definition 2.6 *A* term *is defined as follows:*

(1) A constant is a term.

(2) A variable is a term.

(3) If f is an n-place function symbol, $n \geq 1$, and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.

(4) Nothing else is a term.

So, a term is either a constant, a variable or a function of terms. Recall that a constant may also be viewed as a nullary function symbol. An atomic formula now consists of a predicate symbol and a number of terms to be taken as the arguments of the predicate symbol.

Definition 2.7 *An* atomic formula*, or* atom *for short, is an expression of the form $P(t_1, \ldots, t_n)$, where P is an n-place predicate symbol, $n \geq 0$, and $t_1, \ldots, t_n$ are terms.*

EXAMPLE 2.9

If P is a unary predicate symbol and x is a variable, then $P(x)$ is an atom. $Q(f(y), c, g(f(x), z))$ is an atom if Q is a ternary predicate symbol, c is a constant, f a unary function symbol, g a binary function symbol, and x, y and z are variables. For the same predicate symbols P and Q, $P(Q)$ is not an atom, because Q is not a term but a predicate symbol.

Composite formulas can be formed using the five connectives given in Section 2.1, together with the two quantifiers $\forall$ and $\exists$ just introduced. As was done for propositional logic, we now define the notion of a well-formed formula in predicate logic. The following definition also introduces the additional notions of free and bound variables.

Definition 2.8 *A* well-formed formula *in predicate logic, and the set of* free variables *of a well-formed formula are defined as follows:*

(1) An atom is a well-formed formula. The set of free variables of an atomic formula consists of all the variables occurring in the terms in the atom.

(2) Let F be a well-formed formula with an associated set of free variables. Then, $(\neg F)$ is a well-formed formula. The set of free variables of $(\neg F)$ equals the set of free variables of F.

(3) Let F and G be well-formed formulas and let for each of these formulas a set of free variables be given. Then, $(F \vee G)$, $(F \wedge G)$, $(F \rightarrow G)$ and $(F \leftrightarrow G)$ are well-formed formulas. The set of free variables of each of these last mentioned formulas is equal to the union of the sets of free variables of F and G.

(4) If F is well-formed formula and x is an element of the set of free variables of F, then both $(\forall x F)$ and $(\exists x F)$ are well-formed formulas. The set of free variables of each of these formulas is equal to the set of free variables of F from which the variable x has been removed. The variable x is called bound *by the quantifier $\forall$ or $\exists$.*

(5) Nothing else is a well-formed formula.

Note that we have introduced the notion of a formula in the preceding definition only from a purely syntactical point of view: nothing has been said about the meaning of such a formula.

Parentheses will be omitted from well-formed formulas as long as ambiguity cannot occur; the quantifiers then have a higher priority than the connectives.

Definition 2.9 *A well-formed formula is called a* closed formula*, or a* sentence*, if its set of free variables is empty; otherwise it is called an* open formula.

EXAMPLE 2.10

The set of free variables of the formula $\forall x \exists y (P(x) \rightarrow Q(y, z))$ is equal to $\{z\}$. So, only one of the three variables in the formula is a free variable. The formula $\forall x (P(x) \vee R(x))$ has no free variables at all, and thus is an example of a sentence.

In what follows, we shall primarily be concerned with closed formulas; the term formula will be used to mean a closed formula, unless explicitly stated otherwise.

In the formula $\forall x (A(x) \rightarrow G(x))$ all occurrences of the variable x in $A(x) \rightarrow G(x)$ are governed by the associated universal quantifier; $A(x) \rightarrow G(x)$ is called the *scope* of this quantifier.

EXAMPLE 2.11

The scope of the universal quantifier in the formula

$$\forall x (P(x) \rightarrow \exists y R(x, y))$$

is $P(x) \rightarrow \exists y R(x, y)$; the scope of the existential quantifier is the subformula $R(x, y)$.

In propositional logic, the truth value of a formula under a given interpretation is obtained by assigning either the truth value *true* or *false* to each of its constituent atoms according to this specific interpretation. Defining the semantics of first-order predicate logic is somewhat more involved than in propositional logic. In predicate logic, a structure representing the 'reality' is associated with the *meaningless* set of symbolic formulas: in a structure the objects or elements of the domain of discourse, or domain for short, are enlisted, together with functions and relations defined on the domain.

Definition 2.10 *A* structure *S is a tuple*

$$S = (D, \{\bar{f}_i^n : D^n \rightarrow D, n \geq 1\}, \{\bar{P}_i^m : D^m \rightarrow \{\mathit{true}, \mathit{false}\}, m \geq 0\})$$

having the following components:

(1) A non-empty set of elements D, called the domain *of S;*

(2) A set of functions *defined on D^n, $\{\bar{f}_i^n : D^n \rightarrow D, n \geq 1\}$;*

(3) A non-empty set of mappings, called predicates*, from D^m to the set of truth values $\{\mathit{true}, \mathit{false}\}$, $\{\bar{P}_i^m : D^m \rightarrow \{\mathit{true}, \mathit{false}\}, m \geq 0\}$.*

The basic idea underlying the definition of a structure is that we associate functions $\bar{f}_i^n$ to function symbols f_i and predicates $\bar{P}_i^m$ to predicate symbols P_i. Hence, we have to express how a given meaningless formula should be interpreted in a given structure: it is not possible to state anything about the truth value of a formula as long as it has not been prescribed which elements from the structure are to be associated with the elements in the formula.

EXAMPLE 2.12

Consider the formula $A(c)$. We associate the predicate having the intended meaning 'is a car' with the predicate symbol A. The formula should be *true* if the constant representing a Bugatti is associated with c; on the other hand, the same formula should be *false* if the constant representing a Volvo truck is associated with c. However, if we associate the predicate 'Truck' with A, the truth values of $A(c)$ for the two constants should be opposite to the ones mentioned before.

In the following definition, we introduce the notion of an assignment, which is a function that assigns elements from the domain of a structure to the variables in a formula.

Definition 2.11 *An* assignment *(*valuation*) v to a set of formulas F in a given structure S with domain D is a mapping from the set of variables in F to D.*

The interpretation of (terms and) formulas in a structure S under an assignment v now consists of the following steps. First, the constants in the formulas are assigned elements from D. Secondly, the variables are replaced by the particular elements from D that have been assigned to them by v. Then, the predicate and function symbols occurring in the formulas are assigned predicates and functions from S. Finally, the truth values of the formulas are determined.

Before the notion of an interpretation is defined more formally, a simple example in which no function symbols occur, is given. For the reader who is not interested in the formal aspects of logic, it suffices to merely study this example.

EXAMPLE 2.13

The open formula

$$F = A(x) \rightarrow O(x)$$

contains the unary predicate symbols A and O, and the free variable x. Consider the structure S consisting of the domain $D = \{\mathit{bugatti}, \mathit{volvo\text{-}truck}, \mathit{alfa\text{-}romeo}\}$ and the set of predicates comprising of the following elements:

- a unary predicate *Car*, with the intented meaning 'is a car', defined by $\mathit{Car}(\mathit{bugatti}) = \mathit{true}$, $\mathit{Car}(\mathit{alfa\text{-}romeo}) = \mathit{true}$ and $\mathit{Car}(\mathit{volvo\text{-}truck}) = \mathit{false}$, and
- the unary predicate *FourWheels* with the intended meaning 'has four wheels', defined by $\mathit{FourWheels}(\mathit{bugatti}) = \mathit{false}$, $\mathit{FourWheels}(\mathit{volvo\text{-}truck}) = \mathit{false}$ and $\mathit{FourWheels}(\mathit{alfa\text{-}romeo}) = \mathit{true}$.

Let us take for the predicate symbol A the predicate *Car*, and for the predicate symbol O the predicate *FourWheels*. It will be obvious that the atom $A(x)$ is *true* in S under any assignment v for which $\mathit{Car}(v(x)) = \mathit{true}$; so, for example for the assignment $v(x) = \mathit{alfa\text{-}romeo}$, we have that $A(x)$ is *true* in S under v. Furthermore, F is *true* in the structure S under the assignment v with $v(x) = \mathit{alfa\text{-}romeo}$, since $A(x)$ and $O(x)$ are both *true* in S under v. On the other hand, F is *false* in the structure S under the assignment v' with $v'(x) = \mathit{bugatti}$, because $\mathit{Car}(\mathit{bugatti}) = \mathit{true}$ and $\mathit{FourWheels}(\mathit{bugatti}) = \mathit{false}$ in S. Now, consider the closed formula

$$F' = \forall x (A(x) \rightarrow O(x))$$

and again the structure S. It should be obvious that F' is *false* in S.

Table 2.5: Laws of equivalence for quantifiers.

$\neg\exists x P(x) \equiv \forall x \neg P(x)$	(a)
$\neg\forall x P(x) \equiv \exists x \neg P(x)$	(b)
$\forall x (P(x) \wedge Q(x)) \equiv \forall x P(x) \wedge \forall x Q(x)$	(c)
$\exists x (P(x) \vee Q(x)) \equiv \exists x P(x) \vee \exists x Q(x)$	(d)
$\forall x P(x) \equiv \forall y P(y)$	(e)
$\exists x P(x) \equiv \exists y P(y)$	(f)

Definition 2.12 *An* interpretation *of terms in a structure $S = (D, \{\bar{f}_i^n\}, \{\bar{P}_i^m\})$ under an assignment v, denoted by I_v^S, is defined as follows:*

(1) $I_v^S(c_i) = d_i$, $d_i \in D$, where c_i is a constant.

(2) $I_v^S(x_i) = v(x_i)$, where x_i is a variable.

(3) $I_v^S(f_i^n(t_1, \ldots, t_n)) = \bar{f}_i^n(I_v^S(t_1), \ldots, I_v^S(t_n))$, where $\bar{f}_i^n$ is a function from S associated with the function symbol f_i^n.

The truth value of a formula in a structure S under an assignment v for a given interpretation I_v^S is obtained as follows:

(1) $I_v^S(P_i^m(t_1, \ldots, t_m)) = \bar{P}_i^m(I_v^S(t_1), \ldots, I_v^S(t_m))$, meaning that an atom $P_i^m(t_1, \ldots, t_m)$ is true in the structure S under the assignment v for the interpretation I_v^S if $\bar{P}_i^m(I_v^S(t_1), \ldots, I_v^S(t_m))$ is true, where $\bar{P}_i^m$ is the predicate from S associated with P_i^m.

(2) If the truth values of the formulas F and G have been determined, then the truth values of $\neg F$, $F \wedge G$, $F \vee G$, $F \rightarrow G$ and $F \leftrightarrow G$ are defined by the meanings of the connectives as listed in Table 2.1.

(3) $\exists x F$ is true under v if there exists an assignment v' differing from v at most with regard to x, such that F is true under v'.

(4) $\forall x F$ is true under v if for each v' differing from v at most with regard to x, F is true under v'.

The notions valid, invalid, satisfiable, unsatisfiable, logical consequence, equivalence and model have meanings in predicate logic similar to their meanings in propositional logic. In addition to the equivalences listed in Table 2.4, predicate logic also has some laws of equivalence for quantifiers, which are given in Table 2.5. Note that the properties $\forall x(P(x) \vee Q(x)) \equiv \forall x P(x) \vee \forall x Q(x)$ and $\exists x (P(x) \wedge Q(x)) \equiv \exists x P(x) \wedge \exists x Q(x)$ do *not* hold.

We conclude this subsection with another example.

EXAMPLE 2.14

We take the unary (meaningless) predicate symbols C, F, V, W and E, and the constants a and b from a given first-order language. Now, consider the following formulas:

(1) $\forall x (C(x) \rightarrow V(x))$

(2) $F(a)$

(3) $\forall x (F(x) \rightarrow C(x))$

(4) $\neg E(a)$

(5) $\forall x ((C(x) \wedge \neg E(x)) \rightarrow W(x))$

(6) $F(b)$

(7) $\neg W(b)$

(8) $E(b)$

Consider the structure S in the reality with a domain consisting of the elements and *bugatti*, which are assigned to the constants a and b, respectively. The set of predicates in S comprises the unary predicates *Car*, *Fast*, *Vehicle*, *FourWheels*, and *Exception*, which are taken for the predicate symbols C, F, V, W, and E, respectively. The structure S and the mentioned interpretation have been carefully chosen so as to satisfy the above-given closed formulas, for instance by giving the following intended meaning to the predicates:

Car	=	'is a car'
Fast	=	'is a fast car'
Vehicle	=	'is a vehicle'
FourWheels	=	'has four wheels'
Exception	=	'is an exception'

In the given structure S, the formula numbered 1 expresses the knowledge that every car is a vehicle. The fact that an alfa-romeo is a fast car, has been stated in formula 2. Formula 3 expresses that every fast car is a car, and formula 4 states that an alfa-romeo is not an exception to the rule that cars have four wheels, which has been formalized in logic by means of formula 5. A Bugatti is a fast car (formula 6), but contrary to an alfa-romeo it does not have 4 wheels (formula 7), and therefore is an exception to the last mentioned rule; the fact that Bugattis are exceptions is expressed by means of formula 8.

It should be noted that in another structure with another domain and other predicates, the formulas given above might have completely different meanings.

2.3 Clausal form of logic

Before turning our attention to reasoning in logic, we introduce in this section a syntactically restricted form of predicate logic, called the *clausal form of logic*, which will play an important role in the remainder of this chapter. This restricted form however, can be shown to be as expressive as full first-order predicate logic. The clausal form of logic is often employed, in particular in the fields of theorem proving and logic programming.

We start with the definition of some new notions.

Definition 2.13 *A* literal *is an atom, called a* positive literal*, or a negation of an atom, called a* negative literal.

Definition 2.14 *A* clause *is a closed formula of the form*

$$\forall x_1 \cdots \forall x_s (L_1 \vee \cdots \vee L_m)$$

where each L_i, $i = 1, \ldots, m$, $m \geq 0$, *is a literal, with* $L_i \neq L_j$ *for each* $i \neq j$, *and* $x_1, \ldots, x_s$, $s \geq 0$, *are variables occurring in* $L_1 \vee \cdots \vee L_m$. *If* $m = 0$, *the clause is said to be the* empty clause, *denoted by* $\Box$.

The empty clause $\Box$ is interpreted as a formula which is always *false*, in other words, $\Box$ is an unsatisfiable formula.

A clause

$$\forall x_1 \cdots \forall x_s (A_1 \vee \cdots \vee A_k \vee \neg B_1 \vee \cdots \vee \neg B_n)$$

where $A_1, \ldots, A_k, B_1, \ldots, B_n$ are atoms and $x_1, \ldots, x_s$ are variables, is equivalent to

$$\forall x_1 \cdots \forall x_s (B_1 \wedge \cdots \wedge B_n \rightarrow A_1 \vee \cdots \vee A_k)$$

as a consequence of the laws $\neg F \vee G \equiv F \rightarrow G$ and $\neg F \vee \neg G \equiv \neg(F \wedge G)$, and is often written as

$$A_1, \ldots, A_k \leftarrow B_1, \ldots, B_n$$

The last notation is the more conventional one in logic programming. The commas in $A_1, \ldots, A_k$ each stand for a disjunction, and the commas in $B_1, \ldots, B_n$ indicate a conjunction. $A_1, \ldots, A_k$ are called the *conclusions* of the clause, and $B_1, \ldots, B_n$ the *conditions*.

Each well-formed formula in first-order predicate logic can be translated into a set of clauses, which is viewed as the conjunction of its elements. As we will see, this translation process may slightly alter the meaning of the formulas. We shall illustrate the translation process by means of an example. Before proceeding, we define two normal forms which are required for the translation process.

Definition 2.15 *A formula* F *is in* prenex normal form *if* F *is of the form*

$$Q_1 x_1 \cdots Q_n x_n M$$

where each Q_i, $i = 1, \ldots, n$, $n \geq 1$, *equals one of the two quantifiers* $\forall$ *and* $\exists$, *and where* M *is a formula in which no quantifiers occur.* $Q_1 x_1 \ldots Q_n x_n$ *is called the* prefix *and* M *is called the* matrix *of the formula* F.

Definition 2.16 *A formula* F *in prenex normal form is in* conjunctive normal form *if the matrix of* F *is of the form*

$$F_1 \wedge \cdots \wedge F_n$$

where each F_i, $i = 1, \ldots, n$, $n \geq 1$, *is a disjunction of literals.*

EXAMPLE 2.15

Consider the following three formulas:

$$\forall x(P(x) \vee \exists y Q(x, y))$$
$$\forall x \exists y \forall z((P(x) \wedge Q(x, y)) \vee \neg R(z))$$
$$\forall x \exists y((\neg P(x) \vee Q(x, y)) \wedge (P(y) \vee \neg R(x)))$$

The first formula is not in prenex normal form because of the occurrence of an existential quantifier in the 'inside' of the formula. The other two formulas are both in prenex normal form; moreover, the last formula is also in conjunctive normal form.

The next example illustrates the translation of a well-formed formula into a set of clauses. The translation scheme presented in the example however is general and can be applied to any well-formed formula in first-order predicate logic.

EXAMPLE 2.16

Consider the following formula:

$$\forall x(\exists y P(x, y) \vee \neg \exists y(\neg Q(x, y) \rightarrow R(f(x, y))))$$

This formula is transformed in eight steps, first into prenex normal form, subsequently into conjunctive normal form, amongst others by applying the laws of equivalence listed in the tables 2.4 and 2.5, and finally into a set of clauses.

Step 1. Eliminate all implication symbols using the equivalences $F \rightarrow G \equiv \neg F \vee G$ and $\neg(\neg F) \equiv F$:

$$\forall x(\exists y P(x, y) \vee \neg \exists y(Q(x, y) \vee R(f(x, y))))$$

If a formula contains bi-implication symbols, these can be removed by applying the equivalence

$$F \leftrightarrow G \equiv (F \rightarrow G) \wedge (G \rightarrow F)$$

Step 2. Diminish the scope of the negation symbols in such a way that each negation symbol only governs a single atom. This can be accomplished by using the equivalences $\neg \forall x F(x) \equiv \exists x \neg F(x)$, $\neg \exists x F(x) \equiv \forall x \neg F(x)$, $\neg(\neg F) \equiv F$, together with the laws of De Morgan:

$$\forall x(\exists y P(x, y) \vee \forall y(\neg Q(x, y) \wedge \neg R(f(x, y))))$$

Step 3. Rename the variables in the formula using the equivalences $\forall x F(x) \equiv \forall y F(y)$ and $\exists x F(x) \equiv \exists y F(y)$, so that each quantifier has its own uniquely named variable:

$$\forall x(\exists y P(x, y) \vee \forall z(\neg Q(x, z) \wedge \neg R(f(x, z))))$$

Formulas only differing in the names of their bound variables are called *variants*.

Step 4. Eliminate all existential quantifiers. For any existentially quantified variable x not lying within the scope of a universal quantifier, all occurrences of x in the formula within the scope of the existential quantifier can be replaced by a new, that is, not previously used, constant symbol c. The particular existential quantifier may then be removed. For instance, the elimination of the existential quantifier in the formula $\exists x P(x)$ yields a formula $P(c)$. However, if an existentially quantified variable y lies within the scope of one or more universal quantifiers with the variables $x_1, \ldots, x_n$, $n \geq 1$, then the variable y may be functionally dependent upon $x_1, \ldots, x_n$. Let this dependency be represented explicitly by means of a new n-place function symbol g such that $g(x_1, \ldots, x_n) = y$. All occurrences of y within the scope of the existential quantifier then are replaced by the function term $g(x_1, \ldots, x_n)$, after which the existential quantifier may be removed. The constants and functions introduced in order to allow for the elimination of existential quantifiers are called *Skolem functions*.

The existentially quantified variable y in the example lies within the scope of the universal quantifier with the variable x, and is replaced by $g(x)$:

$$\forall x(P(x, g(x)) \vee \forall z(\neg Q(x, z) \wedge \neg R(f(x, z))))$$

Note that by replacing the existentially quantified variables by Skolem functions, we lose logical equivalence. Fortunately, it can be shown that a formula F is satisfiable if and only if the formula F', obtained from F by replacing existentially quantified variables in F by Skolem functions, is satisfiable as well. In general, the satisfiability of F and F' will not be based on the same model, since F' contains function symbols not occurring in F. In the following, it will become evident that this property is sufficient for our purposes.

Step 5. Transform the formula into prenex normal form, by placing all the universal quantifiers in front of the formula:

$$\forall x \forall z(P(x, g(x)) \vee (\neg Q(x, z) \wedge \neg R(f(x, z))))$$

Note that this is allowed because by step 3 each quantifier applies to a uniquely named variable; this means that the scope of all quantifiers is the entire formula.

Step 6. Bring the matrix in conjunctive normal form using the distributive laws:

$$\forall x \forall z((P(x, g(x)) \vee \neg Q(x, z)) \wedge (P(x, g(x)) \vee \neg R(f(x, z))))$$

Step 7. Select the matrix by disregarding the prefix:

$$(P(x, g(x)) \vee \neg Q(x, z)) \wedge (P(x, g(x)) \vee \neg R(f(x, z)))$$

All variables in the matrix are now implicitly considered to be universally quantified.

Step 8. Translate the matrix into a set of clauses, by replacing formulas of the form $F \wedge G$ by a set of clauses $\{F', G'\}$, where F' and G' indicate that F and G are now represented using the notational convention of logic programming:

$$\{P(x, g(x)) \leftarrow Q(x, z)), P(x, g(x)) \leftarrow R(f(x, z))i\}$$

or the notation used in automated theorem proving:

$$\{P(x, g(x)) \vee \neg Q(x, z),\ P(x, g(x)) \vee \neg R(f(x, z))\}$$

We conclude this subsection with the definition of a special type of clause, a so-called Horn clause, which is a clause containing at most one positive literal.

Definition 2.17 *A* Horn clause *is a clause having one of the following forms:*

(1) $A \leftarrow$

(2) $\leftarrow B_1, \ldots, B_n,\ n \geq 1$

(3) $A \leftarrow B_1, \ldots, B_n,\ n \geq 1$

A clause of the form 1 is called a unit clause*; a clause of form 2 is called a* goal clause.

Horn clauses are employed in the programming language PROLOG. We will return to this observation in Section 2.7.2.

2.4 Reasoning in logic: inference rules

In the Sections 2.1 and 2.2 we described how a meaning could be attached to a meaningless set of logical formulas. This is sometimes called the *declarative semantics* of logic. The declarative semantics offers a means for investigating for example whether or not a given formula is a logical consequence of a set of formulas. However, it is also possible to answer this question without examining the semantic contents of the formulas concerned, by applying so-called *inference rules*. Contrary to truth tables, inference rules are purely syntactic operations which only are capable of modifying the form of the elements of a given set of formulas. Inference rules either add, replace or remove formulas; most inference rules discussed in this book however add new formulas to a given set of formulas. In general, an inference rule is given as a schema in which a kind of meta-variables occur that may be substituted by arbitrary formulas. An example of such a schema is shown below:

$$\frac{A, A \rightarrow B}{B}$$

The formulas above the line are called the *premises*, and the formula below the line is called the *conclusion* of the inference rule. The above-given inference rule is known as *modus ponens*, and when applied, removes an implication from a formula. Another example of an inference rule, in this case for introducing a logical connective, is the following schema:

$$\frac{A, B}{A \wedge B}$$

Repeated applications of inference rules give rise to what is called a *derivation* or *deduction*. For instance, modus ponens can be applied to draw the conclusion S from the two formulas $P \wedge (Q \vee R)$ and $P \wedge (Q \vee R) \rightarrow S$. It is said that there exists a derivation of the formula S from the set of clauses $\{P \wedge (Q \vee R), P \wedge (Q \vee R) \rightarrow S\}$. This is denoted by:

$$\{P \wedge (Q \vee R), P \wedge (Q \vee R) \rightarrow S\} \vdash S$$

The symbol $\vdash$ is known as the *turnstile*.

EXAMPLE 2.17

Consider the set of formulas $\{P, Q, P \wedge Q \rightarrow S\}$. If the inference rule

$$\frac{A, B}{A \wedge B}$$

is applied to the formulas P and Q, the formula $P \wedge Q$ is derived; the subsequent application of modus ponens to $P \wedge Q$ and $P \wedge Q \rightarrow S$ yields S. So,

$$\{P, Q, P \wedge Q \rightarrow S\} \vdash S$$

Now that we have introduced inference rules, it is relevant to investigate how the declarative semantics of a particular class of formulas and its *procedural semantics*, described by means of inference rules, are interrelated: if these two notions are related to each other, we are in the desirable circumstance of being able to assign a meaning to formulas which have been derived using inference rules, simply by our knowledge of the declarative meaning of the original set of formulas. On the other hand, when starting with the known meaning of a set of formulas, it will then be possible to derive only formulas which can be related to that meaning. These two properties are known as the soundness and the completeness, respectively, of a collection of inference rules.

More formally, a collection of inference rules is said to be *sound* if and only if for each formula F derived by applying these inference rules on a given set of well-formed formulas S of a particular class (for example clauses), we have that F is a logical consequence of S. This property can be expressed more tersely as follows, using the notations introduced before:

if $S \vdash F$ then $S \vDash F$.

In other words, a collection of inference rules is sound if it preserves truth under the operations of a derivation. This property is of great importance, because only by applying sound inference rules it is possible to assign a meaning to the result of a derivation.

EXAMPLE 2.18

The previously discussed inference rule modus ponens is an example of a sound inference rule. From the given formulas F and $F \rightarrow G$, the formula G can be derived by applying modus ponens, that is, we have $\{F, F \rightarrow G\} \vdash G$. On the other hand, if $F \rightarrow G$ and F are both *true* under a particular interpretation w, then from the truth Table 2.1 we have that G is *true* under w as well. So, G is a logical consequence of the two given formulas: $\{F, F \rightarrow G\} \vDash G$.

The reverse property that by applying a particular collection of inference rules, each logical consequence F of a given set of formulas S can be derived, is called the *completeness* of the collection of inference rules:

if $S \vDash F$ then $S \vdash F$.

EXAMPLE 2.19

The collection of inference rules only consisting of modus ponens is not complete for all well-formed formulas in propositional logic. For example, it is not possible to derive the formula P from $\neg Q$ and $P \vee Q$, although P is a logical consequence of the two formulas. However, by combining modus ponens with other inference rules, it is possible to obtain a complete collection of inference rules.

The important question now arises if there exists a mechanical proof procedure, employing a particular sound and complete collection of inference rules, which is capable of determining whether or not a given formula F can be derived from a given set of formulas S. In 1936, A. Church and A.M. Turing showed, independently, that such a general proof procedure does not exist for first-order predicate logic. This property is called the *undecidability* of first-order predicate logic. All known proof procedures are only capable of deriving F from S (that is, are able to prove $S \vdash F$) if F is a logical consequence of S (that is, if $S \vDash F$); if F is not a logical consequence of S, then the proof procedure is not guaranteed to terminate.

However, for propositional logic there do exist proof procedures which always terminate and yield the right answer: for checking whether a given formula is a logical consequence of a certain set of formulas, we can simply apply truth tables. So, propositional logic is decidable.

The undecidability of first-order predicate logic has not refrained the research area of automated theorem proving from further progress. The major result of this research has been the development of an efficient and flexible inference rule, which is both sound and complete for proving inconsistency, called *resolution*. This is sometimes called *refutation completeness* (see below). However, the resolution rule is only suitable for manipulating formulas in clausal form. Hence, to use this inference rule on a set of arbitrary logical formulas in first-order predicate logic, it is required to translate each formula into the clausal form of logic by means of the procedure discussed in Section 2.3. This implies that resolution is *not* complete for *unrestricted* first-order predicate logic. The formulation of resolution as a suitable inference rule for automated theorem proving in the clausal form of logic has been mainly due to J.A. Robinson, who departed from earlier work by D. Prawitz. The final working-out of resolution in various algorithms, supplemented with specific implementation techniques, has been the work of a large number of researchers. Resolution is the subject of the remainder of this chapter.

2.5 Resolution and propositional logic

We begin this section with a brief, informal sketch of the principles of resolution. Consider a set of formulas S in clausal form. Suppose we are given a formula G, also in clausal form, for which we have to prove that it can be derived from S by applying resolution. Proving $S \vdash G$ is equivalent to proving that the set of clauses W, consisting of the clauses in S supplemented with the negation of the formula G, that is $W = S \cup \{\neg G\}$, is unsatisfiable. Resolution on W now proceeds as follows: first, it is checked whether or not W contains the empty clause $\Box$; if this is the case, then W is unsatisfiable, and G is a logical consequence of S. If the empty clause $\Box$ is not in W, then the resolution rule is applied on a suitable pair of clauses from W, yielding a new clause. Every clause derived this way is added to W, resulting in a new set of clauses on which the same resolution procedure is applied. The entire procedure is repeated until some generated set of clauses has been shown to contain the empty clause $\Box$, indicating unsatisfiability of W, or until all possible new clauses have been derived.

The basic principles of resolution are best illustrated by means of an example from propositional logic. In Section 2.6 we turn our attention to predicate logic.

EXAMPLE 2.20

Consider the following set of clauses:

$$\{C_1 = P \vee R, C_2 = \neg P \vee Q\}$$

These clauses contain *complementary* literals, that is, literals having opposite truth values, namely P and $\neg P$. Applying resolution, a new clause C_3 is derived being the disjunction of the original clauses C_1 and C_2 in which the complementary literals have been cancelled out. So, application of resolution yields the clause

$$C_3 = R \vee Q$$

which then is added to the original set of clauses.

The resolution principle is described more precisely in the following definition.

Definition 2.18 *Consider the two clauses C_1 and C_2 containing the literals L_1 and L_2 respectively, where L_1 and L_2 are complementary. The procedure of* resolution *proceeds as follows:*

(1) Delete L_1 from C_1 and L_2 from C_2, yielding the clauses C_1' and C_2';

(2) Form the disjunction C' of C_1' and C_2';

(3) Delete (possibly) redundant literals from C', thus obtaining the clause C.

The resulting clause C is called the resolvent *of C_1 and C_2. The clauses C_1 and C_2 are said to be the* parent clauses *of the resolvent.*

Resolution has the important property that when two given parent clauses are *true* under a given interpretation, their resolvent is *true* under the same interpretation as well: resolution is a sound inference rule. In the following theorem we prove that resolution is sound for the case of propositional logic.

THEOREM 1 *(*soundness of resolution*) Consider two clauses C_1 and C_2 containing complementary literals. Then, any resolvent C of C_1 and C_2 is a logical consequence of $\{C_1, C_2\}$.*

Proof: We are given that the two clauses C_1 and C_2 contain complementary literals. So, it is possible to write C_1 and C_2 as $C_1 = L \vee C_1'$ and $C_2 = \neg L \vee C_2'$ respectively for some literal L. By definition, a resolvent C is equal to $C_1' \vee C_2'$ from which possibly redundant literals have been removed. Now, suppose that C_1 and C_2 are both *true* under an interpretation w. We then have to prove that C is *true* under the same interpretation w as well. Clearly, either L or $\neg L$ is *false*. Suppose that L is *false* under w, then C_1 obviously contains more than one literal, since otherwise C_1 would be *false* under w. It follows that C_1' is *true* under w. Hence, $C_1' \vee C_2'$, and therefore also C, is *true* under w. Similarly, it can be shown that the resolvent is *true* under w if it is assumed that L is *true*. So, if C_1 and C_2 are *true* under w then C is *true* under w as well. Hence, C is a logical consequence of C_1 and C_2. $\diamond$

Resolution is also a complete inference rule. Proving the completeness of resolution is beyond the scope of this book; we therefore confine ourselves to merely stating the property.

EXAMPLE 2.21

In the definition of a clause in Section 2.3, it was mentioned that a clause was not allowed to contain duplicate literals. This condition appears to be a necessary requirement for the completeness of resolution. For example, consider the following set of formulas:

$$S = \{P \vee P, \neg P \vee \neg P\}$$

It will be evident that S is unsatisfiable, since $P \vee P \equiv P$ and $\neg P \vee \neg P \equiv \neg P$. However, if resolution is applied to S then in every step the tautology $P \vee \neg P$ is derived. It is not possible to derive the empty clause $\Box$.

Until now we have used the notion of a derivation only in an intuitive sense. Before giving some more examples, we define the notion of a derivation in a formal way.

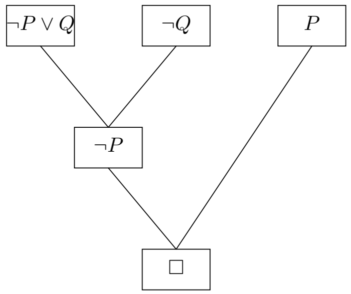


Figure 2.2: A refutation tree.

Definition 2.19 *Let S be a set of clauses and let C be a single clause. A* derivation *of C from S, denoted by $S \vdash_{\mathcal{B}} C$, is a finite sequence of clauses $C_1, C_2, \ldots, C_n$, $n \geq 1$, where each C_k either is a clause in S or a resolvent with parent clauses C_i and C_j, $i < k$, $j < k$, $i \neq j$, from the sequence, and $C = C_n$. If $C_n = \Box$, then the derivation is said to be a* refutation *of S, indicating that S is unsatisfiable.*

EXAMPLE 2.22

Consider the following set of clauses:

$$S = \{\neg P \vee Q, \neg Q, P\}$$

From $C_1 = \neg P \vee Q$ and $C_2 = \neg Q$ we obtain the resolvent $C_3 = \neg P$. From the clauses C_3 and $C_4 = P$ we derive $C_5 = \Box$. So, S is unsatisfiable. The sequence of clauses C_1, C_2, C_3, C_4, C_5 is a refutation of S. Note that it is not the only possible refutation of S. In general, a set S of clauses may have more than one refutation.

Notice that by the choice of the empty clause $\Box$ as a formula that is *false* under all interpretations, which is a *semantic* notion, the *proof-theoretical* notion of a refutation has obtained a suitable meaning. A derivation can be depicted in a graph, called a *derivation graph*. In the case of a refutation, the vertices in the derivation graph may be restricted to those clauses and resolvents which directly or indirectly contribute to the refutation. Such a derivation graph has the form of a tree and is usually called a *refutation tree*. The leaves of such a tree are clauses from the original set, and the root of the tree is the empty clause $\Box$. The refutation tree for the derivation discussed in the previous example is shown in Figure 2.2. Note that another refutation of S gives rise to another refutation tree.

2.6 Resolution and first-order predicate logic

An important feature of resolution in first-order predicate logic, taking place in the basic resolution method, is the manipulation of terms. This has not been dealt with in the previous section, where we only had atomic propositions, connectives and auxiliary symbols as building blocks for propositional formulas. In this section, we therefore first discuss the manipulation of terms, before we provide a detailed description of resolution in first-order predicate logic.

2.6.1 Substitution and unification

The substitution of terms for variables in formulas in order to make these formulas syntactically equal, plays a central role in a method known as *unification*. We first introduce the notion of substitution formally and then discuss its role in unification.

Definition 2.20 *A* substitution σ *is a finite set of the form*

$$\{t_1/x_1, \ldots, t_n/x_n\}$$

where each x_i is a variable and where each t_i is a term not equal to x_i, $i = 1, \ldots, n$, $n \geq 0$; the variables $x_1, \ldots, x_n$ differ from each other. An element t_i/x_i of a substitution σ is called a binding *for the variable x_i. If none of the terms t_i in a substitution contains a variable, we have a so-called* ground substitution. *The substitution defined by the empty set is called the* empty substitution, *and is denoted by ϵ.*

Definition 2.21 *An* expression *is a term, a literal, a conjunction of literals or a disjunction of literals; a* simple expression *is a term or an atom.*

A substitution σ can be applied to an expression E, yielding a new expression $E\sigma$ which is similar to E with the difference that the variables in E occurring in σ have been replaced by their associated terms.

Definition 2.22 *Let $\sigma = \{t_1/x_1, \ldots, t_n/x_n\}$, $n \geq 0$, be a substitution and E an expression. Then, $E\sigma$ is an expression obtained from E by simultaneously replacing all occurrences of the variables x_i by the terms t_i. $E\sigma$ is called an* instance *of E. If $E\sigma$ does not contain any variables, then $E\sigma$ is said to be a* ground instance *of E.*

EXAMPLE 2.23

Let $\sigma = \{a/x, w/z\}$ be a substitution and let $E = P(f(x, y), z)$ be an expression. Then, $E\sigma$ is obtained by replacing each variable x in E by the constant a and each variable z by the variable w. The result of the substitution is $E\sigma = P(f(a, y), w)$. Note that $E\sigma$ is not a ground instance.

The application of a substitution to a single expression can be extended to a set of expressions, as demonstrated in the following example.

EXAMPLE 2.24

Application of the substitution $\sigma = \{a/x, b/z\}$ to the set of expressions $\{P(x, f(x, z)), Q(x, w)\}$ yields the following set of instances:

$$\{P(x, f(x, z)), Q(x, w)\}\sigma = \{P(a, f(a, b)), Q(a, w)\}$$

The first element of the resulting set of instances is a ground instance; the second one is not ground, since it contains the variable w.

Definition 2.23 *Let $\theta = \{t_1/x_1, \ldots, t_m/x_m\}$ and $\sigma = \{s_1/y_1, \ldots, s_n/y_n\}$, $m \geq 1$, $n \geq 1$, be substitutions. The* composition *of these substitutions, denoted by $\theta\sigma$, is obtained by removing from the set*

$$\{t_1\sigma/x_1, \ldots, t_m\sigma/x_m, s_1/y_1, \ldots, s_n/y_n\}$$

all elements $t_i\sigma/x_i$ for which $x_i = t_i\sigma$, and furthermore, all elements s_j/y_j for which $y_j \in \{x_1, \ldots, x_m\}$

The composition of subtitutions is *associative*, i.e., for any expression E and substitutions ϕ, θ and σ we have that $E(\phi\sigma)\theta = E\phi(\sigma\theta)$; the operation is not commutative. Let $\theta = \{t_1/x_1, \ldots, t_m/x_m\}$ be a substitution, and let V be the set of variables occurring in $\{t_1, \ldots, t_m\}$, then θ is *idempotent*, i.e., $E(\theta\theta) = E\theta$, iff $V \cap \{x_1, \ldots, x_m\} = \varnothing$.

Note that the last definition gives us a means for replacing two substitutions by a single one, being the composition of these substitutions. However, it is not always necessary to actually compute the composition of two subsequent substitutions σ and θ before applying them to an expression E: it can easily be proven that $E(\sigma\theta) = (E\sigma)\theta$. The proof of this property is left to the reader as an exercise (see Exercise 2.11); here, we merely give an example.

EXAMPLE 2.25

Consider the expression $E = Q(x, f(y), g(z, x))$ and the two substitutions $\sigma = \{f(y)/x, z/y\}$ and $\theta = \{a/x, b/y, y/z\}$. We compute the composition $\sigma\theta$ of σ and θ: $\sigma\theta = \{f(b)/x, y/z\}$. Application of the compound substitution $\sigma\theta$ to E yields the instance $E(\sigma\theta) = Q(f(b), f(y), g(y, f(b)))$. We now compare this instance with $(E\sigma)\theta$. We first apply σ to E, resulting in $E\sigma = Q(f(y), f(z), g(z, f(y)))$. Subsequently, we apply θ to $E\sigma$ and obtain the instance $(E\sigma)\theta = Q(f(b), f(y), g(y, f(b)))$. So, for the given expression and substitutions, we have $E(\sigma\theta) = (E\sigma)\theta$.

In propositional logic, a resolvent of two parent clauses containing complementary literals, such as P and $\neg P$, was obtained by taking the disjunction of these clauses after cancelling out such a pair of complementary literals. It was easy to check for complementary literals in this case, since we only had to verify equality of the propositional atoms in the chosen literals and the presence of a negation in exactly one of them. Now, suppose that we want to compare the two literals $\neg P(x)$ and $P(a)$ occurring in two different clauses in first-order predicate logic. These two literals are 'almost' complementary. However, the first literal contains a variable as an argument of its predicate symbol, whereas the second one contains a constant. It is here where substitution comes in. Note that substitution can be applied to make expressions syntactically equal. Moreover, the substitution which is required to obtain syntactic equality of two given expressions also indicates the difference between the two. If we apply the substitution $\{a/x\}$ to the example above, we obtain syntactic equality of the two atoms $P(x)$ and $P(a)$. So, the two literals $\neg P(x)$ and $P(a)$ become complementary after substitution.

The *unification algorithm* is a general method for comparing expressions; the algorithm computes, if possible, the substitution that is needed to make the given expressions syntactically equal. Before we discuss the algorithm, we introduce some new notions.

Definition 2.24 *A substitution* σ *is called a* unifier *of a given set of expressions* $\{E_1, \ldots, E_m\}$ *if* $E_1\sigma = \cdots = E_m\sigma, m \geq 2$. *A set of expressions is called* unifiable *if it has a unifier.*

Definition 2.25 *A unifier* θ *of a unifiable set of expressions* $E = \{E_1, \ldots, E_m\}$, $m \geq 2$, *is said to be a* most general unifier *(mgu) if for each unifier* σ *of* E *there exists a substitution* λ *such that* $\sigma = \theta\lambda$.

A set of expressions may have more than one most general unifier; however, a most general unifier is unique but for a renaming of the variables.

EXAMPLE 2.26

Consider the set of expressions $\{R(x, f(a, g(y))), R(b, f(z, w))\}$. Some possible unifiers of this set are $\sigma_1 = \{b/x, a/z, g(c)/w, c/y\}$, $\sigma_2 = \{b/x, a/z, f(a)/y, g(f(a))/w\}$ and $\sigma_3 = \{b/x, a/z, g(y)/w\}$. The last unifier is also a most general unifier: by the composition of this unifier with the substitution $\{c/y\}$ we get σ_1; the second unifier is obtained by the composition of σ_3 with $\{f(a)/y\}$.

The unification algorithm, more precisely, is a method for constructing a most general unifier of a finite, non-empty set of expressions. The algorithm considered in this book operates in the following manner. First, the left-most subexpressions in which the given expressions differ is computed. Their difference is placed in a set, called the *disagreement set*. Based on this disagreement set a ('most general') substitution is computed, which is subsequently applied to the given expressions, yielding a partial or total equality. If no such substitution exists, the algorithm terminates with the message that the expressions are not unifiable. Otherwise, the procedure proceeds until each element within each of the expressions has been processed. It can be proven that the algorithm either terminates with a failure message or with a most general unifier of the finite, unifiable set of expressions.

EXAMPLE 2.27

Consider the following set of expressions:

$$S = \{Q(x, f(a), y), Q(x, z, c), Q(x, f(a), c)\}$$

The left-most subexpression in which the three expressions differ is in the second argument of the predicate symbol Q. So, the first disagreement set is $\{f(a), z\}$. By means of the substitution $\{f(a)/z\}$ the subexpressions in the second argument position are made equal. The next disagreement set is $\{y, c\}$. By means of the substitution $\{c/y\}$ these subexpressions are also equalized. The final result returned by the unification algorithm is the unifier $\{f(a)/z, c/y\}$ of S. It can easily be seen that this unifier is a most general one.

The following section shows an implementation of the unification algorithm. In the next section we discuss the role of unification in resolution in first-order predicate logic.

2.6.2 Resolution

Now that we have dealt with the subjects of substitution and unification, we are ready for a discussion of resolution in first-order predicate logic. We start with an informal introduction to the subject by means of an example.

EXAMPLE 2.28

Consider the following set of clauses:

$$\{C_1 = P(x) \vee Q(x), C_2 = \neg P(f(y)) \vee R(y)\}$$

As can be seen, the clauses C_1 and C_2 do not contain complementary literals. However, the atoms $P(x)$, occurring in C_1, and $P(f(y))$, occurring in the literal $\neg P(f(y))$ in the clause C_2, are unifiable. For example, if we apply the substitution $\sigma = \{f(a)/x, a/y\}$ to $\{C_1, C_2\}$, we obtain the following set of instances:

$$\{C_1\sigma = P(f(a)) \vee Q(f(a)), C_2\sigma = \neg P(f(a)) \vee R(a)\}$$

The resulting instances $C_1\sigma$ and $C_2\sigma$ *do* contain complementary literals, namely $P(f(a))$ and $\neg P(f(a))$ respectively. As a consequence, we are now able to find a resolvent of $C_1\sigma$ and $C_2\sigma$, being the clause

$$C_3' = Q(f(a)) \vee R(a)$$

The resolution principle in first-order predicate logic makes use of the unification algorithm for constructing a most general unifier of two suitable atoms; the subsequent application of the resulting substitution to the literals containing the atoms, renders them complementary. In the preceding example, the atoms $P(x)$ and $P(f(y))$ have a most general unifier $\theta = \{f(y)/x\}$. The resolvent obtained after applying θ to C_1 and C_2, is

$$C_3 = Q(f(y)) \vee R(y)$$

The clause C_3' from the previous example is an instance of C_3, the so-called *most general clause*: if we apply the substitution $\{a/y\}$ to C_3, we obtain the clause C_3'.

It should be noted that it is necessary to rename different variables having the same name in both parent clauses before applying resolution, since the version of the unification algorithm discussed in the previous section is not capable of distinguishing between equally named variables actually being the same variable, and equally named variables being different variables because of their occurrence in different clauses.

EXAMPLE 2.29

Consider the atoms $Q(x, y)$ and $Q(x, f(y))$ occurring in two different clauses. In this form our unification algorithm reports failure in unifying these atoms (due to the occur check). We rename the variables x and y in $Q(x, f(y))$ to u and v respectively, thus obtaining the atom $Q(u, f(v))$. Now, if we apply the unification algorithm again to compute a most general unifier of $\{Q(u, f(v)), Q(x, y)\}$, it will come up with the (correct) substitution $\sigma = \{u/x, f(v)/y\}$.

We already mentioned in Section 2.3 that the meaning of a formula is left unchanged by renaming variables. We furthermore recall that formulas only differing in the names of their (bound) variables are called variants.

From the examples presented so far, it should be clear by now that resolution in first-order predicate logic is quite similar to resolution in propositional logic: literals are cancelled out from clauses, thus generating new clauses. From now on, cancelling out a literal L from a clause C will be denoted by $C \backslash L$.

Definition 2.26 *Consider the parent clauses C_1 and C_2, respectively containing the literals L_1 and L_2. If L_1 and $\neg L_2$ have a most general unifier σ, then the clause $(C_1\sigma \backslash L_1\sigma) \vee (C_2\sigma \backslash L_2\sigma)$ is called a* binary resolvent *of C_1 and C_2. Resolution in which each resolvent is a binary resolvent, is known as* binary resolution.

A pair of clauses may have more than one resolvent, since they may contain more than one pair of complementary literals. Moreover, not every resolvent is necessarily a binary resolvent: there are more general ways for obtaining a resolvent. Before giving a more general definition of a resolvent, we introduce the notion of a factor.

Definition 2.27 *If two or more literals in a clause C have a most general unifier σ, then the clause $C\sigma$ is said to be a* factor *of C.*

EXAMPLE 2.30

Consider the following clause:

$$C = P(g(x), h(y)) \vee Q(z) \vee P(w, h(a))$$

The literals $P(g(x), h(y))$ and $P(w, h(a))$ in C have a most general unifier $\sigma = \{g(x)/w, a/y\}$. So,

$$C\sigma = P(g(x), h(a)) \vee Q(z) \vee P(g(x), h(a)) = P(g(x), h(a)) \vee Q(z)$$

is a factor of C. Note that one duplicate literal $P(g(x), h(a))$ has been removed from $C\sigma$.

The generalized form of resolution makes it possible to cancel out more than one literal from one or both of the parent clauses by first computing a factor of one or both of these clauses.

EXAMPLE 2.31

Consider the following set of clauses:

$$\{C_1 = P(x) \vee P(f(y)) \vee R(y), C_2 = \neg P(f(a)) \vee \neg R(g(z))\}$$

In the clause C_1 the two literals $P(x)$ and $P(f(y))$ have a most general unifier $\sigma = \{f(y)/x\}$. If we apply this substitution σ to the clause C_1, then one of these literals can be removed:

$$\begin{aligned}(P(x) \vee P(f(y)) \vee R(y))\sigma &= P(f(y)) \vee P(f(y)) \vee R(y) \\ &= P(f(y)) \vee R(y)\end{aligned}$$

The result is a factor of C_1. The literal $P(f(y))$ in $C_1\sigma$ can now be unified with the atom $P(f(a))$ in the literal $\neg P(f(a))$ occurring in C_2, using the substitution $\{a/y\}$. We obtain the resolvent

$$C_3 = R(a) \vee \neg R(g(z))$$

Note that a total of three literals has been removed from C_1 and C_2. The reader can easily verify that there are several other resolvents from the same parent clauses:

- By taking $L_1 = P(x)$ and $L_2 = \neg P(f(a))$ we get the resolvent $P(f(y)) \vee R(y) \vee \neg R(g(z))$;
- Taking $L_1 = P(f(y))$ and $L_2 = \neg P(f(a))$ results in the resolvent $P(x) \vee R(a) \vee \neg R(g(z))$;
- By taking $L_1 = R(y)$ and $L_2 = \neg R(g(z))$ we obtain $P(x) \vee P(f(g(z))) \vee \neg P(f(a))$.

We now give the generalized definition of a resolvent in which the notion of a factor is incorporated.

Definition 2.28 *A* resolvent *of the parent clauses C_1 and C_2 is one of the following binary resolvents:*

(1) A binary resolvent of C_1 and C_2;

(2) A binary resolvent of C_1 and a factor of C_2;

(3) A binary resolvent of a factor of C_1 and C_2;

(4) A binary resolvent of a factor of C_1 and a factor of C_2.

The most frequent application of resolution is refutation: the derivation of the empty clause $\Box$ from a given set of clauses. The following procedure gives the general outline of this resolution algorithm.

```
procedure Resolution(S)
  clauses ← S;
  while □ ∉ clauses do
    {c_i, c_j} ← SelectResolvable(clauses);
    resolvent ← Resolve(c_i, c_j);
    clauses ← clauses ∪ {resolvent}
  od
end
```

This algorithm is non-deterministic. The selection of parent clauses c_i and c_j can be done in many ways; how it is to be done has not been specified in the algorithm. Several different strategies have been described in the literature, each of them prescribing an unambiguous way of choosing parent clauses from the clause set. Such strategies are called the *control strategies* of resolution or *resolution strategies*. Several of these resolution strategies offer particularly efficient algorithms for making computer-based theorem proving feasible. Some well-known strategies are: *semantic resolution*, which was developed by J.R. Slagle in 1967, *hyperresolution* developed by J.A. Robinson in 1965, and various forms of *linear resolution*, such as *SLD resolution*, in the development of which R.A. Kowalski played an eminent role. At present, SLD resolution in particular is a strategy of major interest, because of its relation to the programming language PROLOG.

2.7 Resolution strategies

Most of the basic principles of resolution have been discussed in the previous section. However, one particular matter, namely the efficiency of the resolution algorithm, has not explicitly been dealt with as yet. It is needless to say that the subject of efficiency is an important one for automated reasoning.

Unfortunately, the general refutation procedure introduced in Section 2.6.3 is quite inefficient, since in many cases it will generate a large number of redundant clauses, that is, clauses not contributing to the derivation of the empty clause.

EXAMPLE 2.32

Consider the following set of clauses:

$$S = \{P, \neg P \vee Q, \neg P \vee \neg Q \vee R, \neg R\}$$

To simplify referring to them, the clauses are numbered as follows:

(1) P
(2) $\neg P \vee Q$
(3) $\neg P \vee \neg Q \vee R$
(4) $\neg R$

If we apply the resolution principle by systematically generating all resolvents, without utilizing a more specific strategy in choosing parent clauses, the following resolvents are successively added to S:

(5) Q (using 1 and 2)
(6) $\neg Q \vee R$ (using 1 and 3)
(7) $\neg P \vee R$ (using 2 and 3)
(8) $\neg P \vee \neg Q$ (using 3 and 4)
(9) R (using 1 and 7)
(10) $\neg Q$ (using 1 and 8)
(11) $\neg P \vee R$ (using 2 and 6)
(12) $\neg P$ (using 2 and 8)
(13) $\neg P \vee R$ (using 3 and 5)
(14) $\neg Q$ (using 4 and 6)
(15) $\neg P$ (using 4 and 7)

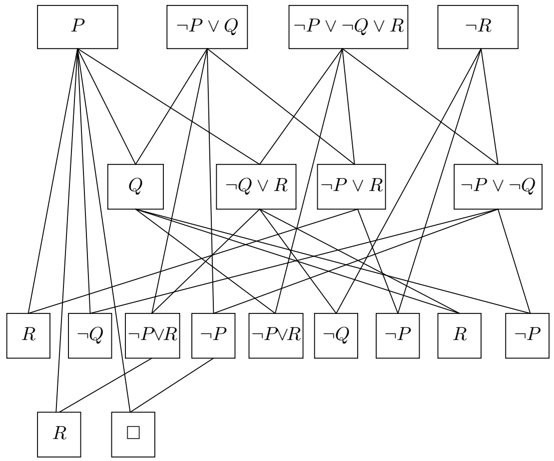


Figure 2.3: Refutation of $\{P, \neg P \vee Q, \neg P \vee \neg Q \vee R, \neg R\}$.

(16) R (using 5 and 6)
(17) $\neg P$ (using 5 and 8)
(18) R (using 1 and 11)
(19) $\Box$ (using 1 and 12)

This derivation of the empty clause $\Box$ from S has been depicted in Figure 2.3 by means of a derivation graph. As can be seen, by systematically generating all resolvents in a straightforward manner, fifteen of them were obtained, while, for instance, taking the two resolvents

(5′) $\neg P \vee R$ (using 2 and 3)
(6′) R (using 1 and 5′)

would lead directly to the derivation of the empty clause:

(7′) $\Box$ (using 4 and 6′)

In the latter refutation, significantly less resolvents were generated.

The main goal of applying a resolution strategy is to restrict the number of redundant clauses generated in the process of resolution. This improvement in efficiency is achieved by incorporating particular algorithmic refinements in the resolution principle. Some important resolution strategies will be discussed in the following two sections.

2.7.1 Semantic resolution

Semantic resolution is the name of a class of resolution strategies all having in common that the process of resolution is controlled by the declarative semantics of the clauses to be processed. We will briefly introduce the general idea and present some special forms of semantic resolution informally.

Basic ideas

Consider an unsatisfiable set of clauses S. It is possible to divide the set of clauses S into two separate subsets on the basis of a particular interpretation I: the subset S_1 contains the clauses from S which are *false* in I, and the subset S_2 contains the clauses which are *true* in I. Since S is unsatisfiable, no interpretation can ever make all clauses either *true* or *false*. So, the clause set S is split into two non-empty subsets. This semantic splitting can be used as the basis for a control strategy in which one of the parent clauses is chosen from S_1, and the other one from S_2. The generated resolvent is either added to S_1 or to S_2, dependent upon the interpretation I. In the next example, the particulars of this form of resolution are illustrated.

EXAMPLE 2.33

Consider once more the following unsatisfiable set of clauses: $S = \{P, \neg P \vee Q, \neg P \vee \neg Q \vee R, \neg R\}$. Furthermore, consider the interpretation I, defined by

$$\begin{aligned} I(P) &= \textit{false}, \\ I(Q) &= \textit{false}, \text{ and} \\ I(R) &= \textit{false}. \end{aligned}$$

Using this interpretation, we divide the set S into the following two subsets S_1 and S_2:

$$\begin{aligned} S_1 &= \{P\} \\ S_2 &= \{\neg P \vee Q, \neg P \vee \neg Q \vee R, \neg R\} \end{aligned}$$

The reader can verify that using the control strategy mentioned above, only the resolvents Q, $\neg Q \vee R$, $\neg P \vee R$, R and $\Box$ will successively be generated.

A further refinement of the described strategy can be obtained by assigning a particular *order* to the literals in the clauses. For example, in propositional logic an ordering is imposed on the propositional symbols occurring in the set of clauses. Resolution now is restricted not only by requiring that the two parent clauses are selected from the different subsets S_1 and S_2 of S (obtained from an interpretation I), but in addition, by demanding that the literal from the clause selected from S_1 to be resolved upon, is in that clause the highest one according to the ordering imposed.

Set-of-support strategy

A popular form of semantic resolution is the *set-of-support strategy*. As we mentioned in the foregoing, resolution is generally applied to prove that a specific clause G is the logical

consequence of a satisfiable set of clauses S. Usually such a proof is by refutation, that is, it has the form of a derivation of the empty clause $\Box$ from $W = S \cup \{\neg G\}$. The information that the set of clauses S is satisfiable, is exploited in the set-of-support strategy to decrease the number of resolvents generated. Obviously, it is not sensible to select both parent clauses from S: since S is satisfiable, the resulting resolvent could never be the empty clause $\Box$. In the set-of-support strategy a given set of clauses W is divided into two disjoint sets: the set S being the original satisfiable set of clauses and the set T initially only containing the clauses to be proven. The set T is called the *set of support*. Now, in each resolution step at least one of the parent clauses has to be a member of the set of support. Each resulting resolvent is added to T. It is said that these clauses 'support' the clauses that were to be proven, hence the name 'set of support'. The set-of-support strategy is a powerful control strategy, which prevents the generation of many resolvents not contributing to the actual proof. The strategy is both sound and complete.

EXAMPLE 2.34

Consider the following set of clauses:

$$W = \{P, \neg P \vee Q, \neg P \vee \neg Q \vee R, \neg R\}$$

It can easily be seen that the following subset $S \subset W$ is satisfiable:

$$S = \{P, \neg P \vee Q, \neg P \vee \neg Q \vee R\}$$

(For example, choose an interpretation I such that $I(P) = I(Q) = I(R) = \textit{true}$.) The remaining clause from W constitutes the set of support $T = \{\neg R\}$; so, $S \cup T = W$. For ease of exposition, we again number the clauses in S and T:

(1) P
(2) $\neg P \vee \neg Q \vee R$
(3) $\neg P \vee Q$
(4) $\neg R$

Resolution using the set-of-support strategy successively generates the following resolvents:

(5) $\neg P \vee \neg Q$ (using 2 and 4)
(6) $\neg Q$ (using 1 and 5)
(7) $\neg P$ (using 3 and 5)
(8) $\Box$ (using 1 and 7).

The set-of-support strategy can be described aptly in algorithmic form as follows:

procedure SOS-strategy(W, S)

consistent $\leftarrow$ *true*;
while *consistent* **and** $S \neq \varnothing$ **do**

$C \leftarrow$ SelectClause(S);
$S \leftarrow S \backslash \{C\}$;
resolvents $\leftarrow$ Resolve(C, W);
$W \leftarrow W \cup \{C\}$;
Process(*resolvents*,W);
consistent $\leftarrow \Box \notin$ *resolvents*;
if *consistent* **then**
$S \leftarrow S \cup$ *resolvents*
fi
od;
if not *consistent* **then**
print("Inconsistency found.")
fi
end

The *SelectClause* select a clause from the set-of-support. This clause is called the *given clause.* One possibility is to let S behave like a queue; in that case clauses are selected in the order in which they appear in S.

Note that this strategy can be considered to be a form of top-down inference: the set of support when explored using this strategy can be looked upon as a set of goals.

Hyperresolution

It is often possible to shorten the derivation obtained by binary resolution, by combining a number of derivation steps into one. An advantage of this is that the number of newly generated clauses is decreased and this has normally a favourable effect on the length of derivations. Hyperresolution is a popular inference rule that exactly does this.

Definition 2.29 *Let I be an interpretation for the set of clause S. A set of clauses $\{S_1, \ldots, S_n, N\} \subseteq S$ is called a* semantic conflict *if the following holds:*

(1) $I(S_i) = false$, $1 \leq i \leq n$;

(2) if $R_1 = N$, then $\{S_i, R_i\} \vdash_{\mathcal{B}} R_{i+1}$, $i = 1, \ldots, n$;

(3) $\mathcal{I}(R_{n+1}) = false$.

The clauses S_i are called satellites *(or electrons), the clause N is called the* nucleus*, and the clause R_{n+1} is known as the* hyperresolvent*. The inference rule:*

$$\frac{S_1, \ldots, S_n, N}{R_{n+1}}$$

is called the hyperresolution rule*. A derivation of C from S by means of hyperresolution is denoted by $S \vdash_{\mathcal{H}} C$.*

Different inference rules are obtained for particular choices of the interpretation I. One choice leads to the rule known as positive hyperresolution, whereas the complementary choice leads to negative hyperresolution.

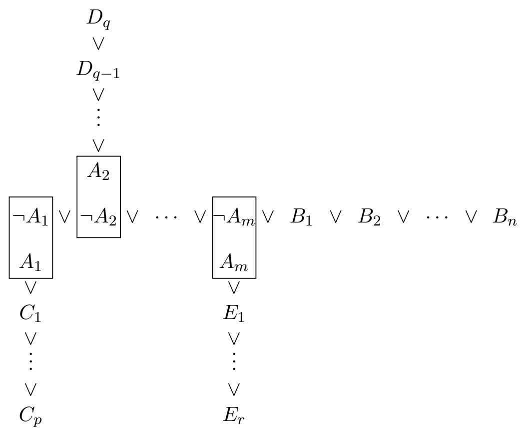


Figure 2.4: Nucleus and satellites.

Definition 2.30 *Let I be an interpretation, such that for each positive literal L it holds that $I(L) = false$, then the application of hyperresolution is called* positive hyperresolution*. The satellites and the hyperresolvent are positive clauses in that case. If it holds that $I(L) = true$, then the application of hyperresolution is called* negative hyperresolution*. The satellites and the hyperresolvent are then negative clauses. A derivation by means of positive hyperresolution is denoted by $S \vdash_{\mathcal{P}} C$; similarly, for negative hyperresolution a derivation is denoted by $S \vdash_{\mathcal{N}} C$.*

Positieve hyperresolution is used as a form of bottom-up problem solving, i.e., a form of data-driven inference. The satellites can then be interpreted as data, and each newly generated hyperresolvent (a positive clause) as newly generated data. In contrast, negative hyperresolution can be seen as a form of top-down problem solving, i.e. hypothesis-driven inference.

The underlying idea of positive hyperresolution is that negative literals are deleted by the positive satellites from the nucleus, yielding a positive clause. This is illustrated schematically in Figure 2.4. Similarly, in negative hyperresolution positive literals are deleted.

EXAMPLE 2.35

Consider the following set of clauses:

$$V = \{\neg P \vee \neg Q \vee R \vee S,\ P \vee S,\ Q \vee R,\ \neg R,\ \neg S\}$$

Application of positive hyperresolution yields the following clause in one step:

$$\{\neg P \vee \neg Q \vee R \vee S,\ P \vee S,\ Q \vee R\} \vdash_{\mathcal{P}} R \vee S$$

The nucleus of positive hyperresolution is the mixed clause: $\neg P \vee \neg Q \vee R \vee S$; the satellites are $P \vee S$ and $Q \vee R$. This is the only possibility in this case. Using binary resolution would have required two derivation steps.

We obtain the following derivation:

1. $\neg P \vee \neg Q \vee R \vee S$
2. $P \vee S$
3. $Q \vee R$
4. $\neg R$
5. $\neg S$
6. $R \vee S$ (via clause 1, 2 en 3)
7. S (via clause 4 en 6)
8. $\Box$ (via clause 5 en 7).

Negative hyperresolution yields the following:

$$\{\neg P \vee \neg Q \vee R \vee S,\ \neg R,\ \neg S\} \vdash_{\mathcal{N}} \neg P \vee \neg Q$$

The clause $\neg P \vee \neg Q \vee R \vee S$ is again the nucleus, but this time there are two negative (unit) clause $\neg R$ and $\neg S$ that act as satellites. The complete derivation of the empty clause using negative hyperresolution is as follows:

1. $\neg P \vee \neg Q \vee R \vee S$
2. $P \vee S$
3. $Q \vee R$
4. $\neg R$
5. $\neg S$
6. $\neg P \vee \neg Q$ (via clause 1, 4 en 5)
7. $\neg Q$ (via clause 2, 5 en 6)
8. $\Box$ (via clause 3,4 en 7).

In the next example we study the use of hyperresolution using predicate logic.

EXAMPLE 2.36

Consider the following clauses:

$$\begin{aligned} S = \{&Q(a) \vee R(x),\\ &\neg Q(x) \vee R(x),\\ &\neg R(a) \vee \neg S(a),\\ &S(x)\} \end{aligned}$$

First, the clause are numbered:

1. $Q(a) \vee R(x)$
2. $\neg Q(x) \vee R(x)$
3. $\neg R(a) \vee \neg S(a)$

4. $S(x)$.

Application of positive hyperresolution yields:

5. $R(x) \vee R(a)$ (via clause 1 and 2)
6. $\Box$ (via clause 5, 3 and 4).

The following derivation is obtained by negative hyperresolution:

5. $\neg R(a)$ (via clause 3 and 4)
6. $\neg Q(a)$ (via clause 5 and 2)
7. $\Box$ (via clause 5, 6 and 1).

Note that in some cases, a hyperresolution step is exactly the same as the application of binary resolution.

Hyperresolution is not only sound but also refutation complete. However, when combined with the set-of-support strategy it is possible that refutation completeness is lost, as illustrated by means of the following example.

EXAMPLE 2.37

Consider the set of clauses $V = S \cup W$, where

$$\begin{aligned} S &= \{\neg P \vee \neg R\}\\ W &= \{P,\ R \vee \neg Q,\ Q\} \end{aligned}$$

where S is the set-of-support. Note that W is satisfiable. Application of positive hyperresolution with the set-of-support strategy does yield no resolvents, as it is not possible to generate a positive clause from the clause in S as a nucleus and the clauses in W as satellites. Note, however, that S is unsatisfiable. Negative hyperresolution is able to find the inconsistency in this case.

2.7.2 SLD resolution: a special form of linear resolution

Linear resolution has been named after the structure of the derivation graph created by this class of strategies: in every resolution step the last generated resolvent is taken as a parent clause. The other parent clause is either a clause from the original set of clauses or a resolvent that has been generated before. A special form of linear resolution is *input resolution*. In this strategy, each resolution step, with the exception of the first one, is carried out on the last generated resolvent and a clause from the original set of clauses. The former clauses are called *goal clauses*; the latter clauses are called *input clauses*, thus explaining the name of the strategy. Input resolution is a complete strategy for Horn clauses; for the clausal form of logic in general however, input resolution is not complete.

A variant of input resolution which currently attracts a great deal of attention is *SLD resolution* for Horn clauses. In this resolution strategy, input resolution is extended with a *selection rule* which determines at every step which literal from the goal clause is selected for resolution. The remainder of this section discusses SLD resolution.

An SLD derivation is defined as follows:

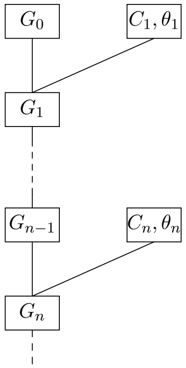


Figure 2.5: Derivation tree of SLD resolution.

Definition 2.31 *Let $\{C_i\}$ be a set of Horn clauses with*

$$C_i = B \leftarrow B_1, \ldots, B_p$$

where $p \geq 0$, and let G_0 be a goal clause of the form

$$G_0 = \ \leftarrow A_1, \ldots, A_q$$

where $q \geq 0$. An SLD derivation *is a finite or infinite sequence $G_0, G_1, \ldots$ of goal clauses, a sequence $C_1, C_2, \ldots$ of variants of input clauses and a sequence $\theta_1, \theta_2, \ldots$ of most general unifiers, such that each G_{i+1} is derived from $G_i = \ \leftarrow A_1, \ldots, A_k$ and C_{i+1} using θ_{i+1} if the following conditions hold:*

(1) A_j is the atom in the goal clause G_i chosen by the selection rule to be resolved upon, and

(2) C_{i+1} is an input clause of the form

$$C_{i+1} = B \leftarrow B_1, \ldots, B_p$$

(in which variables have been renamed, if necessary), such that $A_j\theta_{i+1} = B\theta_{i+1}$, where θ_{i+1} is a most general unifier of A_j and B.

(3) $G_i + 1$ is the clause

$$G_{i+1} = \ \leftarrow (A_1, \ldots, A_{j-1}, B_1, \ldots, B_p, A_{j+1}, \ldots, A_k)\theta_{i+1}$$

If for some $n \geq 0$, $G_n = \square$, then the derivation is called an SLD refutation *and the number n is called the* length of the refutation.

Note that a new goal clause G_{i+1} is the resolvent of the last computed resolvent G_i and (a variant of) an input clause C_{i+1}. Figure 2.5 shows the general form of a derivation tree by SLD resolution. In this figure the sequence of successive goal clauses (resolvents) $G_0, G_1, \ldots$ has been indicated.

EXAMPLE 2.38

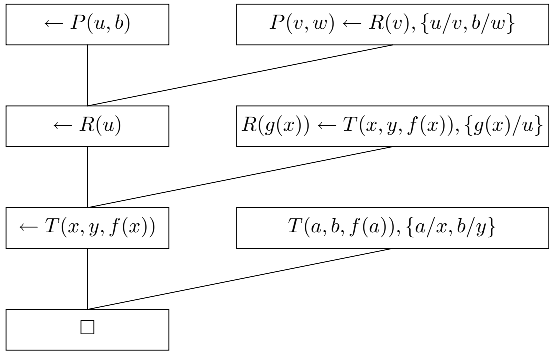


Figure 2.6: An SLD refutation.

Consider the following set of Horn clauses:

$$\{R(g(x)) \leftarrow T(x, y, f(x)), T(a, b, f(a)), P(v, w) \leftarrow R(v)\}$$

Furthermore, let the following goal clause be given:

$$\leftarrow P(u, b)$$

The clause set obtained by adding the goal clause to the original set of clauses is unsatisfiable. This can be proven using SLD resolution. Figure 2.6 depicts this proof by SLD refutation as a derivation tree.

SLD resolution is both sound and complete for Horn clauses. It furthermore is similar to the set-of-support strategy in the sense that it is also a resolution strategy controlled by a set of goals. So, SLD resolution is a form of top-down inference as well. In general it is advantageous to restrict applying the resolution principle to clauses satisfying the Horn clause format: various resolution algorithms for propositional Horn clause logic are known to have a worst-case time complexity almost linear in the number of literals. When applying some resolution strategy suitable for the clausal form of logic in general, we always have to face the danger of a combinatorial explosion. Moreover, for systems based on SLD resolution many efficient implementation techniques have been developed by now, one of which will be discussed in the next section. But there definitely are problems for which a resolution strategy applying some form of bottom-up inference turns out to be more efficient than SLD resolution.

Before introducing the notion of a search space for SLD resolution, we give another example.

EXAMPLE 2.39

Consider the following set of Horn clauses:

$$\begin{aligned} C_1 &= P(x) \leftarrow P(f(x)) \\ C_2 &= P(f(f(a))) \leftarrow \end{aligned}$$

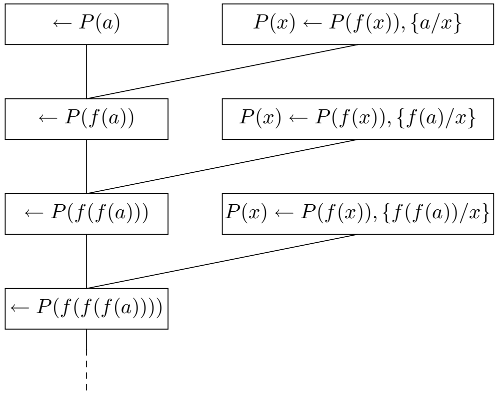


Figure 2.7: Infinite derivation tree by SLD resolution.

If these clauses are 'tried' in the order in which they are specified, then for the goal clause $\leftarrow P(a)$ no refutation is found in a finite number of steps, although the resulting set of clauses obviously is unsatisfiable. The corresponding derivation tree is shown in Figure 2.7. However, if the clauses C_1 and C_2 are processed in the reverse order C_2, C_1, then a refutation will be found in finite time: the resulting refutation tree is shown in Figure 2.8.

Now let the search space for SLD resolution for a given goal on a set of clauses be a graph in which every possible SLD derivation is shown. Such a search space is often called an *SLD tree*. The branches of the tree terminating in the empty clause $\Box$ are called *success branches*. Branches corresponding to infinite derivations are called *infinite branches*, and the branches representing derivations which have not been successful and cannot be pursued any further are called *failure branches*. The *level* of a vertex in an SLD tree is obtained by assigning the number 0 to the root of the tree; the level of each other vertex of the tree is obtained by incrementing the level of its parent vertex by 1.

EXAMPLE 2.40

Figure 2.9 shows the SLD tree corresponding to SLD resolution on the set of clauses from the previous example. The right branch of the tree is a success branch and corresponds to the refutation depicted in Figure 2.8; the left branch is an example of a failure branch.

It can easily be seen that a specific, fixed order in choosing parent clauses for resolution such as in the previous example, corresponds to a depth-first search in the search space. Note that such a depth-first search defines an incomplete resolution procedure, whereas a breadth-first search strategy defines a complete one. Although SLD resolution is both sound and complete for Horn clauses, in practical realizations for reasons of efficiency, variants of the algorithm are used that are neither sound nor complete. First of all, in many implementations the

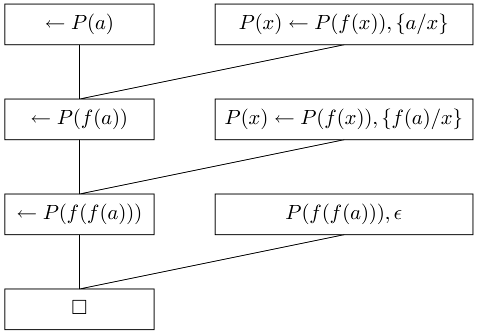


Figure 2.8: Refutation by SLD resolution.

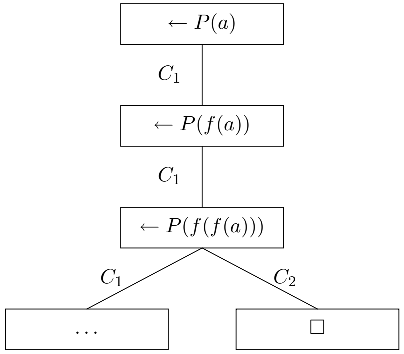


Figure 2.9: An SLD tree.

'expensive' occur check has been left out from the unification algorithm, thus destroying the soundness; the lack of the occur check might lead to circular variable bindings and yield 'resolvents' that are no logical consequences of the set of clauses. Furthermore, often the original clauses are 'tried' in some specific order, such as for example the order in which the clauses have been specified; the next input clause is only examined after the previous one has been fully explored. As a consequence, the algorithm might not be able to find a proof of a given theorem: due to an inappropriate choice of the order in which the clauses are processed, an infinite derivation tree can be created. This way, completeness of SLD resolution will be lost.

We have mentioned before that SLD resolution is of major interest because of its relation with the programming language PROLOG. In PROLOG, the control strategy employed is roughly an implementation of SLD resolution; the variant used however, is neither sound nor complete. In most (standard) PROLOG systems, the selection rule picks the leftmost atom from a goal for resolution. A depth-first strategy for searching the SLD tree is used: most PROLOG systems 'try' the clauses in the order in which they have been specified. Furthermore, in many PROLOG systems, for efficiency reasons, the occur check has been left out from the implementation.

The Horn clause subset of logic is not as expressive as the full clausal form of logic is. As is shown in the following example, this might lead to problems when translating the logical formulas into the Horn clause subset. We next show what solution PROLOG offers to this problem.

EXAMPLE 2.41

In Section 2.2 we defined the following predicates with their associated intended meaning:

Car	=	'is a car'
Fast	=	'is a fast car'
Vehicle	=	'is a vehicle'
FourWheels	=	'has four wheels'
Exception	=	'is an exception'

The formula $\forall x(Car(x) \rightarrow Vehicle(x))$ represents the knowledge that every car is a vehicle. This formula is logically equivalent to $\forall x(\neg Car(x) \vee Vehicle(x))$ and results in the following PROLOG clause:

```
vehicle(X) :- car(X).
```

The knowledge that a Bugatti is a fast car, is represented in PROLOG by a single fact:

```
fast(bugatti).
```

The implication

$$\forall x((Car(x) \wedge \neg Exception(x)) \rightarrow FourWheels(x))$$

stating that almost every car, except for instance a Bugatti, has four wheels. This formula is equivalent to

$$\forall x(\neg(Car(x) \wedge \neg Exception(x)) \vee FourWheels(x))$$

and to the formula

$$\forall x(\neg Car(x) \vee Exception(x) \vee FourWheels(x))$$

in disjunctive normal form. Unfortunately, it is not possible to translate this formula directly into PROLOG representation, since the clause contains two positive literals instead of at most one.

However, it is possible to represent the knowledge expressed by the clause in PROLOG, by means of the rather special programming trick offered by the meaning of the standard predicate `not`, which will be discussed below. The PROLOG clause we arrive at is the following:

```
fourwheels(X) :-
    car(X),
    not(exception(X)).
```

Note that in the analogous example in Section 2.2 it was necessary to specify that an alfa-romeo is not an exception to the general rule that cars have four wheels. In fact, for a correct behaviour of a proof procedure it was necessary to specify for each artery explicitly whether or not it is an exception to the rule. In most applications however, it is unreasonable to expect users to explicitly express all negative information relevant to the employed proof procedure. This problem can be handled by considering a ground literal $\neg P$ proven if an attempt to prove P using SLD resolution has not succeeded. So, in the particular case of the example, it is assumed that the goal clause `not(exception(alfa-romeo))` is proved.

The inference rule that a negative literal is assumed proven when the attempt to prove the complementary literal has failed is called *negation as failure*. Negation as failure is similar to the so-called *closed-world assumption* which is quite common in database applications. In PROLOG, an even stronger assumption, known as negation as *finite* failure, is made by taking $\neg P$ proven only if proving P using SLD resolution has failed in a finite number of steps. The PROLOG predicate `not` is the implementation of this negation as finite failure and therefore should not be taken as the ordinary negation: it is an extra-logical feature of PROLOG.

2.8 Applying logic for building intelligent systems

In the preceding sections, much space has been devoted to the many technical details of knowledge representation and automated reasoning using logic. In the present section, we shall indicate how logic can actually be used for building a logic-based intelligent system. As logic offers suitable basis for model-based systems, more about the use of logic in the context of intelligent systems will said in Chapter 6.

In the foregoing, we have seen that propositional logic offers rather limited expressiveness, which in fact is too limited for most real-life applications. First-order predicate logic offers much more expressive power, but that alone does not yet render the formalism suitable for building intelligent systems. There are some problems: any automated reasoning method for full first-order logic is doomed to have a worst-case time complexity at least as bad as that of checking satisfiability in propositional logic, which is known to be NP-complete (this means that no one has been able to come up with a better deterministic algorithm than an exponentially time-bounded one, although it has not been proven that better ones do not exist). Furthermore, we know that first-order predicate logic is undecidable; so, it is not even sure that an algorithm for checking satisfiability will actually terminate. Fortunately, the circumstances are not always as bad as that. A worst-case characterization seldom gives a realistic indication of the time an algorithm generally will spend on solving an arbitrary problem. Moreover, several suitable syntactic restrictions on first-order formulas have been formulated from which a substantial improvement of the time complexity of the algorithm is obtained; the Horn clause format we have paid attention to is one such restriction.

Since syntactic restrictions are only acceptable as far as permitted by a problem domain, we consider some examples, such as the logical circuit introduced in Chapter 1. However, first we discuss special standard predicates that are often used in modelling practical applications.

2.8.1 Reasoning with equality and ordering predicates

The special binary predicate symbols $>$ (*ordering predicate*) and $=$ *equality predicate* are normally specified in infix position, since this is normal mathematical practice. Both equality and the ordering predicates have a special meaning, which is described by means of a collection of axioms. The meaning of the equality predicate is defined by means of the following four axioms:

E_1 (*reflexivity*): $\forall x(x = x)$

E_2 (*symmetry*): $\forall x \forall y(x = y \rightarrow y = x)$

E_3 (*transitivity*): $\forall x \forall y \forall z(x = y \land y = z \rightarrow x = z)$

E_4 (*substitutivity*): $\forall x_1 \ldots \forall x_n \forall y_1 \ldots \forall y_n((x_1 = y_1 \land \ldots \land x_n = y_n) \rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n))$, and $\forall x_1 \ldots \forall x_n \forall y_1 \ldots \forall y_n((x_1 = y_1 \land \ldots \land x_n = y_n \land P(x_1, \ldots, x_n)) \rightarrow P(y_1, \ldots, y_n))$

Axiom E_1 states that each term in the domain of discourse is equal to itself; axiom E_2 expresses that the order of the arguments of the equality predicate is irrelevant. Axiom E_3 furthermore states that two terms which are equal to some common term, are equal to each other. Note that axiom E_2 follows from the axioms E_1 and E_3; nevertheless, it is usually mentioned explicitly. The three axioms E_1, E_2 and E_3 together imply that equality is an *equivalence relation*. Addition of axiom E_4 renders it a *congruence relation*. The first part of axiom E_4 states that equality is preserved under the application of a function; the second part expresses that equal terms may be substituted for each other in formulas.

EXAMPLE 2.42

Consider the following set of clauses S:

$$S = \{\neg P(f(x), y) \lor Q(x, x), P(f(a), a), a = b\}$$

Suppose that, in addition, we have the equality axioms. If we add the clause $\neg Q(b, b)$ to S, the resulting set of clauses will be unsatisfiable. This can easily be seen informally as follows: we have $P(f(a), a) \equiv P(f(b), a)$ using the given clause $a = b$ and the equality axiom E_4. Now, we replace the atom $P(f(a), a)$ by the equivalent atom $P(f(b), a)$ and apply binary resolution.

The explicit addition of the equality axioms to the other formulas in a knowledge base suffices for rendering equality available for use in an intelligent system. However, it is well known that proving theorems in the presence of the equality axioms can be very inefficient, since many redundant clauses may be generated using resolution. Again, several refinements of the (extended) resolution principle have been developed to overcome the inefficiency problem. For dealing with equality, the resolution principle has for example been extended with an extra inference rule: *paramodulation*. Informally speaking, the principle of paramodulation is the following: if clause C contains a term t and if we have a clause $t = s$, then derive a clause by substituting s for a single occurrence of t in C. Therefore, in practical realizations equality is often only present implicitly in the knowledge base, that is, it is used as a 'built-in' predicate.

Another, more restrictive way to deal with equality is *demodulation*, which adds directionality to equality, meaning that one side of the equality may be replaces (rewitten) by the other side, but not the other way around.

Definition 2.32 *A* demodulator *is a positive unit clause with equality predicate of the form* $(l = r)$*, where* l *and* r *are terms. Let* $C \lor L^t$ *a clause; where* L^t *indicates that the literal* L *contains the term* t*. Let* σ *be a substitution such that* $l\sigma = t$*, then* demodulatie *is defined as the inference rule:*

$$\frac{C \lor L^t, (l = r)}{C \lor L^{t \rightarrow r\sigma}}$$

where $L^{t \rightarrow r\sigma}$ *indicates that term* t *is rewritten to* $r\sigma$.

Thus a demodulator $(l = r)$ can be interpreted as a *rewrite rule* $l \rightarrow r$ with a particular orientation (here from left to right).

EXAMPLE 2.43

Consider the demodulator

$$(brother(father(x)) = uncle(x))$$

and the clause

$$(age(brother(father(John))) = 55)$$

Application of the demodulator yields

$$(age(uncle(John)) = 55)$$

The applied substitution was $\sigma = \{Jan/x\}$.

In many real-life applications, a universally quantified variable ranges over a finite domain $D = \{c_i \mid i = 1, \ldots, n, n \geq 0\}$. The following property usually is satisfied: $\forall x(x = c_1 \vee x = c_2 \vee \ldots \vee x = c_n)$, with $c_i \neq c_j$ if $i \neq j$. This property is known as the *unique names assumption*; from this assumption we have that objects with different names are different.

EXAMPLE 2.44

Consider the following set of clauses S:

$$S = \{\neg P(x) \vee x = a\}$$

We suppose that the equality axioms as well as the unique name assumption hold. Now, if we add the clause $P(b)$ to S, we obtain an inconsistency, since the derivable clause $b = a$ contradicts with the unique name assumption.

The ordering predicates $<$ and $>$ define a *total order* on the set of real numbers. They express the usual, mathematical 'less than' and 'greater than' binary relations between real numbers. Their meaning is defined by means of the following axioms:

O_1 (*irreflexivity*): $\forall x \neg(x < x))$

O_2 (*antisymmetry*): $\forall x \forall y(x < y \rightarrow \neg(y < x))$

O_3 (*transitivity*): $\forall x \forall y \forall z((x < y \wedge y < z) \rightarrow x < z)$

O_4 (*trichonomy law*): $\forall x \forall y((x < y \vee x = y \vee x > y)$

Axiom O_1 states that no term is less that itself; axiom O_2 expresses that reversing the order of the arguments of the predicate $<$ reverses the meaning. Axiom O_3 furthermore states that if a term is less than some other term, and this term is less than a third term, then the first term is less than the third one as well. Note that axiom O_2 follows from O_1 and O_3. The axioms O_1, O_2 and O_3 concern the ordering predicate $<$. The axioms for the ordering predicate $>$ are similar to these: we may just substitute $>$ for $<$ to obtain them. Axiom O_4 states that a given term is either less than, equal to or greater than another given term. Again, in practical realizations, these axioms usually are not added explicitly to the knowledge base, but are assumed to be present implicitly as 'built-in' predicates or as evaluable predicates, that after it has been checked that all variables are instantiated to constants, are evaluated as an expression, returning true or false.

2.8.2 Reasoning models

In this section, a number of examples will be discussed in order to give the reader a feeling how real-world models look like. Of course, given the space available, the examples discussed are still toy examples, but nevertheless inspired by the realistic problems.

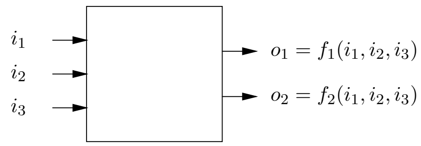


Figure 2.10: Black-box logical circuit.

Functional models

In Chapter 1 we have discussed logical circuits as examples of deep knowledge. Here, we study a logical formalisation of this circuit and dicuss ways in which a specification of a logical circuit can be used for simulation purposes.

A logical circuit can be considered a black-box with *inputs* $i_1, \ldots, i_n$ and *outputs* $o_1, \ldots, o_m$, such that $o_k = f_k(i_1, \ldots, i_n)$, as shown in Figure 2.10. For simplicities sake it is assumed that i_j and o_k are Boolian variables which can take values *false* and *true*, normally in this context of Boolean logic represented as 0 and 1. Hence, the functions f_k are Boolean functions of n Boolean variables. The following components make up the internals of a circuit:

- *gates*, and
- *wires*, connecting the gates.

Here, the following gates with two inputs and one output are distinguished:

- the AND-gate,
- the OR-gate,
- the XOR-gate ('exclusive' OR).

In addition, we have the NOT-gate (or 'inverter') which has one input and one output. Logical gates are drawn using pictograms as shown in Figure 2.11.

Let $\mathcal{L} = (C, A, B, I, O)$ be a *logical circuit*, where C denotes a set of components, A their properties, and B represents the wires; I and O are sets of inputs and outputs, respectively. If $x \in C$ is a components, then, for example, $Andg(x)$ indices an AND-gate.

Input and outputs are specified as follows:

- $i(k, x)$ is the kth input of gate x;
- $o(k, x)$ is the kth output of gate x.

The wires B are represented by the predicate symbol

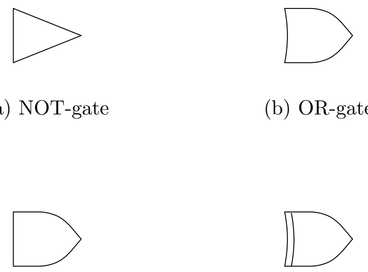
$Connects(x, y)$

indicating that gates x and y are connected by a wire. Values of input or output signals are specified as follows

$v(k) = b$

indicating that input or output k transmits a signal with value b. For example, $v(i(1, a)) = 1$ indicates that the first input of component a is equal to 1.

The properties of the gates can now be specified as follows:



(a) NOT-gate (b) OR-gate

(c) AND-gate (d) XOR-gate

Figure 2.11: Pictograms of gates.

$$\begin{array}{l} \forall x((Andg(x) \wedge \\ \quad (v(i(1,x)) = 1) \wedge \\ \quad (v(i(2,x)) = 1)) \\ \quad \rightarrow \\ \quad (v(o(1,x)) = 1)) \end{array}$$

$$\begin{array}{l} \forall x \forall n((Andg(x) \wedge \\ \quad (v(i(n,x)) = 0)) \\ \quad \rightarrow \\ \quad (v(o(1,x)) = 0)) \end{array}$$

$$\begin{array}{l} \forall x \forall n(Org(x) \wedge \\ \quad (v(i(n,x)) = 1)) \\ \quad \rightarrow \\ \quad (v(o(1,x)) = 1)) \end{array}$$

$$\begin{array}{l} \forall x((Org(x) \wedge \\ \quad (v(i(1,x)) = 0) \wedge \\ \quad (v(i(2,x)) = 0)) \\ \quad \rightarrow \\ \quad (v(o(1,x)) = 0)) \end{array}$$

$$\begin{array}{l} \forall x \forall z((Xorg(x) \wedge \\ \quad (v(i(1,x)) = z) \wedge \\ \quad (v(i(2,x)) = z)) \\ \quad \rightarrow \\ \quad (v(o(1,x)) = 0)) \end{array}$$

$$\begin{array}{l} \forall x \forall y \forall z((Xorg(x) \wedge \\ \quad (v(i(1,x)) = y) \wedge \\ \quad (v(i(2,x)) = z) \wedge \\ \quad (y \neq z)) \end{array}$$

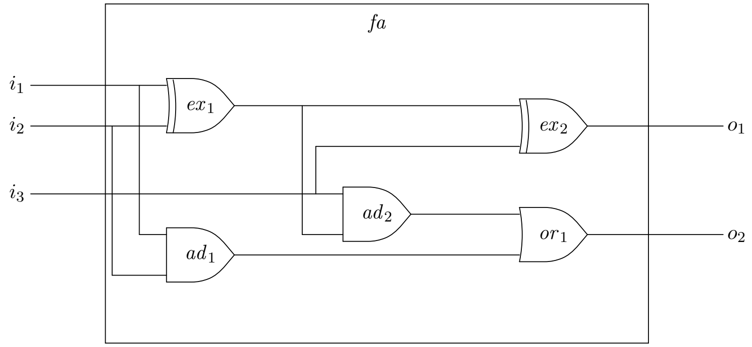


Figure 2.12: The full adder.

$$\begin{array}{l} \rightarrow \\ (v(o(1,x)) = 1)) \end{array}$$

The binary predicate symbol *Connects* is used to specify properties of wires, and these properties are also part of the set A:

$$\begin{array}{l} \forall x \forall y \forall z((Connects(x,y) \wedge \\ \quad (v(x) = z)) \\ \quad \rightarrow \\ \quad (v(y) = z)) \end{array}$$

Now, let us again consider the full-adder circuit already introduced in Chapter 1.

EXAMPLE 2.45

The full-adder shown in Figure 2.12 consists of the following gates:

$$\begin{array}{l} Adder(fa) \\ Xorg(ex_1) \\ Xorg(ex_2) \\ Andg(ad_1) \\ Andg(ad_2) \\ Org(or_1) \end{array}$$

where $Adder(fa)$; represents the entire full-adder, i.e. the back-box as shown in Figure 2.10. The three inputs to the circuit are specified as follows:

$$\begin{array}{l} v(i(1,fa)) = 1 \\ v(i(2,fa)) = 0 \\ v(i(3,fa)) = 1 \end{array}$$

i.e. we have that $i_1 = 1$, $i_2 = 0$ and $i_3 = 1$. Similarly, the outputs o_1 and o_2 are represented as follows:

$v(o(1, fa)) = 0$
$v(o(2, fa)) = 1$

Hence, it holds that $o_1 = 0$ and $o_2 = 1$. Finally, we need to specify the wires connecting the gates:

$Connects(i(1, fa), i(1, ex_1))$
$Connects(i(2, fa), i(2, ex_1))$
$Connects(i(1, fa), i(1, ad_1))$
$Connects(i(2, fa), i(2, ad_1))$
$Connects(i(3, fa), i(2, ex_2))$
$Connects(i(3, fa), i(1, ad_2))$
$Connects(o(1, ex_1), i(1, ex_2))$
$Connects(o(1, ex_1), i(2, ad_2))$
$Connects(o(1, ad_2), i(1, or_1))$
$Connects(o(1, ad_1), i(2, or_1))$
$Connects(o(1, ex_2), o(1, fa))$
$Connects(o(1, or_1), o(2, fa))$

Note that we have not indicated that the *Connects* predicate is symmetric, i.e. signals go in one direction only.

An obvious application of a logical circuit is *simulation*. As this would amount to transforming input into output, positive hyperresolution, i.e. a form of bottom-up inference, seems most suitable. We thus obtain the following specification of the problem we need to solve:

$$C \cup B \cup A \cup \{v(i(k, j)) = b \mid \text{for certain } k, j\} \cup \{\neg v(o(x, y)) = z\} \vdash_{\mathcal{P}} \Box$$

Note that $\neg v(o(x, y)) = z$ has to be included in order make sure that the empty clause $\Box$ is derived.

Spatial and anatomical reasoning

In certain fields of medicine, for example neurology, knowledge concerning the anatomy of the human body is of major importance. The form of automated reasoning in which knowledge concerning the anatomy of the human body is applied, is known as *anatomical reasoning* or, more in general, as *spatial reasoning*.

The point of departure for any intelligent system implementing anatomical reasoning, is the axiomatization of the basic anatomical relations. The more precise the description of the anatomical structures must be, the more complex the resulting axiomatization will be. Not always is a precise three-dimensional specification of anatomical relations required. In our approach to anatomical reasoning, it suffices to indicate only that certain anatomical structures are connected to each other in a qualitative way, as axiomatized by the *Connected* predicate. This predicate is defined as a transitive, irreflexive relation, as follows:

$\forall x \forall y \forall z (Connected(x, y) \wedge Connected(y, z) \rightarrow Connected(x, z))$
$\forall x (\neg Connected(x, x))$

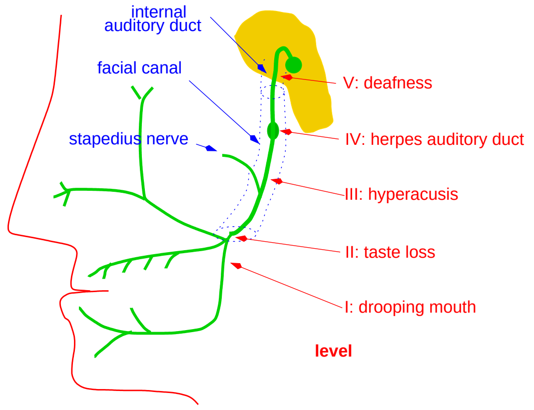


Figure 2.13: Levels in the facial nerve.

Note that the *Connected* predicate is by its transitive and irreflexive properties also antisymmetric. (So, a theorem prover is capable of detecting an inconsistency given the formulas $Connected(a, b)$ and $Connected(b, a)$ from the two axioms given above.)

As a starting point for our discussion concerning the specification of anatomical reasoning in logic, we consider an actual example (the diagnosis of lesions of the facial nerve) taken from a textbook of neurology.

The classical picture of *facial palsy* is well known. These patients have a mouth that droops and may draw to the opposite side. They cannot wrinkle the forehead or close the eye at the affected side. Facial palsy is due to a lesion of the facial nerve (cranial nerve VII); this nerve can be affected by a large variety of disorders. The severity and nature of the complaints and signs that may be observed in the patient depend on the level of facial nerve lesion. Knowledge of the branching pattern of the nerve and the consequences of a lesion of a particular branch is important in diagnostic problem solving. Figure 2.13 gives a schematic overview of the branching pattern of this nerve. The facial nerve is a mixed nerve; it contains motor fibers that supply striated muscle fibers, sensory fibers that carry taste from the anterior two-third of the tongue and some sensation, and parasympathetic fibers.

The facial nerve emerges from the brain stem and leaves the skull via the internal auditory meatus. Next, a small nerve is branched off (the stapedius nerve), that supplies the stapedius muscle (a small muscle that is attached to the ear drum, regulating its tension). The facial nerve proceeds its way through the facial canal (in the temporal bone), next branching off the chorda tympani, a nerve mostly consisting of parasympathetic fibers that supply the submandibular and sublingual glands. The facial nerve leaves the facial canal through the stylomastoid foramen. Finally, it splits up in a number of branches that supply the superficial musculature of the face and scalp (e.g. orbicularis oris et oculi, buccinator, platysma).

A bit simplified, we distinguish the following five levels of facial nerve lesions (consult again Figure 2.13):

Level 1: A lesion outside the stylomastoid foramen produces signs such as drooping of the mouth. The patient cannot whistle, wink, close the eye or wrinkle the forehead. When the patient attempts to close the eye, the eye bulb will turn upward (*Bell's sign*).

Level 2: A lesion of the nerve in its course through the facial canal will result in all the signs as present in a level 1 lesion, but in addition there is reduced salivation (lesion of the chorda tympani) and loss of taste in the anterior two-thirds of the tongue.

Level 3: All signs of a level 2 lesion are present, but in addition the stapedius nerve is affected, causing hyperacusis (due to paralysis of the stapedius muscle).

Level 4: A lesion of the geniculate ganglion is usually due to herpes zoster, in which case herpetic lesions are visible on the ear drum and external auditory canal (*Ramsay Hunt syndrome*). Typically, a patient will experience pain in and behind the ear.

Level 5: A lesion in the internal auditory meatus is usually associated with acoustic nerve (cranial nerve VIII) involvement, because the last nerve also runs through this canal. In addition to the signs mentioned for lower level lesions, deafness will be present.

This completes the description of the neurological knowledge involved in diagnosing facial nerve lesions.

In formalizing this knowledge using first-order predicate logic, we start by completing the axiomatization of the anatomical relationships by giving a domain-specific fill-in for the *Connected* predicate. The atom $Connected(x, y)$ is intended to mean that the facial nerve runs from level x up to level y:

$$\begin{array}{l} Connected(stylomastoid\_foramen, chorda\_tympani) \\ Connected(chorda\_tympani, stapedius\_nerve) \\ Connected(stapedius\_nerve, geniculate\_ganglion) \\ Connected(geniculate\_ganglion, internal\_auditory\_meatus) \end{array}$$

Note that we have employed anatomical terms to denote the various levels.

To relate anatomical structures and signs that may arise due to facial nerve lesion, we have to express that the signs associated with a lesion at a certain level x includes all the signs of a lesion at a lower level y:

$$\forall x \forall y (Lesion(x) \wedge Connected(y, x) \rightarrow Lesion(y))$$

This completes our axiomatization of the knowledge that forms the basis of logical anatomical reasoning.

We next specify the relationship between a lesion at a certain level and the specific anatomical structures that will be affected by this lesion, expressed by the unary predicate symbol *Affected*. We use a bi-implication, because given a lesion at a certain level we may want to know which structures will be affected by this lesion; on the other hand, the observation of malfunction of certain structures may be interpreted as evidence for a lesion at a certain level (in Chapter 6 will see that it is more natural to use abduction for diagnostic reasoning using causal knowledge):

$$\begin{array}{l} (Lesion(stylomastoid\_foramen) \leftrightarrow \\ \quad (Affected(orbicularis\_oris) \wedge \\ \quad\; Affected(orbicularis\_oculi) \wedge \\ \quad\; Affected(buccinator) \wedge \\ \quad\; Affected(frontalis\_muscle) \wedge \\ \quad\; Affected(platysma))) \end{array}$$

$$\begin{array}{l} (Lesion(chorda\_tympani) \leftrightarrow \\ \quad (Affected(sensory\_taste\_fibers) \wedge \\ \quad\; Affected(sublingual\_gland) \wedge \\ \quad\; Affected(submaxillary\_gland))) \end{array}$$

$$(Lesion(stapedius\_nerve) \leftrightarrow Affected(stapedius\_muscle))$$

$$(Lesion(geniculate\_ganglion) \leftrightarrow Affected(sensory\_fibers\_ear))$$

$$(Lesion(internal\_auditory\_meatus) \leftrightarrow Affected(acoustic\_nerve))$$

Finally, paralysis of certain muscles and disturbed sensation will give rise to specific signs and complaints in the patient. This knowledge is again expressed using a collection of bi-implications:

$$\begin{array}{l} (Affected(orbicularis\_oris) \leftrightarrow (Sign(mouth\_droops) \wedge \\ \quad Sign(cannot\_whistle))) \end{array}$$

$$(Affected(orbicularis\_oculi) \leftrightarrow (Sign(cannot\_close\_eyes) \wedge Sign(Bell)))$$

$$(Affected(buccinator) \leftrightarrow Sign(flaccid\_cheeks))$$

$$(Affected(frontalis\_muscle) \leftrightarrow Sign(cannot\_wrinkle\_forehead))$$

$$(Affected(platysma) \leftrightarrow Sign(paresis\_superficial\_neck\_musculature))$$

$$\begin{array}{l} ((Affected(sublingual\_gland) \wedge Affected(submaxillary\_gland)) \leftrightarrow \\ \quad Complaint(dry\_mouth)) \end{array}$$

$$(Affected(sensory\_taste\_fibers) \leftrightarrow Complaint(taste\_loss\_anterior\_part\_tongue))$$

$$(Affected(stapedius\_muscle) \leftrightarrow Complaint(hyperacusis))$$

$$\begin{array}{l} (Affected(sensory\_fibers\_ear) \leftrightarrow \\ \quad (Complaint(pain\_behind\_ear) \wedge \\ \quad\; Complaint(pain\_within\_ear) \wedge \\ \quad\; Sign(herpetic\_lesions))) \end{array}$$

$$(Affected(acoustic\_nerve) \leftrightarrow Complaint(deafness))$$

Let T be the logical theory given above. For example, after automatic translation to clausal form, using hyperresolution we are now capable to derive:

$$T \cup \{Lesion(stapedius\_nerve)\} \cup \{\neg Sign(x)\} \cup \{\neg Complaint(y)\} \vdash_{\mathcal{P}} \Box$$

where for x we have *mouth_droops*, *cannot_whistle*, *cannot_close_eyes*, *Bell*, *flacid_cheeks*, *cannot_wrinkle_forehead* and *paresis_superficial_neck_musculature*; for y we have *hyperacusis*, *dry_mouth* and *taste_loss_anterior_part_tongue*. Note that all results but the complaint *hyperacusis* have been derived using the anatomical axioms for the *Connected* predicate. Reasoning from signs and complaints to the level of a facial nerve lesion is also possible (essentially employing the bi-implication), but here we need a meta-level primitive that selects from the unit clause concerning *Lesion* the one specifying knowledge regarding the highest level of the lesion.

In our formalization of the anatomical reasoning model, we did not make a distinction between left- and right-sided lesions of the facial nerve. The extension of the logical theory to include this distinction is straightforward.

Causal reasoning

Although in the previous section we primarily focussed on the logical specification of anatomical relations, implicitly other reasoning models were also involved. Some of the formulas presented in the previous section express some relationship between cause and effect of nerve damage, thus representing *causal knowledge*. The reasoning about such cause–effect relationships is known as *causal reasoning*. In the present section, we shall study the logic of causal reasoning in medicine, taking the logical specification of physiological processes as a point of departure.

The representation of causal knowledge in logic is rather straightforward; it may be represented by means of a collection of logical implications of the following form:

$$cause \rightarrow effect$$

where both *cause* and *effect* are conjunctions of literals. Most literals refer to the *state* of some *parameter*; the states of all parameters together describe the entire physiological process. As an example of a parameter consider the level of a substance in the blood; the actual level of the substance stands for the parameter's state.

A state is either *numeric* or *qualitative*. An example of a numeric state (of parameter level of *sodium* in the *blood*) is expressed by the following unit clause:

$$conc(blood, sodium) = 125$$

In clinical practice, numeric parameters are often changed to qualitative states. In the above case, we get:

$$conc(blood, sodium) = decreased$$

There are several common types of causal reasoning in medicine. We shall study the *negative feedback process* and its logical specification in some detail.

In terms of cause–effect relationships, the global specification of a negative feedback process leads to the following logical theory T (to simplify matters, we have assumed that a cause consists of a single literal):

$$\begin{array}{l} s \rightarrow r_1 \\ r'_1 \rightarrow r_2 \\ \quad\vdots \\ r'_{n-1} \rightarrow r_n \\ r'_n \rightarrow \neg s \end{array}$$

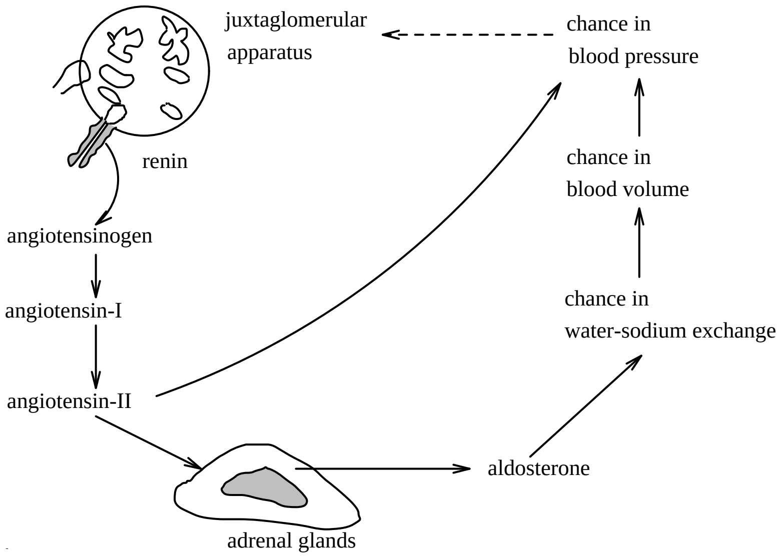


Figure 2.14: The renin–angiotensin–aldosterone system.

where s, r_i, r'_i, $1 \leq i \leq n$, $n \geq 1$ are literals in first-order logic; the literals r_i, r'_i are similar, in the sense that substitution of terms for variables occurring in these literals can make them syntactically equal. Note that we have $T \models \neg s$; in words: the negative feedback is a semantic consequence of the process description.

To investigate the applicability of this approach to formalizing causal reasoning in medicine, we have chosen a particular example of a negative feedback process from the literature, viz. the renin–angiotensin–aldosterone system. We start by giving a brief summary of the medical knowledge involved.

The regulation of the blood pressure is accomplished by the collaborative effort of a number of control systems in the human body. One of these control systems is the *renin–angiotensin–aldosterone system*. Figure 2.14 gives a pictorial overview of this control system. We review the regulatory factors involved.

The proteolytic enzyme *renin* is released by cells of the *juxtaglomerular apparatus* in the kidneys. A decrease in the mean renal arterial pressure increases the renin secretion, and an increase in the mean renal arterial pressure leads to a decrease in the renin secretion. Renin acts on an α_2-globulin (*angiotensinogen*) that circulates in the blood, liberating the decapeptid *angiotensin-I*. In turn, the octapeptid *angiotensin-II* is liberated from *angiotensin-I* by *angiotensin converting enzyme* (ACE) that is produced in the lungs.

Angiotensin-II is a powerful arteriolar constrictor; administration of angiotensin-II leads to an increase in arterial blood pressure. In addition, it stimulates the secretion of *aldosterone*, a hormone produced by the adrenal cortex; an increase in angiotensin-II levels in the blood increases aldosterone levels in the blood. Aldosterone stimulates active reabsorption of Na^+

from the urine and the secretion of K^+ to the urine. Water moves with the reabsorbed Na^+ to the blood, which causes an increase in blood volume. This in turn leads to an increase in blood pressure. Finally, an increase in blood pressure inhibits the secretion of renin. This completes our description.

When considering the physiological process described above in terms of a causal model, we have to analyse its behaviour in terms of causes and effects. We start this formalization by introducing a number of predicate and function symbols and constants that will be used to represent parameters and states. For the representation of the level of a substance in the blood we employ the binary function symbol *conc*. The unary function symbol *pressure* stands for blood pressure. The conversion of one substance into another substance by some enzyme, will be represented by the binary function symbol *conversion*. Finally, we distinguish two constants: *decreased* and *increased* to express the states of various parameters in a qualitative way. Step by step, the text given above will now be translated into a logical theory T.

A decrease in the blood pressure yields an increase in renin blood levels:

$$\mathit{pressure}(\mathit{blood}) = \mathit{decreased} \rightarrow \mathit{conc}(\mathit{blood}, \mathit{renin}) = \mathit{increased}$$

The relationship between renin levels in the blood and conversion of angiotensinogen into angiotensin-I is expressed as follows:

$$\forall v(\mathit{conc}(\mathit{blood}, \mathit{renin}) = v \rightarrow \mathit{conversion}(\mathit{angiotensinogen}, \mathit{angiotensin\_I}) = v)$$

where the universally quantified variable v stands for *increased* or *decreased*.

The relationship between decreased or increased conversion of angiotensinogen into angiotensin-I is represented by means of the following logical implication:

$$\begin{array}{l}\forall v(\mathit{conversion}(\mathit{angiotensinogen}, \mathit{angiotensin\_I}) = v \rightarrow \\ \quad \mathit{conc}(\mathit{blood}, \mathit{angiotensin\_I}) = v)\end{array}$$

A change v in the blood level of angiotensin-I leads to an inverse change in the ACE levels, and a similar change in angiotensin-II levels:

$$\begin{array}{l}\forall v(\mathit{conc}(\mathit{blood}, \mathit{angiotensin\_I}) = v \rightarrow \\ \quad (\neg(\mathit{conc}(\mathit{blood}, \mathrm{ACE}) = v) \wedge \mathit{conversion}(\mathit{angiotensin\_I}, \mathit{angiotensin\_II}) = v))\end{array}$$

$$\begin{array}{l}\forall v(\mathit{conversion}(\mathit{angiotensin\_I}, \mathit{angiotensin\_II}) = v \rightarrow \\ \quad \mathit{conc}(\mathit{blood}, \mathit{angiotensin\_II}) = v)\end{array}$$

Angiotensin-II produces arterial vasoconstriction and an increase in aldosterone levels:

$$\begin{array}{l}\mathit{conc}(\mathit{blood}, \mathit{angiotensin\_II}) = \mathit{increased} \\ \rightarrow \\ (\mathit{Vasoconstriction}(\mathit{arteries}, \mathit{peripheral}) \wedge \\ \ \mathit{conc}(\mathit{blood}, \mathit{aldosterone}) = \mathit{increased})\end{array}$$

Arterial vasoconstriction produces an increase in bloodpressure:

$$\mathit{Vasoconstriction}(\mathit{arteries}, \mathit{peripheral}) \rightarrow \mathit{pressure}(\mathit{blood}) = \mathit{increased}$$

An increase in the aldosterone levels results in an increase of blood sodium and a decrease of the potassium levels:

$$\begin{array}{l}\mathit{conc}(\mathit{blood}, \mathit{aldosterone}) = \mathit{increased} \rightarrow \\ (\mathit{conc}(\mathit{blood}, \mathit{sodium}) = \mathit{increased} \wedge \mathit{conc}(\mathit{blood}, \mathit{potassium}) = \mathit{decreased})\end{array}$$

The reabsorption of sodium is accompanied by the reabsorption of water, causing an increase in blood volume; more in general, a change in sodium level causes a change in blood volume:

$$\forall v\ (\mathit{conc}(\mathit{blood}, \mathit{sodium}) = v \rightarrow \mathit{volume}(\mathit{blood}) = v)$$

The change in blood volume causes a similar change in cardiac output:

$$\forall v(\mathit{volume}(\mathit{blood}) = v \rightarrow \mathit{output}(\mathit{heart}) = v)$$

A change in cardiac output causes the same change in blood pressure:

$$\forall v(\mathit{output}(\mathit{heart}) = v \rightarrow \mathit{pressure}(\mathit{blood}) = v)$$

A change in bloodpressure causes an inverse change in renin levels:

$$\forall v(\mathit{pressure}(\mathit{blood}) = v \rightarrow \neg(\mathit{conc}(\mathit{blood}, \mathit{renin}) = v))$$

Finally, we need to express that 'increased' and 'decreased' are different notions:

$$\neg(\mathit{increased} = \mathit{decreased})$$

This completes our formalization of the causal knowledge concerning the renin–angiotensin–aldosterone system.

After (automatic) conversion of this logical theory T to clausal form, a resolution-based theorem prover is capable of deriving in six steps

$$T \cup \{\mathit{pressure}(\mathit{blood}) = \mathit{decreased}\} \vdash \Box$$

the last step being the derivation of

$$\mathit{pressure}(\mathit{blood}) = \mathit{increased}$$

among others via the intermediate derivation of

$$\mathit{Vasoconstriction}(\mathit{arteries}, \mathit{peripheral})$$

yielding a contradiction with

$$\mathit{pressure}(\mathit{blood}) = \mathit{decreased}$$

2.9 Logic as a representation formalism

Compared to other knowledge representation formalisms in artificial intelligence, logic has the great advantage of having a clear syntax and semantics. A logical deductive system in principle offers a set of inference rules, which is sound and complete: each formula derived using such a set of inference rules has a meaning that is unique in terms of the meaning of the formulas it was derived from. So, logic offers a starting point for studying the foundations of knowledge representation and manipulation.

First-order logic in its pure form however has hardly ever been used as a knowledge representation formalism in intelligent systems. This is partly due to the difficulty of expressing domain knowledge in logical formulas. When in a specific problem domain the knowledge is not available in a form 'close' to logic, a lot of energy has to be invested into converting intelligent knowledge to logical formulas, and in this process often valuable information is lost. Moreover, the type of logic that has been dealt with in this chapter, which is sometimes called *standard logic*, is not suitable for encoding all types of knowledge. For example, reasoning about time, and reasoning about reasoning strategies to be followed, often called meta-inference, cannot be represented directly in first-order predicate logic. Moreover, in standard logic it is not possible to handle incomplete and uncertain information, or to deal adequately with exceptional cases to general rules. Currently however, a lot of research is going on concerning *non-standard logics* for expressing such concepts in a formal way.

Exercises

(2.1) Consider the interpretation $v : PROP \rightarrow \{true, false\}$ in propositional logic, which is defined by $v(P) = false$, $v(Q) = true$ and $v(R) = true$. What is the truth value of the formula $((\neg P) \wedge Q) \vee (P \rightarrow (Q \vee R))$ given this interpretation v?

(2.2) For each of the following formulas in propositional logic determine whether it is valid, invalid, satisfiable, unsatisfiable or a combination of these, using truth tables:

(a) $P \vee (Q \rightarrow \neg P)$

(b) $P \vee (\neg P \wedge Q \wedge R)$

(c) $P \rightarrow \neg P$

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

(e) $(P \rightarrow Q) \rightarrow (Q \rightarrow P)$

(2.3) Suppose that $F_1, \ldots, F_n$, $n \geq 1$, and G are formulas in propositional logic, such that the formula G is a logical consequence of $\{F_1, \ldots, F_n\}$. Construct the truth table of the implication $F_1 \wedge \cdots \wedge F_n \rightarrow G$. What do you call such a formula?

(2.4) Prove the following statements using the laws of equivalence for propositional logic:

(a) $P \rightarrow Q \equiv \neg P \rightarrow \neg Q$

(b) $P \rightarrow (Q \rightarrow R \equiv (P \wedge Q) \rightarrow R$

(c) $(P \wedge \neg Q) \rightarrow R \equiv (P \wedge \neg R) \rightarrow Q$

(d) $P \vee (\neg Q \vee R) \equiv (\neg P \wedge Q) \rightarrow R$

(2.5) Prove that the proposition $((P \rightarrow Q) \rightarrow P) \rightarrow P$, known as Peirce's law, is a tautology, using the laws of equivalence in propositional logic and the property that for any propositions π and ϕ, the formula $\pi \vee \neg\phi \vee \phi$ is a tautology.

(2.6) In each of the following cases, we restrict ourselves to a form of propositional logic only offering a limited set of logical connectives. Prove by means of the laws of equivalence that every formula in full propositional logic can be translated into a formula only containing the given connectives:

(a) the connectives $\neg$ and $\vee$.

Table 2.6: Meaning of Sheffer stroke.

F	G	$F\|G$
true	*true*	*false*
true	*false*	*true*
false	*true*	*true*
false	*false*	*true*

(b) the connective $|$ which is known as the Sheffer stroke; its meaning is defined by the truth table given in Table 2.6.

(2.7) Consider the following formula in first-order predicate logic: $\forall x(P(x) \vee Q(y))$. Suppose that the following structure

$$S = (\{2, 3\}, \varnothing, \{A : \{2, 3\} \rightarrow \{true, false\}, B : \{2, 3\} \rightarrow \{true, false\})$$

is given. The predicates A and B are associated with the predicate symbols P and Q, respectively. Now, define the predicates A and B, and a valuation v in such a way that the given formula is satisfied in the given structure S and valuation v.

(2.8) Consider the following statements. If a statement is correct, then prove its correctness using the laws of equivalence; if it is not correct, then give a counterexample.

(a) $\forall x P(x) \equiv \neg\exists x \neg P(x)$

(b) $\forall x \exists y P(x, y) \equiv \forall y \exists x P(x, y)$

(c) $\exists x(P(x) \rightarrow Q(x)) \equiv \forall x P(x) \rightarrow \exists x Q(x)$

(d) $\forall x(P(x) \vee Q(x)) \equiv \forall x P(x) \vee \forall x Q(x)$

(2.9) Transform the following formulas into the clausal form of logic:

(a) $\forall x \forall y \exists z(P(z, y) \wedge (\neg P(x, z) \rightarrow Q(x, y)))$

(b) $\exists x(P(x) \rightarrow Q(x)) \wedge \forall x(Q(x) \rightarrow R(x)) \wedge P(a)$

(c) $\forall x(\exists y(P(y) \wedge R(x, y)) \rightarrow \exists y(Q(y) \wedge R(x, y)))$

(2.10) For each of the following sets of clauses, determine whether or not it is satisfiable. If a given set is unsatisfiable, then give a refutation of the set using binary resolution; otherwise give an interpretation satisfying it:

(a) $\{\neg P \vee Q, P \vee \neg R, \neg Q, \neg R\}$

(b) $\{\neg P \vee Q \vee R, \neg Q \vee S, P \vee S, \neg R, \neg S\}$

(c) $\{P \vee Q, \neg P \vee Q, P \vee \neg Q, \neg P \vee \neg Q\}$

(d) $\{P \vee \neg Q, Q \vee R \vee \neg P, Q \vee P, \neg P\}$

(2.11) Let E be an expression and let σ and θ be substitutions. Prove that $E(\sigma\theta) = (E\sigma)\theta$.

(2.12) For each of the following sets of expressions, determine whether or not it is unifiable. If a given set if unifiable, then compute a most general unifier:

(a) $\{P(a, x, f(x)), P(x, y, x)\}$

(b) $\{P(x, f(y), y), P(w, z, g(a, b))\}$

(c) $\{P(x, z, y), P(x, z, x), P(a, x, x)\}$

(d) $\{P(z, f(x), b), P(x, f(a), b), P(g(x), f(a), y)\}$

(2.13) Use binary resolution to show that each one of the following sets of clauses is unsatisfiable:

(a) $\{P(x, y) \vee Q(a, f(y)) \vee P(a, g(z)), \neg P(a, g(x)) \vee Q(a, f(g(b))), \neg Q(x, y)\}$

(b) $\{append(nil, x, x), append(cons(x, y), z, cons(x, u)) \vee \neg append(y, z, u),$
$\neg append(cons(1, cons(2, nil)), cons(3, nil), x)\}$

(c) $\{R(x, x), R(x, y) \vee \neg R(y, x), R(x, y) \vee \neg R(x, z) \vee \neg R(z, y), R(a, b), \neg R(b, a)\}$

Remark. The first three clauses in exercise (c) define an equivalence relation.

(2.14) Consider the set of clauses $\{\neg P, P \vee Q, \neg Q, R\}$. We employ the set-of-support resolution strategy. Why do we not achieve a refutation if we set the set of support initially to the clause R?

(2.15) Develop a logic knowledge base for a problem domain you are familiar with.