First-order Predicate Logic

3

Many practical, relevant problems cannot be or can only very inconveniently be formulated in the language of propositional logic, as we can easily recognize in the following example. The statement

"*Robot 7 is situated at the xy position (35, 79)*"

can in fact be directly used as the propositional logic variable

"*Robot_7_is_situated_at_xy_position_(35, 79)*"

for reasoning with propositional logic, but reasoning with this kind of proposition is very inconvenient. Assume 100 of these robots can stop anywhere on a grid of 100×100 points. To describe every position of every robot, we would need $100 \cdot 100 \cdot 100 = 1\,000\,000 = 10^6$ different variables. The definition of relationships between objects (here robots) becomes truly difficult. The relation

"*Robot A is to the right of robot B*."

is semantically nothing more than a set of pairs. Of the 10 000 possible pairs of x-coordinates there are $(99 \cdot 98)/2 = 4851$ ordered pairs. Together with all 10 000 combinations of possible y-values for both robots, there are $(100 \cdot 99) = 9900$ formulas of the type

$$\begin{aligned} &Robot\_7\_is\_to\_the\_right\_of\_robot\_12 \Leftrightarrow \\ &\quad Robot\_7\_is\_situated\_at\_xy\_position\_(35, 79) \\ &\quad \wedge Robot\_12\_is\_situated\_at\_xy\_position\_(10, 93) \vee \ldots \end{aligned}$$

defining these relations, each of them with $(10^4)^2 \cdot 0.485 = 0.485 \cdot 10^8$ alternatives on the right side. In first-order predicate logic, we can define a predicate

W. Ertel, *Introduction to Artificial Intelligence*, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-58487-4_3

Position(*number*, *xPosition*, *yPosition*). The above relation must no longer be enumerated as a huge number of pairs, rather it is described abstractly with a rule of the form

$$\forall u \forall v\, is\_further\_right(u, v) \Leftrightarrow \\ \exists x_u \exists y_u \exists x_v \exists y_v\, position(u, x_u, y_u) \wedge position(v, x_v, y_v) \wedge x_u > x_v,$$

Where $\forall u$ is read as "for every u" and $\exists v$ as "there exists v".

In this chapter we will define the syntax and semantics of *first-order predicate logic* (*PL1*), show that many applications can be modeled with this language and that there is a complete and sound calculus for this language.

3.1 Syntax

First we solidify the syntactic structure of terms.

Definition 3.1 Let V be a set of variables, K a set of constants, and F a set of function symbols. The sets V, K and F are pairwise disjoint. We define the set of *terms* recursively:

- All variables and constants are (atomic) terms.
- If $t_1, \ldots, t_n$ are terms and f an n-place function symbol, then $f(t_1, \ldots, t_n)$ is also a term.

Some examples of terms are $f(\sin(\ln(3)), \exp(x))$ and $g(g(g(x)))$. To be able to establish logical relationships between terms, we build formulas from terms.

Definition 3.2 Let P be a set of predicate symbols. *Predicate logic formulas* are built as follows:

- If $t_1, \ldots, t_n$ are terms and p an n-place predicate symbol, then $p(t_1, \ldots, t_n)$ is an (atomic) formula.
- If A and B are formulas, then $\neg A$, (A), $A \wedge B$, $A \vee B$, $A \Rightarrow B$, $A \Leftrightarrow B$ are also formulas.
- If x is a variable and A a formula, then $\forall x\, A$ and $\exists x\, A$ are also formulas. $\forall$ is the universal quantifier and $\exists$ the existential quantifier.
- $p(t_1, \ldots, t_n)$ and $\neg p(t_1, \ldots, t_n)$ are called literals.

Table 3.1 Examples of formulas in first-order predicate logic. Please note that *mother* here is a function symbol

Formula	Description
$\forall x\ frog(x) \Rightarrow green(x)$	All frogs are green
$\forall x\ frog(x) \wedge brown(x) \Rightarrow big(x)$	All brown frogs are big
$\forall x\ likes(x, cake)$	Everyone likes cake
$\neg\forall x\ likes(x, cake)$	Not everyone likes cake
$\neg\exists x\ likes(x, cake)$	No one likes cake
$\exists x\ \forall y\ likes(y, x)$	There is something that everyone likes
$\exists x\ \forall y\ likes(x, y)$	There is someone who likes everything
$\forall x\ \exists y\ likes(y, x)$	Everything is loved by someone
$\forall x\ \exists y\ likes(x, y)$	Everyone likes something
$\forall x\ customer(x) \Rightarrow likes(bob, x)$	Bob likes every customer
$\exists x\ customer(x) \wedge likes(x, bob)$	There is a customer whom bob likes
$\exists x\ baker(x) \wedge \forall y\ customer(y) \Rightarrow likes(x, y)$	There is a baker who likes all of his customers
$\forall x\ older(mother(x), x)$	Every mother is older than her child
$\forall x\ older(mother(mother(x)), x)$	Every grandmother is older than her daughter's child
$\forall x\ \forall y\ \forall z\ rel(x, y) \wedge rel(y, z) \Rightarrow rel(x, z)$	*rel* is a transitive relation

- Formulas in which every variable is in the scope of a quantifier are called *first-order sentences* or *closed formulas*. Variables which are not in the scope of a quantifier are called *free variables*.
- Definitions 2.8 (CNF) and 2.10 (Horn clauses) hold for formulas of predicate logic literals analogously.

In Table 3.1 several examples of PL1 formulas are given along with their intuitive interpretations.

3.2 Semantics

In propositional logic, every variable is directly assigned a truth value by an interpretation. In predicate logic, the meaning of formulas is recursively defined over the construction of the formula, in that we first assign constants, variables, and function symbols to objects in the real world.

Definition 3.3 An *interpretation* $\mathbb{I}$ is defined as
- A mapping from the set of constants and variables $K \cup V$ to a set W of names of objects in the world.
- A mapping from the set of function symbols to the set of functions in the world. Every n-place function symbol is assigned an n-place function.
- A mapping from the set of predicate symbols to the set of relations in the world. Every n-place predicate symbol is assigned an n-place relation.

Example 3.1 Let c_1, c_2, c_3 be constants, "*plus*" a two-place function symbol, and "*gr*" a two-place predicate symbol. The truth of the formula

$$F \equiv gr(plus(c_1, c_3), c_2)$$

depends on the interpretation $\mathbb{I}$. We first choose the following obvious interpretation of constants, the function, and of the predicates in the natural numbers:

$$\mathbb{I}_1\colon\ c_1 \mapsto 1,\ c_2 \mapsto 2,\ c_3 \mapsto 3, \quad plus \mapsto +, \quad gr \mapsto > .$$

Thus the formula is mapped to

$$1 + 3 > 2, \quad \text{or after evaluation} \quad 4 > 2.$$

The greater-than relation on the set {1, 2, 3, 4} is the set of all pairs (x, y) of numbers with $x > y$, meaning the set $G = \{(4, 3), (4, 2), (4, 1), (3, 2), (3, 1), (2, 1)\}$. Because $(4, 2) \in G$, the formula F is true under the interpretation $\mathbb{I}_1$. However, if we choose the interpretation

$$\mathbb{I}_2\colon\ c_1 \mapsto 2,\ c_2 \mapsto 3,\ c_3 \mapsto 1, \quad plus \mapsto -, \quad gr \mapsto > ,$$

we obtain

$$2 - 1 > 3, \quad \text{or} \quad 1 > 3.$$

The pair (1, 3) is not a member of G. The formula F is false under the interpretation $\mathbb{I}_2$. Obviously, the truth of a formula in PL1 depends on the interpretation. Now, after this preview, we define truth.

Definition 3.4

- An atomic formula $p(t_1, \ldots, t_n)$ is *true* (or valid) under the interpretation $\mathbb{I}$ if, after interpretation and evaluation of all terms $t_1, \ldots, t_n$ and interpretation of the predicate p through the n-place relation r, it holds that

$$(\mathbb{I}(t_1), \ldots, \mathbb{I}(t_n)) \in r.$$

- The truth of quantifierless formulas follows from the truth of atomic formulas—as in propositional calculus—through the semantics of the logical operators defined in Table 2.1 on page 25.
- A formula $\forall x\, F$ is true under the interpretation $\mathbb{I}$ exactly when it is true given an arbitrary change of the interpretation for the variable x (and only for x)
- A formula $\exists x\, F$ is true under the interpretation $\mathbb{I}$ exactly when there is an interpretation for x which makes the formula true.

The definitions of semantic equivalence of formulas, for the concepts satisfiable, true, unsatisfiable, and model, along with semantic entailment (Definitions 2.4, 2.5, 2.6) carry over unchanged from propositional calculus to predicate logic.

Theorem 3.1 *Theorems* 2.2 (*deduction theorem*) *and* 2.3 (*proof by contradiction*) *hold analogously for PL*1.

Example 3.2 The family tree given in Fig. 3.1 graphically represents (in the semantic level) the relation

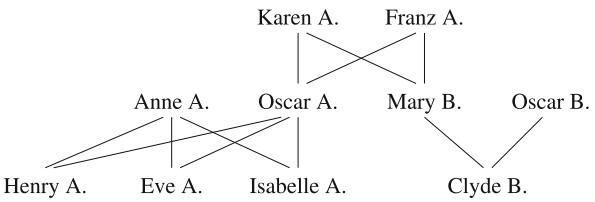

Fig. 3.1 A family tree. The edges going from Clyde B. upward to Mary B. and Oscar B. represent the element (Clyde B., Mary B., Oscar B.) as a child relationship

$$\begin{aligned}\text{Child} = \{&(\text{Oscar A., Karen A., Frank A.}), \quad (\text{Mary B., Karen A., Frank A.}),\\ &(\text{Henry A., Anne A., Oscar A.}), \quad (\text{Eve A., Anne A., Oscar A.}),\\ &(\text{Isabelle A., Anne A., Oscar A.}), \quad (\text{Clyde B., Mary B., Oscar B.})\}\end{aligned}$$

For example, the triple (Oscar A., Karen A., Frank A.) stands for the proposition "*Oscar A. is a child of Karen A. and Frank A.*". From the names we read off the one-place relation

$$\text{Female} = \{\text{Karen A., Anne A., Mary B., Eve A., Isabelle A.}\}$$

of the women. We now want to establish formulas for family relationships. First we define a three-place predicate *child*(x, y, z) with the semantic

$$\mathbb{I}(child(x, y, z)) = w \equiv (\mathbb{I}(x), \mathbb{I}(y), \mathbb{I}(z)) \in \text{Kind}.$$

Under the interpretation $\mathbb{I}(oscar) = \text{Oscar A.}$, $\mathbb{I}(eve) = \text{Eve A.}$, $\mathbb{I}(anne) = \text{Anne A.}$, it is also true that *child*(*eve*, *anne*, *oscar*). For *child*(*eve*, *oscar*, *anne*) to be true, we require, with

$$\forall x\ \forall y\ \forall z\ child(x, y, z) \ \Leftrightarrow\ child(x, z, y),$$

symmetry of the predicate *child* in the last two arguments. For further definitions we refer to Exercise 3.1 on page 63 and define the predicate *descendant* recursively as

$$\begin{aligned}\forall x\ \forall y\, descendant(x,\ y) \Leftrightarrow\ &\exists z\, child(x,\ y,\ z) \vee\\ &(\exists u\, \exists v\, child(x,\ u,\ v)\ \wedge\ descendant(u,\ y)).\end{aligned}$$

Now we build a small knowledge base with rules and facts. Let

$$\begin{aligned}KB \equiv\ & female(karen)\ \wedge\ female(anne)\ \wedge\ female(mary)\\ &\wedge\ female(eve)\ \wedge\ female(isabelle)\\ &\wedge\ child(oscar,\ karen,\ franz)\ \wedge\ child(mary,\ karen,\ franz)\\ &\wedge\ child(eve,\ anne,\ oscar)\ \wedge\ child(henry,\ anne,\ oscar)\\ &\wedge\ child(isabelle,\ anne,\ oscar)\ \wedge\ child(clyde,\ mary,\ oscarb)\\ &\wedge\ (\forall x\ \forall y\ \forall z\, child(x,\ y,\ z)\ \Rightarrow\ child(x,\ z,\ y))\\ &\wedge\ (\forall x\ \forall y\, descendant(x,\ y)\ \Leftrightarrow\ \exists z\, child(x,\ y,\ z)\\ &\vee\ (\exists u\, \exists v\, child(x,\ u,\ v)\ \wedge\ descendant(u,\ y))).\end{aligned}$$

We can now ask, for example, whether the propositions *child*(*eve*, *oscar*, *anne*) or *descendant*(*eve*, *franz*) are derivable. To that end we require a calculus.

3.2.1 Equality

To be able to compare terms, equality is a very important relation in predicate logic. The equality of terms in mathematics is an equivalence relation, meaning it is reflexive, symmetric and transitive. If we want to use equality in formulas, we must either incorporate these three attributes as axioms in our knowledge base, or we must integrate equality into the calculus. We take the easy way and define a predicate "=" which, deviating from Definition 3.2 on page 40, is written using infix notation as is customary in mathematics. (An equation $x = y$ could of course also be written in the form $eq(x, y)$.) Thus, the equality axioms have the form

$$\begin{array}{rll} \forall x & x = x & (\text{reflexivity}) \\ \forall x\ \forall y & x = y \Rightarrow y = x & (\text{symmetry}) \\ \forall x\ \forall y\ \forall z & x = y \wedge y = z \Rightarrow x = z & (\text{transitivity}). \end{array} \tag{3.1}$$

To guarantee the uniqueness of functions, we additionally require

$$\forall x\ \forall y\ x = y \Rightarrow f(x) = f(y) \quad (\text{substitution axiom}) \tag{3.2}$$

for every function symbol. Analogously we require for all predicate symbols

$$\forall x\ \forall y\ x = y \Rightarrow p(x) \Leftrightarrow p(y) \quad (\text{substitution axiom}). \tag{3.3}$$

We formulate other mathematical relations, such as the "<" relation, by similar means (Exercise 3.4 on page 64).

Often a variable must be replaced by a term. To carry this out correctly and describe it simply, we give the following definition.

Definition 3.5 We write $\varphi[x/t]$ for the formula that results when we replace every free occurrence of the variable x in φ with the term t. Thereby we do not allow any variables in the term t that are quantified in φ. In those cases variables must be renamed to ensure this.

Example 3.3 If, in the formula $\forall x\ x = y$, the free variable y is replaced by the term $x + 1$, the result is $\forall x\ x = x + 1$. With correct substitution we obtain the formula $\forall x\ x = y + 1$, which has a very different semantic.

3.3 Quantifiers and Normal Forms

By Definition 3.4 on page 43, the formula $\forall x\ p(x)$ is true if and only if it is true for all interpretations of the variable x. Instead of the quantifier, one could write $p(a_1) \wedge \cdots \wedge p(a_n)$ for all constants $a_1 \cdots a_n$ in K. For $\exists x\ p(x)$ one could write $p(a_1) \vee \cdots \vee p(a_n)$. From this it follows with de Morgan's law that

$$\forall x\ \varphi \equiv \neg\exists x \neg\varphi.$$

Through this equivalence, universal, and existential quantifiers are mutually replaceable.

Example 3.4 The proposition "*Everyone wants to be loved*" is equivalent to the proposition "*Nobody does not want to be loved*".

Quantifiers are an important component of predicate logic's expressive power. However, they are disruptive for automatic inference in AI because they make the structure of formulas more complex and increase the number of applicable inference rules in every step of a proof. Therefore our next goal is to find, for every predicate logic formula, an equivalent formula in a standardized normal form with as few quantifiers as possible. As a first step we bring universal quantifiers to the beginning of the formula and thus define

Definition 3.6 A predicate logic formula φ is in *prenex normal form* if it holds that

- $\varphi = Q_1 x_1 \cdots Q_n x_n\, \psi$.
- ψ is a quantifierless formula.
- $Q_i \in \{\forall, \exists\}$ for $i = 1, \ldots, n$.

Caution is advised if a quantified variable appears outside the scope of its quantifier, as for example x in

$$\forall x\ p(x) \Rightarrow \exists x\ q(x).$$

Here one of the two variables must be renamed, and in

$$\forall x\ p(x) \Rightarrow \exists y\ q(y)$$

the quantifier can easily be brought to the front, and we obtain as output the equivalent formula

$$\forall x\ \exists y\ p(x) \Rightarrow q(y).$$

If, however, we wish to correctly bring the quantifier to the front of

$$(\forall x\ p(x)) \Rightarrow \exists y\ q(y) \tag{3.4}$$

we first write the formula in the equivalent form

$$\neg(\forall x\ p(x)) \vee \exists y\ q(y).$$

The first universal quantifier now turns into

$$(\exists x \neg p(x)) \vee \exists y\ q(y)$$

and now the two quantifiers can finally be pulled forward to

$$\exists x\ \exists y \neg p(x) \vee q(y),$$

which is equivalent to

$$\exists x\ \exists y p(x) \Rightarrow q(y).$$

We see then that in (3.4) on page 46 we cannot simply pull both quantifiers to the front. Rather, we must first eliminate the implications so that there are no negations on the quantifiers. It holds in general that we may only pull quantifiers out if negations only exist directly on atomic sub-formulas.

Example 3.5 As is well known in analysis, convergence of a series $(a_n)_{n\in\mathbb{N}}$ to a limit a is defined by

$$\forall \varepsilon > 0\ \exists n_0 \in \mathbb{N}\ \forall n > n_0 |a_n - a| < \varepsilon.$$

With the function $abs(x)$ for $|x|$, $a(n)$ for a_n, $minus(x, y)$ for x–y and the predicates $el(x, y)$ for $x \in y$, $gr(x, y)$ for $x > y$, the formula reads

$$\forall \varepsilon\,(gr(\varepsilon, 0) \Rightarrow \exists n_0\,(el(n_0, \mathbb{N}) \Rightarrow \forall n\,(gr(n, n_0) \Rightarrow gr(\varepsilon, abs(minus(a(n), a)))))). \tag{3.5}$$

This is clearly not in prenex normal form. Because the variables of the inner quantifiers $\exists n_0$ and $\forall n$ do not occur to the left of their respective quantifiers, no variables must be renamed. Next we eliminate the implications and obtain

$$\forall \varepsilon\ (\neg gr(\varepsilon, 0) \vee \exists n_0\ (\neg el(n_0, \mathbb{N}) \vee \forall n\ (\neg gr(n, n_0) \vee gr(\varepsilon, abs(minus(a(n), a)))))).$$

Because every negation is in front of an atomic formula, we bring the quantifiers forward, eliminate the redundant parentheses, and with

$$\forall \varepsilon\ \exists n_0\ \forall n\ (\neg gr(\varepsilon, 0) \vee \neg el(n_0, \mathbb{N}) \vee \neg gr(n, n_0) \vee gr(\varepsilon, abs(minus(a(n), a))))$$

it becomes a quantified clause in conjunctive normal form.

The transformed formula is equivalent to the output formula. The fact that this transformation is always possible is guaranteed by

Theorem 3.2 *Every predicate logic formula can be transformed into an equivalent formula in prenex normal form.*

In addition, we can eliminate all existential quantifiers. However, the formula resulting from the so-called *Skolemization* is no longer equivalent to the output formula. Its satisfiability, however, remains unchanged. In many cases, especially when one wants to show the unsatisfiability of $KB \wedge \neg Q$, this is sufficient. The following formula in prenex normal form will now be skolemized:

$$\forall x_1\ \forall x_2\ \exists y_1\ \forall x_3\ \exists y_2\ p(f(x_1), x_2, y_1) \vee q(y_1, x_3, y_2).$$

Because the variable y_1 apparently depends on x_1 and x_2, every occurrence of y_1 is replaced by a Skolem function $g(x_1, x_2)$. It is important that g is a new function symbol that has not yet appeared in the formula. We obtain

$$\forall x_1\ \forall x_2\ \forall x_3\ \exists y_2\ p(f(x_1), x_2, g(x_1, x_2)) \vee q(g(x_1, x_2), x_3, y_2)$$

and replace y_2 analogously by $h(x_1, x_2, x_3)$, which leads to

$$\forall x_1\ \forall x_2\ \forall x_3\ p(f(x_1), x_2, g(x_1, x_2)) \vee q(g(x_1, x_2), x_3, h(x_1, x_2, x_3)).$$

Because now all the variables are universally quantified, the universal quantifiers can be left out, resulting in

$$p(f(x_1), x_2, g(x_1, x_2)) \vee q(g(x_1, x_2), x_3, h(x_1, x_2, x_3)).$$

Now we can eliminate the existential quantifier (and thereby also the universal quantifier) in (3.5) on page 47 by introducing the Skolem function $n_0(\varepsilon)$. The skolemized prenex and conjunctive normal form of (3.5) on page 47 thus reads

$$\neg gr(\varepsilon, 0) \vee \neg el(n_0(\varepsilon), \mathbb{N}) \vee \neg gr(n, n_0(\varepsilon)) \vee gr(\varepsilon, abs(minus(a(n), a))).$$

By dropping the variable n_0, the Skolem function can receive the name n_0.

When skolemizing a formula in prenex normal form, all existential quantifiers are eliminated from the outside inward, where a formula of the form $\forall x_1 \ldots \forall x_n\ \exists y\ \varphi$ is replaced by $\forall x_1 \ldots \forall x_n\ \varphi[y/f(x_1, \ldots, x_n)]$, during which f may not appear in φ. If an existential quantifier is on the far outside, such as in $\exists y\ p(y)$, then y must be replaced by a constant (that is, by a zero-place function symbol).

NORMALFORMTRANSFORMATION(*Formula*):
1. *Transformation into prenex normal form*:
   Transformation into conjunctive normal form (Theorem 2.1):
      Elimination of equivalences.
      Elimination of implications.
      Repeated application of de Morgan's law and distributive law.
   Renaming of variables if necessary.
   Factoring out universal quantifiers.
2. *Skolemization*:
   Replacement of existentially quantified variables by new Skolem functions.
   Deletion of resulting universal quantifiers.

Fig. 3.2 Transformation of predicate logic formulas into normal form

The procedure for transforming a formula in conjunctive normal form is summarized in the pseudocode represented in Fig. 3.2. Skolemization has polynomial runtime in the number of literals. When transforming into normal form, the number of literals in the normal form can grow exponentially, which can lead to exponential computation time and exponential memory usage. The reason for this is the repeated application of the distributive law. The actual problem, which results from a large number of clauses, is the combinatorial explosion of the search space for a subsequent resolution proof. However, there is an optimized transformation algorithm which only spawns polynomially many literals [Ede91].

3.4 Proof Calculi

For reasoning in predicate logic, various calculi of natural reasoning such as Gentzen calculus or sequent calculus, have been developed. As the name suggests, these calculi are meant to be applied by humans, since the inference rules are more or less intuitive and the calculi work on arbitrary PL1 formulas. In the next section we will primarily concentrate on the resolution calculus, which is in practice the most important efficient, automatizable calculus for formulas in conjunctive normal form. Here, using Example 3.2 on page 43 we will give a very small "natural" proof. We use the inference rule

$$\frac{A,\quad A \Rightarrow B}{B}\quad \text{(modus ponens, MP)} \quad \text{and} \quad \frac{\forall x\, A}{A[x/t]}\quad (\forall\text{-elimination}, \forall E).$$

The modus ponens is already familiar from propositional logic. When eliminating universal quantifiers one must keep in mind that the quantified variable x must be

Table 3.2 Simple proof with modus ponens and quantifier elimination

WB:	1	$child(eve, anne, oscar)$
WB:	2	$\forall x\ \forall y\ \forall z\ child(x, y, z) \Rightarrow child(x, z, y)$
$\forall E(2)$: *x*/*eve*, *y*/*anne*, *z*/*oscar*	3	$child(eve, anne, oscar) \Rightarrow child(eve, oscar, anne)$
MP(1, 3)	4	$child(eve, oscar, anne)$

replaced by a ground term *t*, meaning a term that contains no variables. The proof of *child*(*eve*, *oscar*, *anne*) from an appropriately reduced knowledge base is presented in Table 3.2.

The two formulas of the reduced knowledge base are listed in rows 1 and 2. In row 3 the universal quantifiers from row 2 are eliminated, and in row 4 the claim is derived with modus ponens.

The calculus consisting of the two given inference rules is not complete. However, it can be extended into a complete procedure by addition of further inference rules. This nontrivial fact is of fundamental importance for mathematics and AI. The Austrian logician Kurt Gödel proved in 1931 that [Göd31a].

Theorem 3.3 (Gödel's completeness theorem) *First-order predicate logic is complete. That is, there is a calculus with which every proposition that is a consequence of a knowledge base KB can be proved. If* $KB \models \varphi$, *then it holds that* $KB \vdash \varphi$.

Every true proposition in first-order predicate logic is therefore provable. But is the reverse also true? Is everything we can derive syntactically actually true? The answer is "yes":

Theorem 3.4 (Correctness) *There are calculi with which only true propositions can be proved. That is, if* $KB \vdash \varphi$ *holds, then* $KB \models \varphi$.

In fact, nearly all known calculi are correct. After all, it makes little sense to work with incorrect proof methods. Provability and semantic consequence are therefore equivalent concepts, as long as correct and complete calculus is being used. Thereby first-order predicate logic becomes a powerful tool for mathematics and AI. The aforementioned calculi of natural deduction are rather unsuited for automatization. Only resolution calculus, which was introduced in 1965 and essentially works with only one simple inference rule, enabled the construction of powerful automated theorem provers, which later were employed as inference machines for expert systems.

3.5 Resolution

Indeed, the correct and complete resolution calculus triggered a logic euphoria during the 1970s. Many scientists believed that one could formulate almost every task of knowledge representation and reasoning in PL1 and then solve it with an automated prover. Predicate logic, a powerful, expressive language, together with a complete proof calculus seemed to be the universal intelligent machine for representing knowledge and solving many difficult problems (Fig. 3.3).

If one feeds a set of axioms (that is, a knowledge base) and a query into such a logic machine as input, the machine searches for a proof and returns it—for one exists and will be found—as output. With Gödel's completeness theorem and the work of Herbrand as a foundation, much was invested into the mechanization of logic. The vision of a machine that could, with an arbitrary non-contradictory PL1 knowledge base, prove any true query was very enticing. Accordingly, until now many proof calculi for PL1 are being developed and realized in the form of theorem provers. As an example, here we describe the historically important and widely used resolution calculus and show its capabilities. The reason for selecting resolution as an example of a proof calculus in this book is, as stated, its historical and

Fig. 3.3 The universal logic machine

didactic importance. Today, resolution represents just one of many calculi used in high-performance provers.

We begin by trying to compile the proof in Table 3.2 on page 50 with the knowledge base of Example 3.2 on page 43 into a resolution proof. First the formulas are transformed into conjunctive normal form and the negated query

$$\neg Q \equiv \neg child(eve, oscar, anne)$$

is added to the knowledge base, which gives

$$\begin{aligned} KB \wedge \neg Q \equiv & (child(eve,\ anne,\ oscar))_1 \wedge \\ & (\neg child(x,\ y,\ z) \vee child(x,\ z,\ y))_2 \wedge \\ & (\neg child(eve,\ oscar,\ anne))_3. \end{aligned}$$

The proof could then look something like

$$\begin{aligned} (2)\ x/eve,\ y/anne,\ z/oscar: & \ (\neg child(eve, anne, oscar) \vee \\ & \quad child(eve, oscar, anne))_4 \\ Res(3,4): & \ (\neg child(eve, anne, oscar))_5 \\ Res(1,5): & \ ()_6, \end{aligned}$$

where, in the first step, the variables x, y, z are replaced by constants. Then two resolution steps follow under application of the general resolution rule from (2.2), which was taken unchanged from propositional logic.

The circumstances in the following example are somewhat more complex. We assume that *everyone knows his own mother* and ask whether *Henry* knows anyone. With the function symbol "*mother*" and the predicate "*knows*", we have to derive a contradiction from

$$(knows(x, mother(x)))_1 \wedge (\neg knows(henry, y))_2.$$

By the replacement *x*/*henry*, *y*/*mother*(*henry*) we obtain the contradictory clause pair

$$(knows(henry, mother(henry)))_1 \wedge (\neg knows(henry, mother(henry)))_2.$$

This replacement step is called *unification*. The two literals are complementary, which means that they are the same other than their signs. The empty clause is now derivable with a resolution step, by which it has been shown that Henry does know someone (his mother). We define

Definition 3.7 Two literals are called *unifiable* if there is a substitution σ for all variables which makes the literals equal. Such a σ is called a *unifier*. A unifier is called the most general unifier (MGU) if all other unifiers can be obtained from it by substitution of variables.

Example 3.6 We want to unify the literals $p(f(g(x)), y, z)$ and $p(u, u, f(u))$. Several unifiers are

$$\begin{array}{lllll}
\sigma_1: & & y/f(g(x)), & z/f(f(g(x))), & u/f(g(x)), \\
\sigma_2: & x/h(v), & y/f(g(h(v))), & z/f(f(g(h(v)))), & u/f(g(h(v))) \\
\sigma_3: & x/h(h(v)), & y/f(g(h(h(v)))), & z/f(f(g(h(h(v))))), & u/f(g(h(h(v)))) \\
\sigma_4: & x/h(a), & y/f(g(h(a))), & z/f(f(g(h(a)))), & u/f(g(h(a))) \\
\sigma_5: & x/a, & y/f(g(a)), & z/f(f(g(a))), & u/f(g(a))
\end{array}$$

where σ_1 is the most general unifier. The other unifiers result from σ_1 through the substitutions $x/h(v)$, $x/h(h(v))$, $x/h(a)$, x/a.

We can see in this example that during unification of literals, the predicate symbols can be treated like function symbols. That is, the literal is treated like a term. Implementations of unification algorithms process the arguments of functions sequentially. Terms are unified recursively over the term structure. The simplest unification algorithms are very fast in most cases. In the worst case, however, the computation time can grow exponentially with the size of the terms. Because for automated provers the overwhelming number of unification attempts fail or are very simple, in most cases the worst case complexity has no dramatic effect. The fastest unification algorithms have nearly linear complexity even in the worst case [Bib82].

We can now give the general resolution rule for predicate logic:

Definition 3.8 The resolution rule for two clauses in conjunctive normal form reads

$$\frac{(A_1 \vee \cdots \vee A_m \vee B), \quad (\neg B' \vee C_1 \vee \cdots \vee C_n) \quad \sigma(B) = \sigma(B')}{(\sigma(A_1) \vee \cdots \vee \sigma(A_m) \vee \sigma(C_1) \vee \cdots \vee \sigma(C_n))}, \tag{3.6}$$

where σ is the MGU of B and B'.

Theorem 3.5 *The resolution rule is correct. That is, the resolvent is a semantic consequence of the two parent clauses.*

For Completeness, however, we still need a small addition, as is shown in the following example.

Example 3.7 The famous Russell paradox reads "*There is a barber who shaves everyone who does not shave himself.*" This statement is contradictory, meaning it is unsatisfiable. We wish to show this with resolution. Formalized in PL1, the paradox reads

$$\forall x \; shaves(barber, x) \;\Leftrightarrow\; \neg shaves(x, x)$$

and transformation into clause form yields (see Exercise 3.6 on page 64)

$$(\neg shaves(barbier,\, x) \;\vee\; \neg shaves(x,\, x))_1 \;\wedge\; (shaves(barbier,\, x) \;\vee\; shaves(x,\, x))_2. \tag{3.7}$$

From these two clauses we can derive several tautologies, but no contradiction. Thus resolution is not complete. We need yet a further inference rule.

Definition 3.9 *Factorization* of a clause is accomplished by

$$\frac{(A_1 \vee A_2 \vee \cdots \vee A_n) \quad \sigma(A_1) = \sigma(A_2)}{(\sigma(A_2) \vee \cdots \vee \sigma(A_n))},$$

where σ is the MGU of A_1 and A_2.

Now a contradiction can be derived from (3.7)

$$\begin{aligned} \text{Fak}(1, \sigma : x/barber): &\quad (\neg shaves(barber, barber))_3 \\ \text{Fak}(2, \sigma : x/barber): &\quad (shaves(barber, barber))_4 \\ \text{Res}(3,\, 4): &\quad ()_5 \end{aligned}$$

and we assert:

Theorem 3.6 *The resolution rule* (3.6) *together with the factorization rule* (3.9) *is refutation complete. That is, by application of factorization and resolution steps, the empty clause can be derived from any unsatisfiable formula in conjunctive normal form.*

3.5.1 Resolution Strategies

While completeness of resolution is important for the user, the search for a proof can be very frustrating in practice. The reason for this is the immense combinatorial search space. Even if there are only very few pairs of clauses in $KB \wedge \neg Q$ in the beginning, the prover generates a new clause with every resolution step, which increases the number of possible resolution steps in the next iteration. Thus it has long been attempted to reduce the search space using special strategies, preferably without losing completeness. The most important strategies are the following.

Unit resolution prioritizes resolution steps in which one of the two clauses consists of only one literal, called a unit clause. This strategy preserves completeness and leads in many cases, but not always, to a reduction of the search space. It therefore is a heuristic process (see Sect. 6.3).

One obtains a guaranteed reduction of the search space by application of the *set of support strategy*. Here a subset of $KB \wedge \neg Q$ is defined as the set of support (SOS). Every resolution step must involve a clause from the SOS, and the resolvent is added to the SOS. This strategy is incomplete. It becomes complete when it is ensured that the set of clauses is satisfiable without the SOS (see Exercise 3.7 on page 64). The negated query $\neg Q$ is often used as the initial SOS.

In *input resolution*, a clause from the input set $KB \wedge \neg Q$ must be involved in every resolution step. This strategy also reduces the search space, but at the cost of completeness.

With the *pure literal rule* all clauses that contain literals for which there are no complementary literals in other clauses can be deleted. This rule reduces the search space and is complete, and therefore it is used by practically all resolution provers.

If the literals of a clause K_1 represent a subset of the literals of the clause K_2, then K_2 can be deleted. For example, the clause

$$(raining(today) \Rightarrow street\_wet(today))$$

is redundant if *street_wet*(*today*) is already valid. This important reduction step is called *subsumption*. Subsumption, too, is complete.

3.5.2 Equality

Equality is an especially inconvenient cause of explosive growth of the search space. If we add (3.1) on page 45 and the equality axioms formulated in (3.2) on page 45 to the knowledge base, then the symmetry clause $\neg x = y \vee y = x$ can be unified with every positive or negated equation, for example. This leads to the derivation of new clauses and equations upon which equality axioms can again be applied, and so on. The transitivity and substitution axioms have similar consequences. Because of this, special inference rules for equality have been developed which get by without explicit equality axioms and, in particular, reduce the search

space. *Demodulation*, for example, allows substitution of a term t_2 for t_1, if the equation $t_1 = t_2$ exists. An equation $t_1 = t_2$ is applied by means of unification to a term t as follows:

$$\frac{t_1 = t_2, \quad (\ldots t \ldots), \sigma(t_1) = \sigma(t)}{(\ldots \sigma(t_2) \ldots)}.$$

Somewhat more general is *paramodulation*, which works with conditional equations [Bib82, Lov78].

The equation $t_1 = t_2$ allows the substitution of the term t_1 by t_2 as well as the substitution t_2 by t_1. It is usually pointless to reverse a substitution that has already been carried out. On the contrary, equations are frequently used to simplify terms. They are thus often used in one direction only. Equations which are only used in one direction are called directed equations. Efficient processing of directed equations is accomplished by so-called *term rewriting systems*. For formulas with many equations there exist special equality provers.

3.6 Automated Theorem Provers

Implementations of proof calculi on computers are called theorem provers. Along with specialized provers for subsets of PL1 or special applications, there exist today a whole line of automated provers for the full predicate logic and higher-order logics, of which only a few will be discussed here. An overview of the most important systems can be found in [McC].

One of the oldest resolution provers was developed at the Argonne National Laboratory in Chicago. Based on early developments starting in 1963, Otter [Kal01], was created in 1984. Above all, Otter was successfully applied in specialized areas of mathematics, as one can learn from its home page:

> "Currently, the main application of Otter is research in abstract algebra and formal logic. Otter and its predecessors have been used to answer many open questions in the areas of finite semigroups, ternary Boolean algebra, logic calculi, combinatory logic, group theory, lattice theory, and algebraic geometry."

Several years later the University of Technology, Munich, created the high-performance prover SETHEO [LSBB92] based on fast PROLOG technology. With the goal of reaching even higher performance, an implementation for parallel computers was developed under the name PARTHEO. It turned out that it was not worthwhile to use special hardware in theorem provers, as is also the case in other areas of AI, because these computers are very quickly overtaken by faster processors and more intelligent algorithms. Munich is also the birthplace of E [Sch02], an award-winning modern equation prover, which we will become familiar with in the next example. On E's homepage one can read the following compact, ironic characterization, whose second part incidentally applies to all automated provers in existence today.

> "E is a purely equational theorem prover for clausal logic. That means it is a program that you can stuff a mathematical specification (in clausal logic with equality) and a hypothesis into, and which will then run forever, using up all of your machines resources. Very occasionally it will find a proof for the hypothesis and tell you so ;-)."

Finding proofs for true propositions is apparently so difficult that the search succeeds only extremely rarely, or only after a very long time—if at all. We will go into this in more detail in Chap. 4. Here it should be mentioned, though, that not only computers, but also most people have trouble finding strict formal proofs.

Though evidently computers by themselves are in many cases incapable of finding a proof, the next best thing is to build systems that work semi-automatically and allow close cooperation with the user. Thereby the human can better apply his knowledge of special application domains and perhaps limit the search for the proof. One of the most successful interactive provers for higher-order predicate logic is Isabelle [NPW02], a common product of Cambridge University and the University of Technology, Munich.

Anyone searching for a high-performance prover should look at the current results of the CASC (CADE ATP System Competition) [SS06].[1] Here we find that the winner from 2001 to 2006 in the PL1 and clause normal form categories was Manchester's prover Vampire, which works with a resolution variant and a special approach to equality. The system Waldmeister of the Max Planck Institute in Saarbrücken has been leading for years in equality proving.

The many top positions of German systems at CASC show that German research groups in the area of automated theorem proving are playing a leading role, today as well as in the past.

3.7 Mathematical Examples

We now wish to demonstrate the application of an automated prover with the aforementioned prover E [Sch02]. E is a specialized equality prover which greatly shrinks the search space through an optimized treatment of equality.

We want to prove that left- and right-neutral elements in a semigroup are equal. First we formalize the claim step by step.

Definition 3.10 A structure $(M,\cdot)$ consisting of a set M with a two-place inner operation "$\cdot$" is called a semigroup if the law of associativity

$$\forall x\ \forall y\ \forall z\ (x \cdot y) \cdot z = x \cdot (y \cdot z)$$

holds. An element $e \in M$ is called left-neutral (right-neutral) if $\forall x\ e \cdot x = x$ $(\forall x\ x \cdot e = x)$.

[1] CADE is the annual "Conference on Automated Deduction" [CAD] and ATP stands for "Automated Theorem Prover".

It remains to be shown that

Theorem 3.7 *If a semigroup has a left-neutral element e_l and a right-neutral element e_r, then $e_l = e_r$.*

First we prove the theorem semi-formally by intuitive mathematical reasoning. Clearly it holds for all $x \in M$ that

$$e_l \cdot x = x \tag{3.8}$$

and

$$x \cdot e_r = x. \tag{3.9}$$

If we set $x = e_r$ in (3.8) and $x = e_l$ in (3.9), we obtain the two equations $e_l \cdot e_r = e_r$ and $e_l \cdot e_r = e_l$. Joining these two equations yields

$$e_l = e_l \cdot e_r = e_r,$$

which we want to prove. In the last step, incidentally, we used the fact that equality is symmetric and transitive.

Before we apply the automated prover, we carry out the resolution proof manually. First we formalize the negated query and the knowledge base *KB*, consisting of the axioms as clauses in conjunctive normal form:

$(\neg e_l = e_r)_1$	negated query
$(m(m(x,y),z) = m(x,m(y,z)))_2$	
$(m(e_l,x) = x)_3$	
$(m(x,e_r) = x)_4$	
equality axioms:	
$(x = x)_5$	(reflexivity)
$(\neg x = y \vee y = x)_6$	(symmetry)
$(\neg x = y \vee \neg y = z \vee x = z)_7$	(transitivity)
$(\neg x = y \vee m(x,z) = m(y,z))_8$	substitution in m
$(\neg x = y \vee m(z,x) = m(z,y))_9$	substitution in m,

where multiplication is represented by the two-place function symbol m. The equality axioms were formulated analogously to (3.1) on page 45 and (3.2) on page 45. A simple resolution proof has the form

$$
\begin{array}{ll}
\mathrm{Res}(3,6,x_6/m(e_l,x_3),y_6/x_3): & (x = m(e_l,x))_{10} \\
\mathrm{Res}(7,10,x_7/x_{10},y_7/m(e_l,x_{10})): & (\neg m(e_l,x) = z \vee x = z)_{11} \\
\mathrm{Res}(4,11,x_4/e_l,x_{11}/e_r,z_{11}/e_l): & (e_r = e_l)_{12} \\
\mathrm{Res}(1,12,\emptyset): & ().
\end{array}
$$

Here, for example, $\mathrm{Res}(3,6,x_6/m(e_l,x_3),y_6/x_3)$ means that in the resolution of clause 3 with clause 6, the variable x from clause 6 is replaced by $m(e_l,x_3)$ with variable x from clause 3. Analogously, y from clause 6 is replaced by x from clause 3.

Now we want to apply the prover E to the problem. The clauses are transformed into the clause normal form language LOP through the mapping

$$(\neg A_1 \vee \cdots \vee \neg A_m \vee B_1 \vee \cdots \vee Bn) \quad \mapsto \quad \mathtt{B_1;\ldots;B_n} < -\mathtt{A_1},\ldots,\mathtt{A_m}.$$

The syntax of LOP represents an extension of the PROLOG syntax (see Chap. 5) for non Horn clauses. Thus we obtain as an input file for E

```
<- eq(el,er).                        # query
eq( m(m(X,Y),Z), m(X,m(Y,Z)) ).      # associativity of m
eq( m(el,X), X ).                    # left-neutral element of m
eq( m(X,er), X ).                    # right-neutral element of m
eq(X,X).                             # equality: reflexivity
eq(Y,X) <- eq(X,Y).                  # equality: symmetry
eq(X,Z) <- eq(X,Y), eq(Y,Z).         # equality: transitivity
eq( m(X,Z), m(Y,Z) ) <- eq(X,Y).     # equality: substitution in m
eq( m(Z,X), m(Z,Y) ) <- eq(X,Y).     # equality: substitution in m
```

where equality is modeled by the predicate symbol `eq`. Calling the prover delivers

```
unixprompt> eproof halbgr1.lop
# Problem status determined, constructing proof object
# Evidence for problem status starts
     0 : [--eq(el,er)] : initial
     1 : [++eq(X1,X2),--eq(X2,X1)] : initial
     2 : [++eq(m(el,X1),X1)] : initial
     3 : [++eq(m(X1,er),X1)] : initial
     4 : [++eq(X1,X2),--eq(X1,X3),--eq(X3,X2)] : initial
     5 : [++eq(X1,m(X1,er))] : pm(3,1)
     6 : [++eq(X2,X1),--eq(X2,m(el,X1))] : pm(2,4)
     7 : [++eq(el,er)] : pm(5,6)
     8 : [] : sr(7,0)
     9 : [] : 8 : {proof}
# Evidence for problem status ends
```

Positive literals are identified by ++ and negative literals by −−. In lines 0 to 4, marked with `initial`, the clauses from the input data are listed again. `pm(a, b)` stands for a resolution step between clause a and clause b. We see that the proof found by E is very similar to the manually created proof. Because we explicitly model the equality by the predicate `eq`, the particular strengths of E do not come into play. Now we omit the equality axioms and obtain

```
<- el = er.                        % query
m(m(X,Y),Z) = m(X,m(Y,Z)) .        % associativity of m
m(el,X) = X .                      % left-neutral element of m
m(X,er) = X .                      % right-neutral element of m
```

as input for the prover.

The proof also becomes more compact. We see in the following output of the prover that the proof consists essentially of a single inference step on the two relevant clauses 1 and 2.

```
unixprompt> eproof halbgr1a.lop
# Problem status determined, constructing proof object
# Evidence for problem status starts
     0 : [--equal(el, er)] : initial
     1 : [++equal(m(el,X1), X1)] : initial
     2 : [++equal(m(X1,er), X1)] : initial
     3 : [++equal(el, er)] : pm(2,1)
     4 : [--equal(el, el)] : rw(0,3)
     5 : [] : cn(4)
     6 : [] : 5 : {proof}
# Evidence for problem status ends
```

The reader might now take a closer look at the capabilities of E (Exercise 3.9 on page 64).

3.8 Applications

In mathematics automated theorem provers are used for certain specialized tasks. For example, the important four color theorem of graph theory was first proved in 1976 with the help of a special prover. However, automated provers still play a minor role in mathematics.

On the other hand, in the beginning of AI, predicate logic was of great importance for the development of expert systems in practical applications. Due to its problems modeling uncertainty (see Sect. 4.4), expert systems today are most often developed using other formalisms.

Today logic plays an ever more important role in verification tasks. Automatic program verification is currently an important research area between AI and software engineering. Increasingly complex software systems are now taking over tasks of more and more responsibility and security relevance. Here a proof of certain safety characteristics of a program is desirable. Such a proof cannot be brought about through testing of a finished program, for in general it is impossible to apply a program to all possible inputs. This is therefore an ideal domain for general or even specialized inference systems. Among other things, cryptographic protocols are in use today whose security characteristics have been automatically verified [FS97, Sch01]. A further challenge for the use of automated provers is the synthesis of software and hardware. To this end, for example, provers should support the software engineer in the generation of programs from specifications.

Software reuse is also of great importance for many programmers today. The programmer looks for a program that takes input data with certain properties and calculates a result with desired properties. A sorting algorithm accepts input data with entries of a certain data type and from these creates a permutation of these entries with the property that every element is less than or equal to the next element. The programmer first formulates a specification of the query in PL1 consisting of two parts. The first part PRE_Q comprises the preconditions, which must hold before the desired program is applied. The second part $POST_Q$ contains the postconditions, which must hold after the desired program is applied.

In the next step a software database must be searched for modules which fulfill these requirements. To check this formally, the database must store a formal description of the preconditions PRE_M and postconditions $POST_M$ for every module M. An assumption about the capabilities of the modules is that the preconditions of the module follow from the preconditions of the query. It must hold that

$$PRE_Q \Rightarrow PRE_M.$$

All conditions that are required as a prerequisite for the application of module M must appear as preconditions in the query. If, for example, a module in the database only accepts lists of integers, then lists of integers as input must also appear as preconditions in the query. An additional requirement in the query that, for example, only even numbers appear, does not cause a problem.

Furthermore, it must hold for the postconditions that

$$POST_M \Rightarrow POST_Q.$$

That is, after application of the module, all attributes that the query requires must be fulfilled. We now show the application of a theorem prover to this task in an example from [Sch01].

Example 3.8 VDM-SL, the Vienna Development Method Specification Language, is often used as a language for the specification of pre- and postconditions. Assume that in the software database the description of a module ROTATE is available, which moves the first list element to the end of the list. We are looking for a module SHUFFLE, which creates an arbitrary permutation of the list. The two specifications read

ROTATE($l : List$) $l' : List$
pre *true*
post
$(l = [] \Rightarrow l' = []) \wedge$
$(l \neq [] \Rightarrow l' = (\text{tail}\, l)\hat{}[\text{head}\, l])$

SHUFFLE($x : List$) $x' : List$
pre *true*
post $\forall i : Item \cdot$
$(\exists x_1, x_2 : List \cdot x = x_1\hat{}[i]\hat{}x_2 \Leftrightarrow$
$\exists y_1, y_2 : List \cdot x' = y_1\hat{}[i]\hat{}y_2)$

Here "$\hat{}$" stands for the concatenation of lists, and "$\cdot$" separates quantifiers with their variables from the rest of the formula. The functions "head l" and "tail l" choose the first element and the rest from the list, respectively. The specification of SHUFFLE indicates that every list element i that was in the list (x) before the application of SHUFFLE must be in the result (x') after the application, and vice versa. It must now be shown that the formula $(PRE_Q \Rightarrow PRE_M) \wedge (POST_M \Rightarrow POST_Q)$ is a consequence of the knowledge base containing a description of the data type *List*. The two VDM-SL specifications yield the proof task

$$\begin{aligned}
&\forall l, l', x, x' : List \cdot (l = x \wedge l' = x' \wedge (w \Rightarrow w)) \wedge \\
&\quad (l = x \wedge l' = x' \wedge ((l = [] \Rightarrow l' = []) \wedge (l \neq [] \Rightarrow l' = (tl\, l)\hat{}[hd\, l]) \\
&\quad \Rightarrow \forall i : Item \cdot (\exists x_1, x_2 : List \cdot x = x_1\hat{}[i]\hat{}x_2 \Leftrightarrow \exists y_1, y_2 : List \cdot x' = y_1\hat{}[i]\hat{}y_2))),
\end{aligned}$$

which can then be proven with the prover SETHEO.

In the coming years the *semantic web* will likely represent an important application of PL1. The content of the World Wide Web is supposed to become interpretable not only for people, but for machines. To this end web sites are being furnished with a description of their semantics in a formal description language. The search for information in the web will thereby become significantly more effective than today, where essentially only text building blocks are searchable.

Decidable subsets of predicate logic are used as description languages. The development of efficient calculi for reasoning is very important and closely connected to the description languages. A query for a future semantically operating search engine could (informally) read: Where in Switzerland next Sunday at elevations under 2000 meters will there be good weather and optimally prepared ski slopes? To answer such a question, a calculus is required that is capable of working very quickly on large sets of facts and rules. Here, complex nested function terms are less important.

As a basic description framework, the World Wide Web Consortium developed the language RDF (Resource Description Framework). Building on RDF, the significantly more powerful language OWL (Web Ontology Language) allows the description of relations between objects and classes of objects, similarly to PL1 [SET09]. *Ontologies* are descriptions of relationships between possible objects.

A difficulty when building a description of the innumerable websites is the expenditure of work and also checking the correctness of the semantic descriptions. Here machine learning systems for the automatic generation of descriptions can be very helpful. An interesting use of "automatic" generation of semantics in the web was introduced by Luis von Ahn of Carnegie Mellon University [vA06]. He developed computer games in which the players, distributed over the network, are supposed to collaboratively describe pictures with key words. Thus the pictures are assigned semantics in a fun way at no cost.

3.9 Summary

We have provided the most important foundations, terms, and procedures of predicate logic and we have shown that even one of the most difficult intellectual tasks, namely the proof of mathematical theorems, can be automated. Automated provers can be employed not only in mathematics, but rather, in particular, in verification tasks in computer science. For everyday reasoning, however, predicate logic in most cases is ill-suited. In the next and the following chapters we show its weak points and some interesting modern alternatives. Furthermore, we will show in Chap. 5 that one can program elegantly with logic and its procedural extensions.

Anyone interested in first-order logic, resolution and other calculi for automated provers will find good advanced instruction in [New00, Fit96, Bib82, Lov78, CL73]. References to Internet resources can be found on this book's web site.

3.10 Exercises

Exercise 3.1 Let the three-place predicate "child" and the one-place predicate "female" from Example 3.2 on page 43 be given. Define:

(a) A one-place predicate "male".
(b) A two-place predicate "father" and "mother".
(c) A two-place predicate "siblings".
(d) A predicate "parents (x, y, z)", which is true if and only if x is the father and y is the mother of z.
(e) A predicate "uncle (x, y)", which is true if and only if x is the uncle of y (use the predicates that have already been defined).
(f) A two-place predicate "ancestor" with the meaning: *ancestors are parents, grandparents, etc. of arbitrarily many generations*.

Exercise 3.2 Formalize the following statements in predicate logic:
(a) Every person has a father and a mother.
(b) Some people have children.
(c) All birds fly.
(d) There is an animal that eats (some) grain-eating animals.
(e) Every animal eats plants or plant-eating animals which are much smaller than itself.

Exercise 3.3 Adapt Exercise 3.1 on page 63 by using one-place function symbols and equality instead of "father" and "mother".

Exercise 3.4 Give predicate logic axioms for the two-place relation "<" as a total order. For a total order we must have (1) Any two elements are comparable. (2) It is symmetric. (3) It is transitive.

Exercise 3.5 Unify (if possible) the following terms and give the MGU and the resulting terms.
(a) $p(x, f(y)), p(f(z), u)$
(b) $p(x, f(x)), p(y, y)$
(c) $x = 4 - 7 \cdot x, \cos y = z$
(d) $x < 2 \cdot x, 3 < 6$
(e) $q(f(x, y, z), f(g(w, w), g(x, x), g(y, y))), q(u, u)$

Exercise 3.6
(a) Transform Russell's Paradox from Example 3.7 on page 54 into CNF.
(b) Show that the empty clause cannot be derived using resolution without factorization from (3.7) on page 54. Try to understand this intuitively.

Exercise 3.7
(a) Why is resolution with the set of support strategy incomplete?
(b) Justify (without proving) why the set of support strategy becomes complete if $(KB \wedge \neg Q)$\SOS is satisfiable.
(c) Why is resolution with the pure literal rule complete?

❄ **Exercise 3.8** Formalize and prove with resolution that in a semigroup with at least two different elements a, b, a left-neutral element e, and a left null element n, these two elements have to be different, that is, that $n \neq e$. Use demodulation, which allows replacement of "like with like".

Exercise 3.9 Obtain the theorem prover E [Sch02] or another prover and prove the following statements. Compare these proofs with those in the text.
(a) The claim from Example 2.3 on page 33.
(b) Russell's paradox from Example 3.7 on page 54.
(c) The claim from Exercise 3.8.