Handbook of Knowledge Representation
Edited by F. van Harmelen, V. Lifschitz and B. Porter

DOI: 10.1016/S1574-6526(07)03001-5

Chapter 1

Knowledge Representation and Classical Logic

Vladimir Lifschitz, Leora Morgenstern, David Plaisted

1.1 Knowledge Representation and Classical Logic

Mathematical logicians had developed the art of formalizing declarative knowledge long before the advent of the computer age. But they were interested primarily in formalizing mathematics. Because of the important role of nonmathematical knowledge in AI, their emphasis was too narrow from the perspective of knowledge representation, their formal languages were not sufficiently expressive. On the other hand, most logicians were not concerned about the possibility of automated reasoning; from the perspective of knowledge representation, they were often too generous in the choice of syntactic constructs. In spite of these differences, classical mathematical logic has exerted significant influence on knowledge representation research, and it is appropriate to begin this Handbook with a discussion of the relationship between these fields.

The language of classical logic that is most widely used in the theory of knowledge representation is the language of first-order (predicate) formulas. These are the formulas that John McCarthy proposed to use for representing declarative knowledge in his Advice Taker paper [171], and Alan Robinson proposed to prove automatically using resolution [230]. Propositional logic is, of course, the most important subset of first-order logic; recent surge of interest in representing knowledge by propositional formulas is related to the creation of fast satisfiability solvers for propositional logic (see Chapter 2). At the other end of the spectrum we find higher-order languages of classical logic. Second-order formulas are particularly important for the theory of knowledge representation, among other reasons, because they are sufficiently expressive for defining transitive closure and related concepts, and because they are used in the definition of circumscription (see Section 6.4).

Now a few words about the logical languages that are *not* considered "classical". Formulas containing modal operators, such as operators representing knowledge and belief (Chapter 15), are not classical. Languages with a classical syntax but a nonclas-

sical semantics, such as intuitionistic logic and the superintuitionistic logic of strong equivalence (see Section 7.3.3), are not discussed in this chapter either. Nonmonotonic logics (Chapters 6 and 19) are nonclassical as well.

This chapter contains an introduction to the syntax and semantics of classical logic and to natural deduction; a survey of automated theorem proving; a concise overview of selected implementations and applications of theorem proving; and a brief discussion of the suitability of classical logic for knowledge representation, a debate as old as the field itself.

1.2 Syntax, Semantics and Natural Deduction

Early versions of modern logical notation were introduced at the end of the 19th century in two short books. One was written by Gottlob Frege [89]; his intention was "to express a content through written signs in a more precise and clear way than it is possible to do through words" [261, p. 2]. The second, by Giuseppe Peano [204], introduces notation in which "every proposition assumes the form and the precision that equations have in algebra" [261, p. 85]. Two other logicians who have contributed to the creation of first-order logic are Charles Sanders Peirce and Alfred Tarski.

The description of the syntax of logical formulas in this section is rather brief. A more detailed discussion of syntactic questions can be found in Chapter 2 of the *Handbook of Logic in Artificial Intelligence and Logic Programming* [68], or in introductory sections of any logic textbook.

1.2.1 Propositional Logic

Propositional logic was carved out of a more expressive formal language by Emil Post [216].

Syntax and semantics

A *propositional signature* is a nonempty set of symbols called *atoms*. (Some authors say "vocabulary" instead of "signature", and "variable" instead of "atom".) *Formulas* of a propositional signature σ are formed from atoms and the 0-place connectives $\bot$ and $\top$ using the unary connective $\neg$ and the binary connectives $\wedge$, $\vee$, $\rightarrow$ and $\leftrightarrow$. (Some authors write & for $\wedge$, $\supset$ for $\rightarrow$, and $\equiv$ for $\leftrightarrow$.)[1]

The symbols FALSE and TRUE are called *truth values*. An *interpretation* of a propositional signature σ (or an *assignment*) is a function from σ into {FALSE, TRUE}. The semantics of propositional formulas defines which truth value is assigned to a formula F by an interpretation I. It refers to the following truth-valued functions, associated with the propositional connectives:

x	$\neg(x)$
FALSE	TRUE
TRUE	FALSE

[1] Note that $\bot$ and $\top$ are not atoms, according to this definition. They do not belong to the signature, and the semantics of propositional logic, defined below, treats them in a special way.

x	y	$\wedge(x, y)$	$\vee(x, y)$	$\rightarrow(x, y)$	$\leftrightarrow(x, y)$
FALSE	FALSE	FALSE	FALSE	TRUE	TRUE
FALSE	TRUE	FALSE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE	FALSE
TRUE	TRUE	TRUE	TRUE	TRUE	TRUE

For any formula F and any interpretation I, the truth value F^I that is *assigned* to F by I is defined recursively, as follows:

- for any atom F, $F^I = I(F)$,
- $\bot^I = \text{FALSE}$, $\top^I = \text{TRUE}$,
- $(\neg F)^I = \neg(F^I)$,
- $(F \odot G)^I = \odot(F^I, G^I)$ for every binary connective $\odot$.

If the underlying signature is finite then the set of interpretations is finite also, and the values of F^I for all interpretations I can be represented by a finite table, called the *truth table* of F.

If $F^I = \text{TRUE}$ then we say that the interpretation I *satisfies* F, or is a *model* of F (symbolically, $I \models F$).

A formula F is a *tautology* if every interpretation satisfies F. Two formulas, or sets of formulas, are *equivalent* to each other if they are satisfied by the same interpretations. It is clear that F is equivalent to G if and only if $F \leftrightarrow G$ is a tautology.

A set Γ of formulas is *satisfiable* if there exists an interpretation satisfying all formulas in Γ. We say that Γ *entails* a formula F (symbolically, $\Gamma \models F$) if every interpretation satisfying Γ satisfies F.[2]

To represent knowledge by propositional formulas, we choose a propositional signature σ such that interpretations of σ correspond to states of the system that we want to describe. Then any formula of σ represents a condition on states; a set of formulas can be viewed as a knowledge base; if a formula F is entailed by a knowledge base Γ then the condition expressed by F follows from the knowledge included in Γ.

Imagine, for instance, that Paul, Quentin and Robert share an office. Let us agree to use the atom p to express that Paul is in the office, and similarly q for Quentin and r for Robert. The knowledge base $\{p, q\}$ entails neither r nor $\neg r$. (The semantics of propositional logic does not incorporate the closed world assumption, discussed below in Section 6.2.4.) But if we add to the knowledge base the formula

$$\neg p \vee \neg q \vee \neg r, \tag{1.1}$$

expressing that at least one person is away, then the formula $\neg r$ (Robert is away) will be entailed.

Explicit definitions

Let Γ be a set of formulas of a propositional signature σ. To extend Γ by an *explicit definition* means to add to σ a new atom d, and to add to Γ a formula of the form

[2] Thus the relation symbol $\models$ is understood either as "satisfies" or as "entails" depending on whether its first operand is an interpretation or a set of formulas.

$d \leftrightarrow F$, where F is a formula of the signature σ. For instance, if

$$\sigma = \{p, q, r\}, \qquad \Gamma = \{p, q\},$$

as in the example above, then we can introduce an explicit definition that makes d an abbreviation for the formula $q \wedge r$ ("both Quentin and Robert are in"):

$$\sigma' = \{p, q, r, d\}, \qquad \Gamma' = \{p, q, d \leftrightarrow (q \wedge r)\}.$$

Adding an explicit definition to a knowledge base Γ is, in a sense, a trivial modification. For instance, there is a simple one-to-one correspondence between the set of models of Γ and the set of models of such an extension: a model of the extended set of formulas can be turned into the corresponding model of Γ by restricting it to σ. It follows that the extended set of formulas is satisfiable if and only if Γ is satisfiable. It follows also that adding an explicit definition produces a "conservative extension": a formula that does not contain the new atom d is entailed by the extended set of formulas if and only if it is entailed by Γ.

It is *not* true, however, that the extended knowledge base is *equivalent* to Γ. For instance, in the example above $\{p, q\}$ does not entail $d \leftrightarrow (q \wedge r)$, of course. This observation is related to the difference between two ways to convert a propositional formula to conjunctive normal form (that is, to turn it into a set of clauses): the more obvious method based on equivalent transformations on the one hand, and Tseitin's procedure, reviewed in Section 2.2 below, on the other. The latter can be thought of as a sequence of steps that add explicit definitions to the current set of formulas, interspersed with equivalent transformations that make formulas smaller and turn them into clauses. Tseitin's procedure is more efficient, but it does not produce a CNF equivalent to the input formula; it only gives us a conservative extension.

Natural deduction in propositional logic

Natural deduction, invented by Gerhard Gentzen [96], formalizes the process of introducing and discharging assumptions, common in informal mathematical proofs.

In the natural deduction system for propositional system described below, derivable objects are *sequents* of the form $\Gamma \Rightarrow F$, where F is a formula, and Γ is a finite set of formulas ("F under assumptions Γ"). For simplicity we only consider formulas that contain neither $\top$ nor $\leftrightarrow$; these connectives can be viewed as abbreviations. It is notationally convenient to write sets of assumptions as lists, and understand, for instance, $A_1, A_2 \Rightarrow F$ as shorthand for $\{A_1, A_2\} \Rightarrow F$, and $\Gamma, A \Rightarrow F$ as shorthand for $\Gamma \cup \{A\} \Rightarrow F$.

The axiom schemas of this system are

$$F \Rightarrow F$$

and

$$\Rightarrow F \vee \neg F.$$

The inference rules are shown in Fig. 1.1. Most of the rules can be can be divided into two groups—introduction rules (the left column) and elimination rules (the right column). Each of the introduction rules tells us how to *derive* a formula of some syntactic form. For instance, the conjunction introduction rule ($\wedge I$) shows that we can derive

$$(\wedge I)\ \frac{\Gamma\Rightarrow F \quad \Delta\Rightarrow G}{\Gamma,\Delta\Rightarrow F\wedge G} \qquad (\wedge E)\ \frac{\Gamma\Rightarrow F\wedge G}{\Gamma\Rightarrow F} \quad \frac{\Gamma\Rightarrow F\wedge G}{\Gamma\Rightarrow G}$$

$$(\vee I)\ \frac{\Gamma\Rightarrow F}{\Gamma\Rightarrow F\vee G} \quad \frac{\Gamma\Rightarrow G}{\Gamma\Rightarrow F\vee G} \qquad (\vee E)\ \frac{\Gamma\Rightarrow F\vee G \quad \Delta_1,F\Rightarrow H \quad \Delta_2,G\Rightarrow H}{\Gamma,\Delta_1,\Delta_2\Rightarrow H}$$

$$(\rightarrow I)\ \frac{\Gamma,F\Rightarrow G}{\Gamma\Rightarrow F\rightarrow G} \qquad (\rightarrow E)\ \frac{\Gamma\Rightarrow F \quad \Delta\Rightarrow F\rightarrow G}{\Gamma,\Delta\Rightarrow G}$$

$$(\neg I)\ \frac{\Gamma,F\Rightarrow\bot}{\Gamma\Rightarrow\neg F} \qquad (\neg E)\ \frac{\Gamma\Rightarrow F \quad \Delta\Rightarrow\neg F}{\Gamma,\Delta\Rightarrow\bot}$$

$$(C)\ \frac{\Gamma\Rightarrow\bot}{\Gamma\Rightarrow F}$$

$$(W)\ \frac{\Gamma\Rightarrow\Sigma}{\Gamma,\Delta\Rightarrow\Sigma}$$

Figure 1.1: Inference rules of propositional logic.

a conjunction if we derive both conjunctive terms; the disjunction introduction rules $(\vee I)$ show that we can derive a disjunction if we derive one of the disjunctive terms. Each of the elimination rules tells us how we can *use* a formula of some syntactic form. For instance, the conjunction elimination rules $(\wedge E)$ show that a conjunction can be used to derive any of its conjunctive terms; the disjunction elimination rules $(\vee E)$ shows that a disjunction can be used to justify reasoning by cases.

Besides introduction and elimination rules, the deductive system includes the contradiction rule (C) and the weakening rule (W).

In most inference rules, the set of assumptions in the conclusion is simply the union of the sets of assumptions of all the premises. The rules $(\rightarrow I)$, $(\neg I)$ and $(\vee E)$ are exceptions; when one of these rule is applied, some of the assumptions from the premises are "discharged".

An example of a proof in this system is shown in Fig. 1.2. This proof can be informally summarized as follows. Assume $\neg p$, $q \rightarrow r$ and $p \vee q$. We will prove r by cases.

Case 1: p. This contradicts the assumption $\neg p$, so that r follows.

Case 2: q. In view of the assumption $q \rightarrow r$, r follows also.

Consequently, from the assumptions $\neg p$ and $q \rightarrow r$ we have derived $(p \vee q) \rightarrow r$.

The deductive system described above is sound and complete: a sequent $\Gamma \Rightarrow F$ is provable in it if and only if $\Gamma \models F$. The first proof of a completeness theorem for propositional logic (involving a different deductive system) is due to Post [216].

Meta-level and object-level proofs

When we want to establish that a formula F is entailed by a knowledge base Γ, the straightforward approach is to use the definition of entailment, that is, to reason about interpretations of the underlying signature. For instance, to check that the formulas $\neg p$ and $q \rightarrow r$ entail $(p \vee q) \rightarrow r$ we can argue that no interpretation of the signature $\{p, q, r\}$ can satisfy both $\neg p$ and $q \rightarrow r$ unless it satisfies $(p \vee q) \rightarrow r$ as well.

A sound deductive system provides an "object-level" alternative to this meta-level approach. Once we proved the sequent $\Gamma \Rightarrow F$ in the deductive system described above, we have established that Γ entails F. For instance, the claim that the formulas $\neg p$ and $q \rightarrow r$ entail $(p \vee q) \rightarrow r$ is justified by Fig. 1.2. As a matter of convenience, informal summaries, as in the example above, can be used instead of formal proofs.

1.	$\neg p \Rightarrow \neg p$		— axiom
2.	$q \to r \Rightarrow q \to r$		— axiom
3.	$p \vee q \Rightarrow p \vee q$		— axiom
4.	$p \Rightarrow p$		— axiom
5.	$p, \neg p \Rightarrow \bot$		— by $(\neg E)$ from 4, 1
6.	$p, \neg p \Rightarrow r$		— by (C) from 5
7.	$q \Rightarrow q$		— axiom
8.	$q, q \to r \Rightarrow r$		— by $(\to E)$ from 7, 2
9.	$p \vee q, \neg p, q \to r \Rightarrow r$		— by $(\vee E)$ from 3, 6, 8
10.	$\neg p, q \to r \Rightarrow (p \vee q) \to r$		— by $(\to I)$ from 9

Figure 1.2: A proof in propositional logic.

Since the system is not only sound but also complete, the object-level approach to establishing entailment is, in principle, always applicable.

Object-level proofs can be used also to establish general properties of entailment. Consider, for instance, the following fact: for any formulas $F_1, \ldots, F_n$, the implications $F_i \to F_{i+1}$ $(i = 1, \ldots, n-1)$ entail $F_1 \to F_n$. We can justify it by saying that if we assume F_1 then $F_2, \ldots, F_n$ will consecutively follow using the given implications. By saying this, we have outlined a method for constructing a proof of the sequent

$$F_1 \to F_2, \ldots, F_{n-1} \to F_n \quad \Rightarrow \quad F_1 \to F_n$$

that consists of $n-1$ implication eliminations followed by an implication introduction.

1.2.2 First-Order Logic

Syntax

In first-order logic, a *signature* is a set of symbols of two kinds—*function constants* and *predicate constants*—with a nonnegative integer, called the *arity*, assigned to each symbol. Function constants of arity 0 are called *object constants*; predicate constants of arity 0 are called *propositional constants*.

Object variables are elements of some fixed infinite sequence of symbols, for instance, $x, y, z, x_1, y_1, z_1, \ldots$. *Terms* of a signature σ are formed from object variables and from function constants of σ. An *atomic formula* of σ is an expression of the form $P(t_1, \ldots, t_n)$ or $t_1 = t_2$, where P is a predicate constant of arity n, and each t_i is a term of σ.[3] Formulas are formed from atomic formulas using propositional connectives and the quantifiers $\forall, \exists$.

An occurrence of a variable v in a formula F is *bound* if it belongs to a subformula of F that has the form $\forall v G$ or $\exists v G$; otherwise it is *free*. If at least one occurrence of v in F is free then we say that v is a *free variable* of F. Note that a formula can contain both free and bound occurrences of the same variable, as in

$$P(x) \wedge \exists x\, Q(x). \tag{1.2}$$

[3] Note that equality is not a predicate constant, according to this definition. Although syntactically it is similar to binary predicate constants, it does not belong to the signature, and the semantics of first-order logic, defined below, treats equality in a special way.

We can avoid such cases by renaming bound occurrences of variables:

$$P(x) \wedge \exists x_1 Q(x_1). \tag{1.3}$$

Both formulas have the same meaning: x has the property P, and there exists an object with the property Q.

A *closed* formula, or a *sentence*, is a formula without free variables. The *universal closure* of a formula F is the sentence $\forall v_1 \dots v_n F$, where $v_1, \dots, v_n$ are the free variables of F.

The result of the *substitution* of a term t for a variable v in a formula F is the formula obtained from F by simultaneously replacing each free occurrence of v by t. When we intend to consider substitutions for v in a formula, it is convenient to denote this formula by an expression like $F(v)$; then we can denote the result of substituting a term t for v in this formula by $F(t)$.

By $\exists! v F(v)$ ("there exists a unique v such that $F(v)$") we denote the formula

$$\exists v \forall w (F(w) \leftrightarrow v = w),$$

where w is the first variable that does not occur in $F(v)$.

A term t is *substitutable* for a variable v in a formula F if, for each variable w occurring in t, no subformula of F that has the form $\forall w G$ or $\exists w G$ contains an occurrence of v which is free in F. (Some authors say in this case that t is free for x in F.) This condition is important because when it is violated, the formula obtained by substituting t for v in F does not usually convey the intended meaning. For instance, the formula $\exists x(f(x) = y)$ expresses that y belongs to the range of f. If we substitute, say, the term $g(a, z)$ for y in this formula then we will get the formula $\exists x(f(x) = g(a, z))$, which expresses that $g(a, z)$ belongs to the range of f—as one would expect. If, however, we substitute the term $g(a, x)$ instead, the result $\exists x(f(x) = g(a, x))$ will *not* express that $g(a, x)$ belongs to the range of f. This is related to the fact that the term $g(a, x)$ is not substitutable for y in $\exists x(f(x) = y)$; the occurrence of x resulting from this substitution is "captured" by the quantifier at the beginning of the formula. To express that $g(a, x)$ belongs to the range of f, we should first rename x in the formula $\exists x(f(x) = y)$ using, say, the variable x_1. The substitution will produce then the formula $\exists x_1(f(x_1) = g(a, x))$.

Semantics

An *interpretation* (or *structure*) of a signature σ consists of

- a nonempty set $|I|$, called the *universe* (or *domain*) of I,
- for every object constant c of σ, an element c^I of $|I|$,
- for every function constant f of σ of arity $n > 0$, a function f^I from $|I|^n$ to $|I|$,
- for every propositional constant P of σ, an element P^I of {FALSE, TRUE},
- for every predicate constant R of σ of arity $n > 0$, a function R^I from $|I|^n$ to {FALSE, TRUE}.

The semantics of first-order logic defines, for any sentence F and any interpretation I of a signature σ, the truth value F^I that is assigned to F by I. Note that the

definition does not apply to formulas with free variables. (Whether $\exists x(f(x) = y)$ is true or false, for instance, is not completely determined by the universe and by the function representing f; the answer depends also on the value of y within the universe.) For this reason, stating correctly the clauses for quantifiers in the recursive definition of F^I is a little tricky. One possibility is to extend the signature σ by "names" for all elements of the universe, as follows.

Consider an interpretation I of a signature σ. For any element ξ of its universe $|I|$, select a new symbol ξ^*, called the *name* of ξ. By σ^I we denote the signature obtained from σ by adding all names ξ^* as object constants. The interpretation I can be extended to the new signature σ^I by defining $(\xi^*)^I = \xi$ for all $\xi \in |I|$.

For any term t of the extended signature that does not contain variables, we will define recursively the element t^I of the universe that is *assigned* to t by I. If t is an object constant then t^I is part of the interpretation I. For other terms, t^I is defined by the equation

$$f(t_1, \ldots, t_n)^I = f^I(t_1^I, \ldots, t_n^I)$$

for all function constants f of arity $n > 0$.

Now we are ready to define F^I for every sentence F of the extended signature σ^I. For any propositional constant P, P^I is part of the interpretation I. Otherwise, we define:

- $R(t_1, \ldots, t_n)^I = R^I(t_1^I, \ldots, t_n^I)$,
- $\bot^I = \text{FALSE}$, $\top^I = \text{TRUE}$,
- $(\neg F)^I = \neg(F^I)$,
- $(F \odot G)^I = \odot(F^I, G^I)$ for every binary connective $\odot$,
- $\forall w F(w)^I = \text{TRUE}$ if $F(\xi^*)^I = \text{TRUE}$ for all $\xi \in |I|$,
- $\exists w F(w)^I = \text{TRUE}$ if $F(\xi^*)^I = \text{TRUE}$ for some $\xi \in |I|$.

We say that an interpretation I *satisfies* a sentence F, or is a *model* of F, and write $I \models F$, if $F^I = \text{TRUE}$. A sentence F is *logically valid* if every interpretation satisfies F. Two sentences, or sets of sentences, are *equivalent* to each other if they are satisfied by the same interpretations. A formula with free variables is said to be *logically valid* if its universal closure is logically valid. Formulas F and G that may contain free variables are *equivalent* to each other if $F \leftrightarrow G$ is logically valid.

A set Γ of sentences is *satisfiable* if there exists an interpretation satisfying all sentences in Γ. A set Γ of sentences *entails* a formula F (symbolically, $\Gamma \models F$) if every interpretation satisfying Γ satisfies the universal closure of F.

Sorts

Representing knowledge in first-order languages can be often simplified by introducing sorts, which requires that the definitions of the syntax and semantics above be generalized.

Besides function constants and predicate constants, a many-sorted signature includes symbols called *sorts*. In addition to an arity n, we assign to every function

constant and every predicate constant its *argument sorts* $s_1, \ldots, s_n$; to every function constant we assign also its *value sort* s_{n+1}. For instance, in the situation calculus (Section 16.1), the symbols *situation* and *action* are sorts; *do* is a binary function symbol with the argument sorts *action* and *situation*, and the value sort *situation*.

For every sort s, we assume a separate infinite sequence of variables of that sort. The recursive definition of a term assigns a sort to every term. Atomic formulas are expressions of the form $P(t_1, \ldots, t_n)$, where the sorts of the terms $t_1, \ldots, t_n$ are the argument sorts of P, and also expressions $t_1 = t_2$ where t_1 and t_2 are terms of the same sort.

An interpretation, in the many-sorted setting, includes a separate nonempty universe $|I|^s$ for each sort s. Otherwise, extending the definition of the semantics to many-sorted languages is straightforward.

A further extension of the syntax and semantics of first-order formulas allows one sort to be a "subsort" of another. For instance, when we talk about the blocks world, it may be convenient to treat the sort *block* as a subsort of the sort *location*. Let b_1 and b_2 be object constants of the sort *block*, let *table* be an object constant of the sort *location*, and let *on* be a binary function constant with the argument sorts *block* and *location*. Not only $on(b_1, table)$ will be counted as a term, but also $on(b_1, b_2)$, because the sort of b_2 is a subsort of the second argument sort of *on*.

Generally, a subsort relation is an order (reflexive, transitive and anti-symmetric relation) on the set of sorts. In the recursive definition of a term, $f(t_1, \ldots, t_n)$ is a term if the sort of each t_i is a subsort of the ith argument sort of f. The condition on sorts in the definition of atomic formulas $P(t_1, \ldots, t_n)$ is similar. An expression $t_1 = t_2$ is considered an atomic formula if the sorts of t_1 and t_2 have a common supersort. In the definition of an interpretation, $|I|^{s_1}$ is required to be a subset of $|I|^{s_2}$ whenever s_1 is a subsort of s_2.

In the rest of this chapter we often assume for simplicity that the underlying signature is nonsorted.

Uniqueness of names

To talk about Paul, Quentin and Robert from Section 1.2.1 in a first-order language, we can introduce the signature consisting of the object constants *Paul*, *Quentin*, *Robert* and the unary predicate constant *in*, and then use the atomic sentences

$$in(Paul), \quad in(Quentin), \quad in(Robert) \tag{1.4}$$

instead of the atoms p, q, r from the propositional representation.

However some interpretations of this signature are unintuitive and do not correspond to any of the 8 interpretations of the propositional signature $\{p, q, r\}$. Those are the intepretations that map two, or even all three, object constants to the same element of the universe. (The definition of an interpretation in first-order logic does not require that c_1^I be different from c_2^I for distinct object constants c_1, c_2.) We can express that $Paul^I$, $Quentin^I$ and $Robert^I$ are pairwise distinct by saying that I satisfies the "unique name conditions"

$$Paul \neq Quentin, \quad Paul \neq Robert, \quad Quentin \neq Robert. \tag{1.5}$$

Generally, the *unique name assumption* for a signature σ is expressed by the formulas

$$\forall x_1 \dots x_m y_1 \dots y_n (f(x_1, \dots, x_m) \neq g(y_1, \dots, y_n)) \tag{1.6}$$

for all pairs of distinct function constants f, g, and

$$\forall x_1 \dots x_n y_1 \dots y_n (f(x_1, \dots, x_n) = f(y_1, \dots, y_n) \\ \rightarrow (x_1 = y_1 \wedge \dots \wedge x_n = y_n)) \tag{1.7}$$

for all function constants f of arity > 0. These formulas entail $t_1 \neq t_2$ for any distinct variable-free terms t_1, t_2.

The set of equality axioms that was introduced by Keith Clark [57] and is often used in the theory of logic programming includes, in addition to (1.6) and (1.7), the axioms $t \neq x$, where t is a term containing x as a proper subterm.

Domain closure

Consider the first-order counterpart of the propositional formula (1.1), expressing that at least one person is away:

$$\neg in(Paul) \vee \neg in(Quentin) \vee \neg in(Robert). \tag{1.8}$$

The same idea can be also conveyed by the formula

$$\exists x \neg in(x). \tag{1.9}$$

But sentences (1.8) and (1.9) are not equivalent to each other: the former entails the latter, but not the other way around. Indeed, the definition of an interpretation in first-order logic does not require that every element of the universe be equal to c^I for some object constant c. Formula (1.9) interprets "at least one" as referring to a certain group that includes *Paul*, *Quentin* and *Robert*, and may also include others.

If we want to express that every element of the universe corresponds to one of the three explicitly named persons then this can be done by the formula

$$\forall x (x = Paul \vee x = Quentin \vee x = Robert). \tag{1.10}$$

This "domain closure condition" entails the equivalence between (1.8) and (1.9); more generally, it entails the equivalences

$$\forall x F(x) \leftrightarrow F(Paul) \wedge F(Quentin) \wedge F(Robert),$$
$$\exists x F(x) \leftrightarrow F(Paul) \vee F(Quentin) \vee F(Robert)$$

for any formula $F(x)$. These equivalences allow us to replace all quantifiers in an arbitrary formula with multiple conjunctions and disjunctions. Furthermore, under the unique name assumption (1.5) any equality between two object constants can be equivalently replaced by $\top$ or $\bot$, depending on whether the constants are equal to each other. The result of these transformations is a propositional combination of the atomic sentences (1.4).

Generally, consider a signature σ containing finitely many object constants $c_1, \dots, c_n$ are no function constants of arity > 0. The *domain closure assumption*

for σ is the formula

$$\forall x(x = c_1 \vee \cdots \vee x = c_n). \tag{1.11}$$

The interpretations of σ that satisfy both the unique name assumption $c_1 \neq c_j$ $(1 \leqslant i < j \leqslant n)$ and the domain closure assumption (1.11) are essentially identical to the interpretations of the propositional signature that consists of all atomic sentences of σ other than equalities. Any sentence F of σ can be transformed into a formula F' of this propositional signature such that the unique name and domain closure assumptions entail $F' \leftrightarrow F$. In this sense, these assumptions turn first-order sentences into abbreviations for propositional formulas.

The domain closure assumption in the presence of function constant of arity > 0 is discussed in Sections 1.2.2 and 1.2.3.

Reification

The first-order language introduced in Section 1.2.2 has variables for people, such as Paul and Quentin, but not for places, such as their office. In this sense, people are "reified" in that language, and places are not. To reify places, we can add them to the signature as a second sort, add *office* as an object constant of that sort, and turn *in* into a binary predicate constant with the argument sorts *person* and *place*. In the modified language, the formula *in*(*Paul*) will turn into *in*(*Paul*, *office*).

Reification makes the language more expressive. For instance, having reified places, we can say that every person has a unique location:

$$\forall x \exists! p \; in(x, p). \tag{1.12}$$

There is no way to express this idea in the language from Section 1.2.2.

As another example illustrating the idea of reification, compare two versions of the situation calculus. We can express that block b_1 is clear in the initial situation S_0 by writing either

$$clear(b_1, S_0) \tag{1.13}$$

or

$$Holds(clear(b_1), S_0). \tag{1.14}$$

In (1.13), *clear* is a binary predicate constant; in (1.14), *clear* is a unary function constant. Formula (1.14) is written in the version of the situation calculus in which (relational) fluents are reified; *fluent* is the first argument sort of the predicate constant *Holds*. The version of the situation calculus introduced in Section 16.1 is the more expressive version, with reified fluents. Expression (1.13) is viewed there as shorthand for (1.14).

Explicit definitions in first-order logic

Let Γ be a set of sentences of a signature σ. To extend Γ by an *explicit definition of a predicate constant* means to add to σ a new predicate constant P of some arity n, and to add to Γ a sentence of the form

$$\forall v_1 \ldots v_n(P(v_1, \ldots, v_n) \leftrightarrow F),$$

where $v_1, \ldots, v_n$ are distinct variables and F is a formula of the signature σ. About the effect of such an extension we can say the same as about the effect of adding an explicit definition to a set of propositional formulas (Section 1.2.1): there is an obvious one-to-one correspondence between the models of the original knowledge base and the models of the extended knowledge base.

With function constants, the situation is a little more complex. To extend a set Γ of sentences of a signature σ by an *explicit definition of a function constant* means to add to σ a new function constant f, and to add to Γ a sentence of the form

$$\forall v_1 \ldots v_n v(f(v_1, \ldots, v_n) = v \leftrightarrow F),$$

where $v_1, \ldots, v_n, v$ are distinct variables and F is a formula of the signature σ such that Γ entails the sentence

$$\forall v_1 \ldots v_n \exists! v F.$$

The last assumption is essential: if it does not hold then adding a function constant along with the corresponding axiom would eliminate some of the models of Γ.

For instance, if Γ entails (1.12) then we can extend Γ by the explicit definition of the function constant *location*:

$$\forall x p(location(x) = p \leftrightarrow in(x, p)).$$

Natural deduction with quantifiers and equality

The natural deduction system for first-order logic includes all axiom schemas and inference rules shown in Section 1.2.1 and a few additional postulates. First, we add the introduction and elimination rules for quantifiers:

$$(\forall I)\ \frac{\Gamma \Rightarrow F(v)}{\Gamma \Rightarrow \forall v F(v)} \qquad (\forall E)\ \frac{\Gamma \Rightarrow \forall v F(v)}{\Gamma \Rightarrow F(t)}$$

where v is not a free variable of any formula in Γ — where t is substitutable for v in $F(v)$

$$(\exists I)\ \frac{\Gamma \Rightarrow F(t)}{\Gamma \Rightarrow \exists v F(v)} \qquad (\exists E)\ \frac{\Gamma \Rightarrow \exists v F(v) \quad \Delta, F(v) \Rightarrow G}{\Gamma, \Delta \Rightarrow G}$$

where t is substitutable for v in $F(v)$ — where v is not a free variable of any formula in Δ, G

Second, postulates for equality are added: the axiom schema expressing its reflexivity

$$\Rightarrow t = t$$

and the inference rules for replacing equals by equals:

$$(Repl)\ \frac{\Gamma \Rightarrow t_1 = t_2 \quad \Delta \Rightarrow F(t_1)}{\Gamma, \Delta \Rightarrow F(t_2)} \qquad \frac{\Gamma \Rightarrow t_1 = t_2 \quad \Delta \Rightarrow F(t_2)}{\Gamma, \Delta \Rightarrow F(t_1)}$$

where t_1 and t_2 are terms substitutable for v in $F(v)$.

This formal system is sound and complete: for any finite set Γ of sentences and any formula F, the sequent $\Gamma \Rightarrow F$ is provable if and only if $\Gamma \models F$. The completeness of (a different formalization of) first-order logic was proved by Gödel [100].

1.		$(1.9) \Rightarrow (1.9)$		— axiom
2.		$\neg in(x) \Rightarrow \neg in(x)$		— axiom
3.		$x = P \Rightarrow x = P$		— axiom
4.	$x = P, \neg in(x)$	$\Rightarrow \neg in(P)$		— by *Repl* from 3, 2
5.	$x = P, \neg in(x)$	$\Rightarrow \neg in(P) \vee \neg in(Q)$		— by $(\vee I)$ from 4
6.	$x = P, \neg in(x)$	$\Rightarrow (1.8)$		— by $(\vee I)$ from 5
7.		$x = Q \Rightarrow x = Q$		— axiom
8.	$x = Q, \neg in(x)$	$\Rightarrow \neg in(Q)$		— by *Repl* from 7, 2
9.	$x = Q, \neg in(x)$	$\Rightarrow \neg in(P) \vee \neg in(Q)$		— by $(\vee I)$ from 8
10.	$x = Q, \neg in(x)$	$\Rightarrow (1.8)$		— by $(\vee I)$ from 9
11.	$x = P \vee x = Q$	$\Rightarrow x = P \vee x = Q$		— axiom
12.	$x = P \vee x = Q, \neg in(x)$	$\Rightarrow (1.8)$		— by $(\vee E)$ from 11, 6, 10
13.		$x = R \Rightarrow x = R$		— axiom
14.	$x = R, \neg in(x)$	$\Rightarrow \neg in(R)$		— by *Repl* from 13, 2
15.	$x = R, \neg in(x)$	$\Rightarrow (1.8)$		— by $(\vee I)$ from 14
16.		$(1.10) \Rightarrow (1.10)$		— axiom
17.		$(1.10) \Rightarrow x = P \vee x = Q$	$\vee x = R$	— by $(\forall E)$ from 16
18.	$(1.10), \neg in(x)$	$\Rightarrow (1.8)$		— by $(\vee E)$ from 17, 12, 15
19.	$(1.9), (1.10)$	$\Rightarrow (1.8)$		— by $(\exists E)$ from 1, 18

Figure 1.3: A proof in first-order logic.

As in the propositional case (Section 1.2.1), the soundness theorem justifies establishing entailment in first-order logic by an object-level argument. For instance, we can prove the claim that (1.8) is entailed by (1.9) and (1.10) as follows: take x such that $\neg in(x)$ and consider the three cases corresponding to the disjunctive terms of (1.10); in each case, one of the disjunctive terms of (1.8) follows. This argument is an informal summary of the proof shown in Fig. 1.3, with the names *Paul*, *Quentin*, *Robert* replaced by P, Q, R.

Since proofs in the deductive system described above can be effectively enumerated, from the soundness and completeness of the system we can conclude that the set of logically valid sentences is recursively enumerable. But it is not recursive [56], even if the underlying signature consists of a single binary predicate constant, and even if we disregard formulas containing equality [135].

As discussed in Section 3.3.1, most descriptions logics can be viewed as decidable fragments of first-order logic.

Limitations of first-order logic

The sentence

$$\forall xy(Q(x, y) \leftrightarrow P(y, x))$$

expresses that Q is the inverse of P. Does there exist a first-order sentence expressing that Q is the *transitive closure* of P? To be more precise, does there exist a sentence F of the signature $\{P, Q\}$ such that an interpretation I of this signature satisfies F if and only if Q^I is the transitive closure of P^I?

The answer to this question is no. From the perspective of knowledge representation, this is an essential limitation, because the concept of transitive closure is the

mathematical counterpart of the important commonsense idea of reachability. As discussed in Section 1.2.3 below, one way to overcome this limitation is to turn to second-order logic.

Another example illustrating the usefulness of second-order logic in knowledge representation is related to the idea of domain closure (Section 1.2.2). If the underlying signature contains the object constants $c_1, \ldots, c_n$ and no function constants of arity > 0 then sentence (1.11) expresses the domain closure assumption: an interpretation I satisfies (1.11) if and only if

$$|I| = \{c_1^I, \ldots, c_n^I\}.$$

Consider now the signature consisting of the object constant c and the unary function constant f. Does there exist a first-order sentence expressing the domain closure assumption for this signature? To be precise, we would like to find a sentence F such that an interpretation I satisfies F if and only if

$$|I| = \{c^I, f(c)^I, f(f(c))^I, \ldots\}.$$

There is no first-order sentence with this property.

Similarly, first-order languages do not allow us to state Reiter's foundational axiom expressing that each situation is the result of performing a sequence of actions in the initial situation ([225, Section 4.2.2]; see also Section 16.3 below).

1.2.3 Second-Order Logic

Syntax and semantics

In second-order logic, the definition of a signature remains the same (Section 1.2.2). But its syntax is richer, because, along with object variables, we assume now an infinite sequence of *function variables* of arity n for each $n > 0$, and an infinite sequence of *predicate variables* of arity n for each $n \geqslant 0$. Object variables are viewed as function variables of arity 0.

Function variables can be used to form new terms in the same way as function constants. For instance, if α is a unary function variable and c is an object constant then $\alpha(c)$ is a term. Predicate variables can be used to form atomic formulas in the same way as predicate constants. In non-atomic formulas, function and predicate variables can be bound by quantifiers in the same way as object variables. For instance,

$$\forall\alpha\beta\exists\gamma\forall x(\gamma(x) = \alpha(\beta(x)))$$

is a sentence expressing the possibility of composing any two functions. (When we say that a second-order formula is a sentence, we mean that all occurrences of all variables in it are bound, including function and predicate variables.)

Note that $\alpha = \beta$ is not an atomic formula, because unary function variables are not terms. But this expression can be viewed as shorthand for the formula

$$\forall x(\alpha(x) = \beta(x)).$$

Similarly, the expression $p = q$, where p and q are unary predicate variables, can be viewed as shorthand for

$$\forall x(p(x) \leftrightarrow q(x)).$$

The condition "Q is the transitive closure of P" can be expressed by the second-order sentence

$$\forall xy(Q(x, y) \leftrightarrow \forall q(F(q) \rightarrow q(x, y))), \tag{1.15}$$

where $F(q)$ stands for

$$\forall x_1 y_1(P(x_1, y_1) \rightarrow q(x_1, y_1))$$
$$\wedge \forall x_1 y_1 z_1((q(x_1, y_1) \wedge q(y_1, z_1)) \rightarrow q(x_1, z_1))$$

(Q is the intersection of all transitive relations containing P).

The domain closure assumption for the signature $\{c, f\}$ can be expressed by the sentence

$$\forall p(G(p) \rightarrow \forall x\ p(x)), \tag{1.16}$$

where $G(p)$ stands for

$$p(c) \wedge \forall x(p(x) \rightarrow p(f(x)))$$

(any set that contains c and is closed under f covers the whole universe).

The definition of an interpretation remains the same (Section 1.2.2). The semantics of second-order logic defines, for each sentence F and each interpretation I, the corresponding truth value F^I. In the clauses for quantifiers, whenever a quantifier binds a function variable, names of arbitrary functions from $|I|^n$ to I are substituted for it; when a quantifier binds a predicate variable, names of arbitrary functions from $|I|^n$ to {FALSE, TRUE} are substituted.

Quantifiers binding a propositional variable p can be always eliminated: $\forall pF(p)$ is equivalent to $F(\bot) \wedge F(\top)$, and $\exists pF(p)$ is equivalent to $F(\bot) \vee F(\top)$. In the special case when the underlying signature consists of propositional constants, second-order formulas (in prenex form) are known as *quantified Boolean formulas* (see Section 2.5.1). The equivalences above allow us to rewrite any such formula in the syntax of propositional logic. But a sentence containing predicate variables of arity > 0 may not be equivalent to any first-order sentence; (1.15) and (1.16) are examples of such "hard" cases.

Object-level proofs in second-order logic

In this section we consider a deductive system for second-order logic that contains all postulates from Sections 1.2.1 and 1.2.2; in rules $(\forall E)$ and $(\exists I)$, if v is a function variable of arity > 0 then t is assumed to be a function variable of the same arity, and similarly for predicate variables. In addition, we include two axiom schemas asserting the existence of predicates and functions. One is the axiom schema of comprehension

$$\Rightarrow \exists p \forall v_1 \dots v_n(p(v_1, \dots, v_n) \leftrightarrow F),$$

where $v_1, \dots, v_n$ are distinct object variables, and p is not free in F. (Recall that $\leftrightarrow$ is not allowed in sequents, but we treat $F \leftrightarrow G$ as shorthand for $(F \rightarrow G) \wedge (G \rightarrow F)$.)

1.		F	$\Rightarrow$ F	— axiom
2.		F	$\Rightarrow$ $p(x) \to p(y)$	— by $(\forall E)$ from 1
3.			$\Rightarrow$ $\exists p \forall z(p(z) \leftrightarrow x = z)$	— axiom (comprehension)
4.	$\forall z(p(z) \leftrightarrow x = z)$		$\Rightarrow$ $\forall z(p(z) \leftrightarrow x = z)$	— axiom
5.	$\forall z(p(z) \leftrightarrow x = z)$		$\Rightarrow$ $p(x) \leftrightarrow x = x$	— by $(\forall E)$ from 4
6.	$\forall z(p(z) \leftrightarrow x = z)$		$\Rightarrow$ $x = x \to p(x)$	— by $(\wedge E)$ from 5
7.			$\Rightarrow$ $x = x$	— axiom
8.	$\forall z(p(z) \leftrightarrow x = z)$		$\Rightarrow$ $p(x)$	— by $(\to E)$ from 7, 6
9.	$F, \forall z(p(z) \leftrightarrow x = z)$		$\Rightarrow$ $p(y)$	— by $(\to E)$ from 8, 2
10.	$\forall z(p(z) \leftrightarrow x = z)$		$\Rightarrow$ $p(y) \leftrightarrow x = y$	— by $(\forall E)$ from 4
11.	$\forall z(p(z) \leftrightarrow x = z)$		$\Rightarrow$ $p(y) \to x = y$	— by $(\wedge E)$ from 10
12.	$F, \forall z(p(z) \leftrightarrow x = z)$		$\Rightarrow$ $x = y$	— by $(\to E)$ from 9, 11
13.		F	$\Rightarrow$ $x = y$	— by $(\exists E)$ from 1, 12
14.			$\Rightarrow$ $F \to x = y$	— by $(\to I)$ from 13

Figure 1.4: A proof in second-order logic. F stands for $\forall p(p(x) \to p(y))$.

The other is the axioms of choice

$$\Rightarrow \forall v_1 \ldots v_n \exists v_{n+1} p(v_1, \ldots, v_{n+1}) \\ \to \exists \alpha \forall v_1 \ldots v_n (p(v_1, \ldots, v_n, \alpha(v_1, \ldots, v_n)),$$

where $v_1, \ldots, v_{n+1}$ are distinct object variables.

This deductive system is sound but incomplete. Adding any sound axioms or inference rules would not make it complete, because the set of logically valid second-order sentences is not recursively enumerable.

As in the case of first-order logic, the availability of a sound deductive system allows us to establish second-order entailment by object-level reasoning. To illustrate this point, consider the formula

$$\forall p(p(x) \to p(y)) \to x = y,$$

which can be thought of as a formalization of "Leibniz's principle of equality": two objects are equal if they share the same properties. Its logical validity can be justified as follows. Assume $\forall p(p(x) \to p(y))$, and take p to be the property of being equal to x. Clearly x has this property; consequently y has this property as well, that is, $x = y$. This argument is an informal summary of the proof shown in Fig. 1.4.

1.3 Automated Theorem Proving

Automated theorem proving is the study of techniques for programming computers to search for proofs of formal assertions, either fully automatically or with varying degrees of human guidance. This area has potential applications to hardware and software verification, expert systems, planning, mathematics research, and education.

Given a set A of axioms and a logical consequence B, a theorem proving program should, ideally, eventually construct a proof of B from A. If B is not a consequence of A, the program may run forever without coming to any definite conclusion. This is the best one can hope for, in general, in many logics, and indeed even this is not always possible. In principle, theorem proving programs can be written just by enumerating

all possible proofs and stopping when a proof of the desired statement is found, but this approach is so inefficient as to be useless. Much more powerful methods have been developed.

History of theorem proving

Despite the potential advantages of machine theorem proving, it was difficult initially to obtain any kind of respectable performance from machines on theorem proving problems. Some of the earliest automatic theorem proving methods, such as those of Gilmore [99], Prawitz [217], and Davis and Putnam [70] were based on Herbrand's theorem, which gives an enumeration process for testing if a theorem of first-order logic is true. Davis and Putnam used Skolem functions and conjunctive normal form clauses, and generated elements of the Herbrand universe exhaustively, while Prawitz showed how this enumeration could be guided to only generate terms likely to be useful for the proof, but did not use Skolem functions or clause form. Later Davis [66] showed how to realize this same idea in the context of clause form and Skolem functions. However, these approaches turned out to be too inefficient. The *resolution* approach of Robinson [229, 230] was developed in about 1963, and led to a significant advance in first-order theorem provers. This approach, like that of Davis and Putnam [70], used clause form and Skolem functions, but made use of a *unification* algorithm to find the terms most likely to lead to a proof. Robinson also used the resolution inference rule which in itself is all that is needed for theorem proving in first-order logic. The theorem proving group at Argonne, Illinois took the lead in implementing resolution theorem provers, with some initial success on group theory problems that had been intractable before. They were even able to solve some previously open problems using resolution theorem provers. For a discussion of the early history of mechanical theorem proving, see [67].

About the same time, Maslov [168] developed the *inverse method* which has been less widely known than resolution in the West. This method was originally defined for classical first-order logic without function symbols and equality, and for formulas having a quantifier prefix followed by a disjunction of conjunctions of clauses. Later the method was extended to formulas with function symbols. This method was used not only for theorem proving but also to show the decidability of some classes of first-order formulas. In the inverse method, substitutions were originally represented as sets of equations, and there appears to have been some analogue of most general unifiers. The method was implemented for classical first-order logic by 1968. The inverse method is based on forward reasoning to derive a formula. In terms of implementation, it is competitive with resolution, and in fact can be simulated by resolution with the introduction of new predicate symbols to define subformulas of the original formula. For a readable exposition of the inverse method, see [159]. For many extensions of the method, see [71].

In the West, the initial successes of resolution led to a rush of enthusiasm, as resolution theorem provers were applied to question-answering problems, situation calculus problems, and many others. It was soon discovered that resolution had serious inefficiencies, and a long series of refinements were developed to attempt to overcome them. These included the unit preference rule, the set of support strategy, hyper-resolution, paramodulation for equality, and a nearly innumerable list of other refinements. The initial enthusiasm for resolution, and for automated deduction in general, soon wore

off. This reaction led, for example, to the development of specialized decision procedures for proving theorems in certain theories [190, 191] and the development of expert systems.

However, resolution and similar approaches continued to be developed. Data structures were developed permitting the resolution operation to be implemented much more efficiently, which were eventually greatly refined [222] as in the Vampire prover [227]. One of the first provers to employ such techniques was Stickel's Prolog Technology Theorem Prover [252]. Techniques for parallel implementations of provers were also eventually considered [34]. Other strategies besides resolution were developed, such as model elimination [162], which led eventually to logic programming and Prolog, the matings method for higher-order logic [3], and Bibel's connection method [28]. Though these methods are not resolution based, they did preserve some of the key concepts of resolution, namely, the use of unification and the combination of unification with inference in clause form first-order logic. Two other techniques used to improve the performance of provers, especially in competitions [253], are *strategy selection* and *strategy scheduling*. Strategy selection means that different theorem proving strategies and different settings of the coefficients are used for different kinds of problems. Strategy scheduling means that even for a given kind of problem, many strategies are used, one after another, and a specified amount of time is allotted to each one. Between the two of these approaches, there is considerable freedom for imposing an outer level of control on the theorem prover to tailor its performance to a given problem set.

Some other provers dealt with higher-order logic, such as the TPS prover of Andrews and others [4, 5] and the interactive NqTHM and ACL2 provers of Boyer, Moore, and Kaufmann [142, 141] for proofs by mathematical induction. Today, a variety of approaches including formal methods and theorem proving seem to be accepted as part of the standard AI tool kit.

Despite early difficulties, the power of theorem provers has continued to increase. Notable in this respect is Otter [177], which is widely distributed, and coded in C with very efficient data structures. Prover9 is a more recent prover of W. McCune in the same style, and is a successor of Otter. The increasing speed of hardware has also significantly aided theorem provers. An impetus was given to theorem proving research by McCune's solution of the Robbins problem [176] by a first-order equational theorem prover derived from Otter. The Robbins problem is a first-order theorem involving equality that had been known to mathematicians for decades but which no one was able to solve. McCune's prover was able to find a proof after about a week of computation. Many other proofs have also been found by McCune's group on various provers; see for example the web page http://www.cs.unm.edu/~veroff/MEDIAN_ALGEBRA/. Now substantial theorems in mathematics whose correctness is in doubt can be checked by interactive theorem provers [196].

First-order theorem provers vary in their user interfaces, but most of them permit formulas to be entered in clause form in a reasonable syntax. Some provers also permit the user to enter first-order formulas; these provers generally provide various ways of translating such formulas to clause form. Some provers require substantial user guidance, though most such provers have higher-order features, while other provers are designed to be more automatic. For automatic provers, there are often many different flags that can be set to guide the search. For example, typical first-order provers

allow the user to select from among a number of inference strategies for first-order logic as well as strategies for equality. For equality, it may be possible to specify a termination ordering to guide the application of equations. Sometimes the user will select incomplete strategies, hoping that the desired proof will be found faster. It is also often possible to set a size bound so that all clauses or literals larger than a certain size are deleted. Of course one does not know in advance what bound to choose, so some experimentation is necessary. A *sliding priority* approach to setting the size bound automatically was presented in [211]. It is sometimes possible to assign various weights to various symbols or subterms or to variables to guide the proof search. Modern provers generally have term indexing [222] built in to speed up inference, and also have some equality strategy involving ordered paramodulation and rewriting. Many provers are based on resolution, but some are based on model elimination and some are based on propositional approaches. Provers can generate clauses rapidly; for example, Vampire [227] can often generate more than 40,000 clauses per second. Most provers rapidly fill up memory with generated clauses, so that if a proof is not found in a few minutes it will not be found at all. However, equational proofs involve considerable simplification and can sometimes run for a long time without exhausting memory. For example, the Robbins problem ran for 8 days on a SPARC 5 class UNIX computer with a size bound of 70 and required about 30 megabytes of memory, generating 49,548 equations, most of which were deleted by simplification. Sometimes small problems can run for a long time without finding a proof, and sometimes problems with a hundred or more input clauses can result in proofs fairly quickly. Generally, simple problems will be proved by nearly any complete strategy on a modern prover, but hard problems may require fine tuning. For an overview of a list of problems and information about how well various provers perform on them, see the web site at www.tptp.org, and for a sketch of some of the main first-order provers in use today, see http://www.cs.miami.edu/~tptp/CASC/ as well as the journal articles devoted to the individual competitions such as [253, 254]. Current provers often do not have facilities for interacting with other reasoning programs, but work in this area is progressing.

In addition to developing first-order provers, there has been work on other logics, too. The simplest logic typically considered is *propositional logic*, in which there are only predicate symbols (that is, Boolean variables) and logical connectives. Despite its simplicity, propositional logic has surprisingly many applications, such as in hardware verification and constraint satisfaction problems. Propositional provers have even found applications in planning. The general validity (respectively, satisfiability) problem of propositional logic is NP-hard, which means that it does not in all likelihood have an efficient general solution. Nevertheless, there are propositional provers that are surprisingly efficient, and becoming increasingly more so; see Chapter 2 of this Handbook for details.

Binary decision diagrams [43] are a particular form of propositional formulas for which efficient provers exist. BDD's are used in hardware verification, and initiated a tremendous surge of interest by industry in formal verification techniques. Also, the Davis–Putnam–Logemann–Loveland method [69] for propositional logic is heavily used in industry for hardware verification.

Another restricted logic for which efficient provers exist is that of temporal logic, the logic of time (see Chapter 12 of this Handbook). This has applications to con-

currency. The model-checking approach of Clarke and others [48] has proven to be particularly efficient in this area, and has also stimulated considerable interest by industry.

Other logical systems for which provers have been developed are the theory of equational systems, for which term-rewriting techniques lead to remarkably efficient theorem provers, mathematical induction, geometry theorem proving, constraints (Chapter 4 of this Handbook), higher-order logic, and set theory.

Not only proving theorems, but finding counterexamples, or building models, is of increasing importance. This permits one to detect when a theorem is not provable, and thus one need not waste time attempting to find a proof. This is, of course, an activity which human mathematicians often engage in. These counterexamples are typically finite structures. For the so-called *finitely controllable* theories, running a theorem prover and a counterexample (model) finder together yields a decision procedure, which theoretically can have practical applications to such theories. Model finding has recently been extended to larger classes of theories [51].

Among the current applications of theorem provers one can list hardware verification and program verification. For a more detailed survey, see the excellent report by Loveland [164]. Among potential applications of theorem provers are planning problems, the situation calculus, and problems involving knowledge and belief.

There are a number of provers in prominence today, including Otter [177], the provers of Boyer, Moore, and Kaufmann [142, 141], Andrew's matings prover [3], the HOL prover [101], Isabelle [203], Mizar [260], NuPrl [62], PVS [201], and many more. Many of these require substantial human guidance to find proofs. The Omega system [240] is a higher order logic proof development system that attempts to overcome some of the shortcomings of traditional first-order proof systems. In the past it has used a natural deduction calculus to develop proofs with human guidance, though the system is changing.

Provers can be evaluated on a number of grounds. One is *completeness*; can they, in principle, provide a proof of every true theorem? Another evaluation criterion is their performance on specific examples; in this regard, the TPTP problem set [255] is of particular value. Finally, one can attempt to provide an analytic estimate of the efficiency of a theorem prover on classes of problems [212]. This gives a measure which is to a large extent independent of particular problems or machines. The *Handbook of Automated Reasoning* [231] is a good source of information about many areas of theorem proving.

We next discuss resolution for the propositional calculus and then some of the many first-order theorem proving methods, with particular attention to resolution. We also consider techniques for first-order logic with equality. Finally, we briefly discuss some other logics, and corresponding theorem proving techniques.

1.3.1 Resolution in the Propositional Calculus

The main problem for theorem proving purposes is given a formula A, to determine whether it is valid. Since A is valid iff $\neg A$ is unsatisfiable, it is possible to determine validity if one can determine satisfiability. Many theorem provers test satisfiability instead of validity.

The problem of determining whether a Boolean formula A is satisfiable is one of the NP-complete problems. This means that the fastest algorithms known require an

amount of time that is asymptotically exponential in the size of A. Also, it is not likely that faster algorithms will be found, although no one can prove that they do not exist.

Despite this negative result, there is a wide variety of methods in use for testing if a formula is satisfiable. One of the simplest is *truth tables*. For a formula A over $\{P_1, P_2, \ldots, P_n\}$, this involves testing for each of the 2^n valuations I over $\{P_1, P_2, \ldots, P_n\}$ whether $I \models A$. In general, this will require time at least proportional to 2^n to show that A is valid, but may detect satisfiability sooner.

Clause form

Many of the other satisfiability checking algorithms depend on conversion of a formula A to *clause form*. This is defined as follows: An *atom* is a proposition. A *literal* is an atom or an atom preceded by a negation sign. The two literals P and $\neg P$ are said to be *complementary* to each other. A *clause* is a disjunction of literals. A formula is in *clause form* if it is a conjunction of clauses. Thus the formula

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

is in clause form. This is also known as *conjunctive normal form*. We represent clauses by sets of literals and clause form formulas by sets of clauses, so that the above formula would be represented by the following set of sets:

$$\{\{P, \neg R\}, \{\neg P, Q, R\}, \{\neg Q, \neg R\}\}.$$

A *unit clause* is a clause that contains only one literal. The *empty clause* $\{\}$ is understood to represent FALSE.

It is straightforward to show that for every formula A there is an equivalent formula B in clause form. Furthermore, there are well-known algorithms for converting any formula A into such an equivalent formula B. These involve converting all connectives to $\wedge$, $\vee$, and $\neg$, pushing $\neg$ to the bottom, and bringing $\wedge$ to the top. Unfortunately, this process of conversion can take exponential time and can increase the length of the formula by an exponential amount.

The exponential increase in size in converting to clause form can be avoided by adding extra propositions representing subformulas of the given formula. For example, given the formula

$$(P_1 \wedge Q_1) \vee (P_2 \wedge Q_2) \vee (P_3 \wedge Q_3) \vee \cdots \vee (P_n \wedge Q_n)$$

a straightforward conversion to clause form creates 2^n clauses of length n, for a formula of length at least $n2^n$. However, by adding the new propositions R_i which are defined as $P_i \wedge Q_i$, one obtains the new formula

$$(R_1 \vee R_2 \vee \cdots \vee R_n) \wedge ((P_1 \wedge Q_1) \leftrightarrow R_1) \wedge \cdots \wedge ((P_n \wedge Q_n) \leftrightarrow R_n).$$

When this formula is converted to clause form, a much smaller set of clauses results, and the exponential size increase does not occur. The same technique works for any Boolean formula. This transformation is satisfiability preserving but not equivalence preserving, which is enough for theorem proving purposes.

Ground resolution

Many first-order theorem provers are based on resolution, and there is a propositional analogue of resolution called *ground resolution*, which we now present as an introduction to first-order resolution. Although resolution is reasonably efficient for first-order logic, it turns out that ground resolution is generally much less efficient than Davis and Putnam-like procedures for propositional logic [70, 69], often referred to as DPLL procedures because the original Davis and Putnam procedure had some inefficiencies. These DPLL procedures are specialized to clause form and explore the set of possible interpretations of a propositional formula by depth-first search and backtracking with some additional simplification rules for unit clauses.

Ground resolution is a decision procedure for propositional formulas in clause form. If C_1 and C_2 are two clauses, and $L_1 \in C_1$ and $L_2 \in C_2$ are complementary literals, then

$$(C_1 - \{L_1\}) \cup (C_2 - \{L_2\})$$

is called a *resolvent* of C_1 and C_2, where the set difference of two sets A and B is indicated by $A - B$, that is, $\{x\colon\ x \in A, x \notin B\}$. There may be more than one resolvent of two clauses, or maybe none. It is straightforward to show that a resolvent D of two clauses C_1 and C_2 is a logical consequence of $C_1 \wedge C_2$.

For example, if C_1 is $\{\neg P, Q\}$ and C_2 is $\{\neg Q, R\}$, then one can choose L_1 to be Q and L_2 to be $\neg Q$. Then the resolvent is $\{\neg P, R\}$. Note also that R is a resolvent of $\{Q\}$ and $\{\neg Q, R\}$, and $\{\}$ (the empty clause) is a resolvent of $\{Q\}$ and $\{\neg Q\}$.

A *resolution proof* of a clause C from a set S of clauses is a sequence $C_1, C_2, \ldots, C_n$ of clauses in which each C_i is either a member of S or a resolvent of C_j and C_k, for j, k less than i, and C_n is C. Such a proof is called a (resolution) *refutation* if C_n is $\{\}$. Resolution is *complete*:

Theorem 1.3.1. *Suppose S is a set of propositional clauses. Then S is unsatisfiable iff there exists a resolution refutation from S.*

As an example, let S be the set of clauses

$$\{\{P\}, \{\neg P, Q\}, \{\neg Q\}\}.$$

The following is a resolution refutation from S, listing with each resolvent the two clauses that are resolved together:

1.	P	given
2.	$\neg P, Q$	given
3.	$\neg Q$	given
4.	Q	1, 2, resolution
5.	$\{\}$	3, 4, resolution

(Here set braces are omitted, except for the empty clause.) This is a resolution refutation from S, so S is unsatisfiable.

Define $\mathbf{R}(S)$ to be $\bigcup_{C1,C2\in S}$ resolvents$(C1, C2)$. Define $\mathbf{R}^1(S)$ to be $\mathbf{R}(S)$ and $\mathbf{R}^{i+1}(S)$ to be $\mathbf{R}(S \cup \mathbf{R}^i(S))$, for $i > 1$. Typical resolution theorem provers essentially generate all of the resolution proofs from S (with some improvements that will

be discussed later), looking for a proof of the empty clause. Formally, such provers generate $\mathbf{R}^1(S)$, $\mathbf{R}^2(S)$, $\mathbf{R}^3(S)$, and so on, until for some i, $\mathbf{R}^i(S) = \mathbf{R}^{i+1}(S)$, or the empty clause is generated. In the former case, S is satisfiable. If the empty clause is generated, S is unsatisfiable.

Even though DPLL essentially constructs a resolution proof, propositional resolution is much less efficient than DPLL as a decision procedure for satisfiability of formulas in the propositional calculus because the total number of resolutions performed by a propositional resolution prover in the search for a proof is typically much larger than for DPLL. Also, Haken [107] showed that there are unsatisfiable sets S of propositional clauses for which the length of the shortest resolution refutation is exponential in the size (number of clauses) in S. Despite these inefficiencies, we introduced propositional resolution as a way to lead up to first-order resolution, which has significant advantages. In order to extend resolution to first-order logic, it is necessary to add *unification* to it.

1.3.2 First-Order Proof Systems

We now discuss methods for partially deciding validity. These construct proofs of first-order formulas, and a formula is valid iff it can be proven in such a system. Thus there are *complete* proof systems for first-order logic, and Gödel's incompleteness theorem does not apply to first-order logic. Since the set of proofs is countable, one can partially decide validity of a formula A by enumerating the set of proofs, and stopping whenever a proof of A is found. This already gives us a theorem prover, but provers constructed in this way are typically very inefficient.

There are a number of classical proof systems for first-order logic: Hilbert-style systems, Gentzen-style systems, natural deduction systems, semantic tableau systems, and others [87]. Since these generally have not found much application to automated deduction, except for semantic tableau systems, they are not discussed here. Typically they specify inference rules of the form

$$\frac{A_1, A_2, \ldots, A_n}{A}$$

which means that if one has already derived the formulas $A_1, A_2, \ldots, A_n$, then one can also infer A. Using such rules, one builds up a proof as a sequence of formulas, and if a formula B appears in such a sequence, one has proved B.

We now discuss proof systems that have found application to automated deduction. In the following sections, the letters $f, g, h, \ldots$ will be used as *function symbols*, $a, b, c, \ldots$ as *individual constants*, x, y, z and possibly other letters as *individual variables*, and $=$ as the equality symbol. Each function symbol has an *arity*, which is a non-negative integer telling how many arguments it takes. A *term* is either a variable, an individual constant, or an expression of the form $f(t_1, t_2, \ldots, t_n)$ where f is a function symbol of arity n and the t_i are terms. The letters $r, s, t, \ldots$ will denote terms.

Clause form

Many first-order theorem provers convert a first-order formula to *clause form* before attempting to prove it. The beauty of clause form is that it makes the syntax of first-order logic, already quite simple, even simpler. Quantifiers are omitted, and Boolean

connectives as well. One has in the end just sets of sets of literals. It is amazing that the expressive power of first-order logic can be reduced to such a simple form. This simplicity also makes clause form suitable for machine implementation of theorem provers. Not only that, but the validity problem is also simplified in a theoretical sense; one only needs to consider the *Herbrand interpretations*, so the question of validity becomes easier to analyze.

Any first-order formula A can be transformed to a clause form formula B such that A is satisfiable iff B is satisfiable. The translation is not validity preserving. So in order to show that A is valid, one translates $\neg A$ to clause form B and shows that B is unsatisfiable. For convenience, assume that A is a *sentence*, that is, it has no free variables.

The translation of a first-order sentence A to clause form has several steps:

- Push negations in.
- Replace existentially quantified variables by Skolem functions.
- Move universal quantifiers to the front.
- Convert the matrix of the formula to conjunctive normal form.
- Remove universal quantifiers and Boolean connectives.

This transformation will be presented as a set of rewrite rules. A rewrite rule $X \to Y$ means that a subformula of the form X is replaced by a subformula of the form Y.

The following rewrite rules push negations in:

$$(A \leftrightarrow B) \to (A \to B) \wedge (B \to A),$$
$$(A \to B) \to ((\neg A) \vee B),$$
$$\neg\neg A \to A,$$
$$\neg(A \wedge B) \to (\neg A) \vee (\neg B),$$
$$\neg(A \vee B) \to (\neg A) \wedge (\neg B),$$
$$\neg\forall x\, A \to \exists x(\neg A),$$
$$\neg\exists x\, A \to \forall x(\neg A).$$

After negations have been pushed in, we assume for simplicity that variables in the formula are renamed so that each variable appears in only one quantifier. Existential quantifiers are then eliminated by replacing formulas of the form $\exists x A[x]$ by $A[f(x_1, \ldots, x_n)]$, where $x_1, \ldots, x_n$ are all the universally quantified variables whose scope includes the formula A, and f is a new function symbol (that does not already appear in the formula), called a *Skolem function*.

The following rules then move quantifiers to the front:

$$(\forall x A) \vee B \to \forall x(A \vee B),$$
$$B \vee (\forall x A) \to \forall x(B \vee A),$$
$$(\forall x A) \wedge B \to \forall x(A \wedge B),$$
$$B \wedge (\forall x A) \to \forall x(B \wedge A).$$

Next, the matrix is converted to conjunctive normal form by the following rules:

$$(A \vee (B \wedge C)) \rightarrow (A \vee B) \wedge (A \vee C),$$
$$((B \wedge C) \vee A) \rightarrow (B \vee A) \wedge (C \vee A).$$

Finally, universal quantifiers are removed from the front of the formula and a conjunctive normal form formula of the form

$$(A_1 \vee A_2 \vee \cdots \vee A_k) \wedge (B_1 \vee B_2 \vee \cdots \vee B_m) \wedge \cdots \wedge (C_1 \vee C_2 \vee \cdots \vee C_n)$$

is replaced by the set of sets of literals

$$\{\{A_1, A_2, \ldots, A_k\}, \{B_1, B_2, \ldots, B_m\}, \ldots, \{C_1, C_2, \ldots, C_n\}\}.$$

This last formula is the clause form formula which is satisfiable iff the original formula is.

As an example, consider the formula

$$\neg\exists x(P(x) \rightarrow \forall y Q(x, y)).$$

First, negation is pushed past the existential quantifier:

$$\forall x(\neg(P(x) \rightarrow \forall y Q(x, y))).$$

Next, negation is further pushed in, which involves replacing $\rightarrow$ by its definition as follows:

$$\forall x\neg((\neg P(x)) \vee \forall y Q(x, y)).$$

Then $\neg$ is moved in past $\vee$:

$$\forall x((\neg\neg P(x)) \wedge \neg\forall y Q(x, y)).$$

Next the double negation is eliminated and $\neg$ is moved past the quantifier:

$$\forall x(P(x) \wedge \exists y\neg Q(x, y)).$$

Now, negations have been pushed in. Note that no variable appears in more than one quantifier, so it is not necessary to rename variables. Next, the existential quantifier is replaced by a Skolem function:

$$\forall x(P(x) \wedge \neg Q(x, f(x))).$$

There are no quantifiers to move to the front. Eliminating the universal quantifier yields the formula

$$P(x) \wedge \neg Q(x, f(x)).$$

The clause form is then

$$\{\{P(x)\}, \{\neg Q(x, f(x))\}\}.$$

Recall that if B is the clause form of A, then B is satisfiable iff A is. As in propositional calculus, the clause form translation can increase the size of a formula by an exponential amount. This can be avoided as in the propositional calculus by

introducing new predicate symbols for sub-formulas. Suppose A is a formula with sub-formula B, denoted by $A[B]$. Let $x_1, x_2, \ldots, x_n$ be the free variables in B. Let P be a new predicate symbol (that does not appear in A). Then $A[B]$ is transformed to the formula $A[P(x_1, x_2, \ldots, x_n)] \wedge \forall x_1 \forall x_2 \ldots \forall x_n (P(x_1, x_2, \ldots, x_n) \leftrightarrow B)$. Thus the occurrence of B in A is replaced by $P(x_1, x_2, \ldots, x_n)$, and the equivalence of B with $P(x_1, x_2, \ldots, x_n)$ is added on to the formula as well. This transformation can be applied to the new formula in turn, and again as many times as desired. The transformation is satisfiability preserving, which means that the resulting formula is satisfiable iff the original formula A was.

Free variables in a clause are assumed to be universally quantified. Thus the clause $\{\neg P(x), Q(f(x))\}$ represents the formula $\forall x(\neg P(x) \vee Q(f(x)))$. A term, literal, or clause not containing any variables is said to be *ground*.

A set of clauses represents the conjunction of the clauses in the set. Thus the set $\{\{\neg P(x), Q(f(x))\}, \{\neg Q(y), R(g(y))\}, \{P(a)\}, \{\neg R(z)\}\}$ represents the formula $(\forall x(\neg P(x) \vee Q(f(x)))) \wedge (\forall y(\neg Q(y) \vee R(g(y)))) \wedge P(a) \wedge \forall z \neg R(z)$.

Herbrand interpretations

There is a special kind of interpretation that turns out to be significant for mechanical theorem proving. This is called a *Herbrand interpretation*. Herbrand interpretations are defined relative to a set S of clauses. The domain D of a Herbrand interpretation I consists of the set of terms constructed from function and constant symbols of S, with an extra constant symbol added if S has no constant symbols. The constant and function symbols are interpreted so that for any finite term t composed of these symbols, t^I is the term t itself, which is an element of D. Thus if S has a unary function symbol f and a constant symbol c, then $D = \{c, f(c), f(f(c)), f(f(f(c))), \ldots\}$ and c is interpreted so that c^I is the element c of D and f is interpreted so that f^I applied to the term c yields the term $f(c)$, f^I applied to the term $f(c)$ of D yields $f(f(c))$, and so on. Thus these interpretations are quite syntactic in nature. There is no restriction, however, on how a Herbrand interpretation I may interpret the predicate symbols of S.

The interest of Herbrand interpretations for theorem proving comes from the following result:

Theorem 1.3.2. *If S is a set of clauses, then S is satisfiable iff there is a Herbrand interpretation I such that $I \models S$.*

What this theorem means is that for purposes of testing satisfiability of clause sets, one only needs to consider Herbrand interpretations. This implicitly leads to a mechanical theorem proving procedure, which will be presented below. This procedure makes use of *substitutions*.

A *substitution* is a mapping from variables to terms which is the identity on all but finitely many variables. If L is a literal and α is a substitution, then $L\alpha$ is the result of replacing all variables in L by their image under α. The application of substitutions to terms, clauses, and sets of clauses is defined similarly. The expression $\{x_1 \mapsto t_1, x_2 \mapsto t_2, \ldots, x_n \mapsto t_n\}$ denotes the substitution mapping the variable x_i to the term t_i, for $1 \leqslant i \leqslant n$.

For example, $P(x, f(x))\{x \mapsto g(y)\} = P(g(y), f(g(y)))$.

If L is a literal and α is a substitution, then $L\alpha$ is called an *instance* of L. Thus $P(g(y), f(g(y)))$ is an instance of $P(x, f(x))$. Similar terminology applies to clauses and terms.

If S is a set of clauses, then a *Herbrand set* for S is an unsatisfiable set T of ground clauses such that for every clause D in T there is a clause C in S such that D is an instance of C. If there is a Herbrand set for S, then S is unsatisfiable.

For example, let S be the following clause set:

$$\{\{P(a)\}, \{\neg P(x), P(f(x))\}, \{\neg P(f(f(a)))\}\}.$$

For this set of clauses, the following is a Herbrand set:

$$\{\{P(a)\}, \{\neg P(a), P(f(a))\}, \{\neg P(f(a)), P(f(f(a)))\}, \{\neg P(f(f(a)))\}\}.$$

The *ground instantiation problem* is the following: Given a set S of clauses, is there a Herbrand set for S?

The following result is known as Herbrand's theorem, and follows from Theorem 1.3.2:

Theorem 1.3.3. *A set S of clauses is unsatisfiable iff there is a Herbrand set T for S.*

It follows from this result that a set S of clauses is unsatisfiable iff the ground instantiation problem for S is solvable. Thus the problem of first-order validity has been reduced to the ground instantiation problem. This is actually quite an achievement, because the ground instantiation problem deals only with syntactic concepts such as replacing variables by terms, and with propositional unsatisfiability, which is easily understood.

Herbrand's theorem implies the completeness of the following theorem proving method:

Given a set S of clauses, let $C_1, C_2, C_3, \ldots$ be an enumeration of all of the ground instances of clauses in S. This set of ground instances is countable, so it can be enumerated. Consider the following procedure **Prover**:

procedure Prover(S)
  for $i = 1, 2, 3, \ldots$ **do**
    if $\{C_1, C_2, \ldots, C_i\}$ is unsatisfiable **then** return "unsatisfiable" **fi**
  od
end **Prover**

By Herbrand's theorem, it follows that **Prover**(S) will eventually return "unsatisfiable" iff S is unsatisfiable. This is therefore a primitive theorem proving procedure. It is interesting that some of the earliest attempts to mechanize theorem proving [99] were based on this idea. The problem with this approach is that it enumerates many ground instances that could never appear in a proof. However, the efficiency of propositional decision procedures is an attractive feature of this procedure, and it may be possible to modify it to obtain an efficient theorem proving procedure. And in fact, many of the theorem provers in use today are based implicitly on this procedure, and thereby on Herbrand's theorem. The *instance-based* methods such as model evolution [23, 25], clause linking [153], the disconnection calculus [29, 245], and OSHL [213] are

based fairly directly on Herbrand's theorem. These methods attempt to apply DPLL-like approaches [69] to first-order theorem proving. Ganzinger and Korovin [93] also study the properties of instance-based methods and show how redundancy elimination and decidable fragments of first-order logic can be incorporated into them. Korovin has continued this line of research with some later papers.

Unification and resolution

Most mechanical theorem provers today are based on unification, which guides the instantiation of clauses in an attempt to make the procedure **Prover** above more efficient. The idea of unification is to find those instances which are in some sense the most general ones that could appear in a proof. This avoids a lot of work that results from the generation of irrelevant instances by **Prover**.

In the following discussion $\equiv$ will refer to syntactic identity of terms, literals, etc. A substitution α is called a *unifier* of literals L and M if $L\alpha \equiv M\alpha$. If such a substitution exists, L and M are said to be *unifiable*. A substitution α is a *most general unifier* of L and M if for any other unifier β of L and M, there is a substitution γ such that $L\beta \equiv L\alpha\gamma$ and $M\beta \equiv M\alpha\gamma$.

It turns out that if two literals L and M are unifiable, then there is a most general unifier of L and M, and such most general unifiers can be computed efficiently by a number of simple algorithms. The earliest in recent history was given by Robinson [230].

We present a simple unification algorithm on terms which is similar to that presented by Robinson. This algorithm is worst-case exponential time, but often efficient in practice. Algorithms that are more efficient (and even linear time) on large terms have been devised since then [167, 202]. If s and t are two terms and α is a most general unifier of s and t, then $s\alpha$ can be of size exponential in the sizes of s and t, so constructing $s\alpha$ is inherently exponential unless the proper encoding of terms is used; this entails representing repeated subterms only once. However, many symbolic computation systems still use Robinson's original algorithm.

```
procedure Unify(r, s);
  [[ return the most general unifier of terms r and s]]
    if r is a variable then
      if r ≡ s then return { } else
     ( if r occurs in s then return fail else
       return {r ↦ s}) else
    if s is a variable then
     ( if s occurs in r then return fail else
      return {s ↦ r}) else
    if the top-level function symbols of r and s
      differ or have different arities then return fail
    else
      suppose r is f(r_1 ... r_n) and s is f(s_1 ... s_n);
      return(Unify_lists([r_1 ... r_n], [s_1 ... s_n]))
end Unify;
```

procedure Unify_lists($[r_1 \ldots r_n]$, $[s_1 \ldots s_n]$);
  if $[r_1 \ldots r_n]$ is empty **then** return {}
    else
      $\theta \leftarrow$ **Unify**(r_1, t_1);
        if $\theta \equiv$ **fail then** return **fail fi**;
      $\alpha \leftarrow$ **Unify_lists**$([r_2 \ldots r_n]\theta, [s_2 \ldots s_n]\theta)$
        if $\alpha \equiv$ **fail then** return **fail fi**;
  return $\{\theta \circ \alpha\}$
end **Unify_lists**;

For this last procedure, $\theta \circ \alpha$ is defined as the composition of the substitutions θ and α, defined by $t(\theta \circ \alpha) = (t\theta)\alpha$. Note that the composition of two substitutions is a substitution. To extend the above algorithm to literals L and M, return **fail** if L and M have different signs or predicate symbols. Suppose L and M both have the same sign and predicate symbol P. Suppose L and M are $P(r_1, r_2, \ldots, r_n)$ and $P(s_1, s_2, \ldots, s_n)$, respectively, or their negations. Then return **Unify_lists**$([r_1 \ldots r_n], [s_1 \ldots s_n])$ as the most general unifier of L and M.

As examples of unification, a most general unifier of the terms $f(x, a)$ and $f(b, y)$ is $\{x \mapsto b, y \mapsto a\}$. The terms $f(x, g(x))$ and $f(y, y)$ are not unifiable. A most general unifier of $f(x, y, g(y))$ and $f(z, h(z), w)$ is $\{x \mapsto z, y \mapsto h(z), w \mapsto g(h(z))\}$.

One can also define unifiers and most general unifiers of *sets* of terms. A substitution α is said to be a unifier of a set $\{t_1, t_2, \ldots, t_n\}$ of terms if $t_1\alpha \equiv t_2\alpha \equiv t_3\alpha \cdots$. If such a unifier α exists, this set of terms is said to be unifiable. It turns out that if $\{t_1, t_2, \ldots, t_n\}$ is a set of terms and has a unifier, then it has a most general unifier, and this unifier can be computed as **Unify**$(f(t_1, t_2, \ldots, t_n), f(t_2, t_3, \ldots, t_n, t_1))$ where f is a function symbol of arity n. In a similar way, one can define most general unifiers of sets of literals.

Finally, suppose C_1 and C_2 are two clauses and A_1 and A_2 are nonempty subsets of C_1 and C_2, respectively. Suppose for convenience that there are no common variables between C_1 and C_2. Suppose the set $\{L\colon\ L \in A_1\} \cup \{\neg L\colon\ L \in A_2\}$ is unifiable, and let α be its most general unifier. Define the *resolvent* of C_1 and C_2 on the subsets A_1 and A_2 to be the clause

$$(C_1 - A_1)\alpha \cup (C_2 - A_2)\alpha.$$

A resolvent of C_1 and C_2 is defined to be a resolvent of C_1 and C_2 on two such sets A_1 and A_2 of literals. A_1 and A_2 are called *subsets of resolution*. If C_1 and C_2 have common variables, it is assumed that the variables of one of these clauses are renamed before resolving to insure that there are no common variables. There may be more than one resolvent of two clauses, or there may not be any resolvents at all.

Most of the time, A_1 and A_2 consist of single literals. This considerably simplifies the definition, and most of our examples will be of this special case. If $A_1 \equiv \{L\}$ and $A_2 \equiv \{M\}$, then L and M are called *literals of resolution*. We call this kind of resolution *single literal resolution*. Often, one defines resolution in terms of *factoring* and single literal resolution. If C is a clause and θ is a most general unifier of two distinct literals of C, then $C\theta$ is called a *factor* of C. Defining resolution in terms of factoring has some advantages, though it increases the number of clauses one must store.

Here are some examples. Suppose C_1 is $\{P(a)\}$ and C_2 is $\{\neg P(x), Q(f(x))\}$. Then a resolvent of these two clauses on the literals $P(a)$ and $\neg P(x)$ is $\{Q(f(a))\}$. This is because the most general unifier of these two literals is $\{x \mapsto a\}$, and applying this substitution to $\{Q(f(x))\}$ yields the clause $\{Q(f(a))\}$.

Suppose C_1 is $\{\neg P(a, x)\}$ and C_2 is $\{P(y, b)\}$. Then $\{\}$ (the empty clause) is a resolvent of C_1 and C_2 on the literals $\neg P(a, x)$ and $P(y, b)$.

Suppose C_1 is $\{\neg P(x), Q(f(x))\}$ and C_2 is $\{\neg Q(x), R(g(x))\}$. In this case, the variables of C_2 are first renamed before resolving, to eliminate common variables, yielding the clause $\{\neg Q(y), R(g(y))\}$. Then a resolvent of C_1 and C_2 on the literals $Q(f(x))$ and $\neg Q(y)$ is $\{\neg P(x), R(g(f(x)))\}$.

Suppose C_1 is $\{P(x), P(y)\}$ and C_2 is $\{\neg P(z), Q(f(z))\}$. Then a resolvent of C_1 and C_2 on the sets $\{P(x), P(y)\}$ and $\{\neg P(z)\}$ is $\{Q(f(z))\}$.

A *resolution proof* of a clause C from a set S of clauses is a sequence $C_1, C_2, \ldots, C_n$ of clauses in which C_n is C and in which for all i, either C_i is an element of S or there exist integers $j, k < i$ such that C_i is a resolvent of C_j and C_k. Such a proof is called a (resolution) *refutation* from S if C_n is $\{\}$ (the empty clause).

A theorem proving method is said to be *complete* if it is able to prove any valid formula. For unsatisfiability testing, a theorem proving method is said to be complete if it can derive **false**, or the empty clause, from any unsatisfiable set of clauses. It is known that resolution is complete:

Theorem 1.3.4. *A set S of first-order clauses is unsatisfiable iff there is a resolution refutation from S.*

Therefore one can use resolution to test unsatisfiability of clause sets, and hence validity of first-order formulas. The advantage of resolution over the **Prover** procedure above is that resolution uses unification to choose instances of the clauses that are more likely to appear in a proof. So in order to show that a first-order formula A is valid, one can do the following:

- Convert $\neg A$ to clause form S.
- Search for a proof of the empty clause from S.

As an example of this procedure, resolution can be applied to show that the first-order formula

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

is valid. Here $\rightarrow$ represents logical implication, as usual. In the refutational approach, one negates this formula to obtain

$$\neg[\forall x \exists y(P(x) \rightarrow Q(x, y)) \wedge \forall x \forall y \exists z(Q(x, y) \rightarrow R(x, z)) \rightarrow \forall x \exists z(P(x) \rightarrow R(x, z))],$$

and shows that this formula is unsatisfiable. The procedure of Section 1.3.3 for translating formulas into clause form yields the following set S of clauses:

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

The following is then a resolution refutation from this clause set:

1.	$P(a)$	(input)
2.	$\neg P(x), Q(x, f(x))$	(input)
3.	$Q(a, f(a))$	(resolution, 1, 2)
4.	$\neg Q(x, y), R(x, g(x, y))$	(input)
5.	$R(a, g(a, f(a)))$	(3, 4, resolution)
6.	$\neg R(a, z)$	(input)
7.	false	(5, 6, resolution)

The designation "input" means that a clause is in S. Since **false** (the empty clause) has been derived from S by resolution, it follows that S is unsatisfiable, and so the original first-order formula is valid.

Even though resolution is much more efficient than the **Prover** procedure, it is still not as efficient as one would like. In the early days of resolution, a number of refinements were added to resolution, mostly by the Argonne group, to make it more efficient. These were the set of support strategy, unit preference, hyper-resolution, subsumption and tautology deletion, and demodulation. In addition, the Argonne group preferred using small clauses when searching for resolution proofs. Also, they employed some very efficient data structures for storing and accessing clauses. We will describe most of these refinements now.

A clause C is called a *tautology* if for some literal L, $L \in C$ and $\neg L \in C$. It is known that if S is unsatisfiable, there is a refutation from S that does not contain any tautologies. This means that tautologies can be deleted as soon as they are generated and need never be included in resolution proofs.

In general, given a set S of clauses, one searches for a refutation from S by performing a sequence of resolutions. To ensure completeness, this search should be *fair*, that is, if clauses C_1 and C_2 have been generated already, and it is possible to resolve these clauses, then this resolution must eventually be done. However, the order in which resolutions are performed is nonetheless very flexible, and a good choice in this respect can help the prover a lot. One good idea is to prefer resolutions of clauses that are small, that is, that have small terms in them.

Another way to guide the choice of resolutions is based on subsumption, as follows: Clause C is said to *subsume* clause D if there is a substitution Θ such that $C\Theta \subseteq D$. For example, the clause $\{Q(x)\}$ subsumes the clause $\{\neg P(a), Q(a)\}$. C is said to *properly subsume* D if C subsumes D and the number of literals in C is less than or equal to the number of literals in D. For example, the clause $\{Q(x), Q(y)\}$ subsumes $\{Q(a)\}$, but does not properly subsume it. It is known that clauses properly subsumed by other clauses can be deleted when searching for resolution refutations from S. It is possible that these deleted clauses may still appear in the final refutation, but once a clause C is generated that properly subsumes D, it is never necessary to use D in any further resolutions. Subsumption deletion can reduce the proof time tremendously, since long clauses tend to be subsumed by short ones. Of course, if two clauses properly subsume each other, one of them should be kept. The use of appropriate data structures [222, 226] can greatly speed up the subsumption test, and indeed term indexing data structures are essential for an efficient theorem prover, both for quickly finding clauses to resolve and for performing the subsumption test. As an example [222], in a run of the Vampire prover on the problem LCL-129-1.p from the

TPTP library of www.tptp.org, in 270 seconds 8,272,207 clauses were generated of which 5,203,928 were deleted because their weights were too large, 3,060,226 were deleted because they were subsumed by existing clauses (*forward subsumption*), and only 8053 clauses were retained.

This can all be combined to obtain a program for searching for resolution proofs from S, as follows:

procedure Resolver(S)
  $R \leftarrow S$;
    while false $\notin R$ **do**
      choose clauses $C_1, C_2 \in R$ fairly, preferring small clauses;
      if no new pairs C_1, C_2 exist **then** return "satisfiable" **fi**;
      $R' \leftarrow \{D\colon\ D$ is a resolvent of C_1, C_2 and D is not a tautology$\}$;
      for $D \in R'$ **do**
        if no clause in R properly subsumes D
        then $R \leftarrow \{D\} \cup \{C \in R\colon\ D$ does not properly subsume $C\}$ **fi**;
      od
    od
end **Resolver**

In order to make precise what a "small clause" is, one defines $\|C\|$, the *symbol size* of clause C, as follows:

$$\begin{aligned}
\|x\| &= 1 \quad \text{for variables } x\\
\|c\| &= 1 \quad \text{for constant symbols } c\\
\|f(t_1,\ldots,t_n)\| &= 1 + \|t_1\| + \cdots + \|t_n\| \quad \text{for terms } f(t_1,\ldots,t_n)\\
\|P(t_1,\ldots,t_n)\| &= 1 + \|t_1\| + \cdots + \|t_n\| \quad \text{for atoms } P(t_1,\ldots,t_n)\\
\|\neg A\| &= \|A\| \quad \text{for atoms } A\\
\|\{L_1, L_2, \ldots, L_n\}\| &= \|L_1\| + \cdots + \|L_n\| \quad \text{for clauses } \{L_1, L_2, \ldots, L_n\}
\end{aligned}$$

Small clauses, then, are those having a small symbol size.

Another technique used by the Argonne group is the *unit preference strategy*, defined as follows: A *unit clause* is a clause that contains exactly one literal. A *unit resolution* is a resolution of clauses C_1 and C_2, where at least one of C_1 and C_2 is a unit clause. The *unit preference* strategy prefers unit resolutions, when searching for proofs. Unit preference has to be modified to permit non-unit resolutions to guarantee completeness. Thus non-unit resolutions are also performed, but not as early. The unit preference strategy helps because unit resolutions reduce the number of literals in a clause.

Refinements of resolution

In an attempt to make resolution more efficient, many, many refinements were developed in the early days of theorem proving. We present a few of them, and mention a number of others. For a discussion of resolution and its refinements, and theorem proving in general, see [53, 163, 45, 271, 87, 155]. It is hard to know which refinements will help on any given example, but experience with a theorem prover can help to give one a better idea of which refinements to try. In general, none of these refinements help very much most of the time.

A literal is called *positive* if it is an atom, that is, has no negation sign. A literal with a negation sign is called *negative*. A clause C is called *positive* if all of the literals in C are positive. C is called *negative* if all of the literals in C are negative. A resolution of C_1 and C_2 is called positive if one of C_1 and C_2 is a positive clause. It is called negative if one of C_1 and C_2 is a negative clause. It turns out that positive resolution is complete, that is, if S is unsatisfiable, then there is a refutation from S in which all of the resolutions are positive. This refinement of resolution is known as P_1 deduction in the literature. Similarly, negative resolution is complete. Hyper-resolution is essentially a modification of positive resolution in which a series of positive resolvents is done all at once. To be precise, suppose that C is a clause having at least one negative literal and $D_1, D_2, \ldots, D_n$ are positive clauses. Suppose C_1 is a resolvent of C and D_1, C_2 is a resolvent of C_1 and $D_2, \ldots,$ and C_n is a resolvent of C_{n-1} and D_n. Suppose that C_n is a positive clause but none of the clauses C_i are positive, for $i < n$. Then C_n is called a *hyper-resolvent* of C and $D_1, D_2, \ldots, D_n$. Thus the inference steps in hyper-resolution are sequences of positive resolutions. In the hyper-resolution strategy, the inference engine looks for a complete collection $D_1 \ldots D_n$ of clauses to resolve with C and only performs the inference when the entire hyper-resolution can be carried out. Hyper-resolution is sometimes useful because it reduces the number of intermediate results that must be stored in the prover.

Typically, when proving a theorem, there is a general set A of axioms and a particular formula F that one wishes to prove. So one wishes to show that the formula $A \rightarrow F$ is valid. In the refutational approach, this is done by showing that $\neg(A \rightarrow F)$ is unsatisfiable. Now, $\neg(A \rightarrow F)$ is transformed to $A \wedge \neg F$ in the clause form translation. One then obtains a set S_A of clauses from A and a set S_F of clauses from $\neg F$. The set $S_A \cup S_F$ is unsatisfiable iff $A \rightarrow F$ is valid. One typically tries to show $S_A \cup S_F$ unsatisfiable by performing resolutions. Since one is attempting to prove F, one would expect that resolutions involving the clauses S_F are more likely to be useful, since resolutions involving two clauses from S_A are essentially combining general axioms. Thus one would like to only perform resolutions involving clauses in S_F or clauses derived from them. This can be achieved by the *set of support* strategy, if the set S_F is properly chosen.

The set of support strategy restricts all resolutions to involve a clause in the *set of support* or a clause derived from it. To guarantee completeness, the set of support must be chosen to include the set of clauses C of S such that $I \not\models C$ for some interpretation I. Sets A of axioms typically have standard models I, so that $I \models A$. Since translation to clause form is satisfiability preserving, $I' \models S_A$ as well, where I' is obtained from I by a suitable interpretation of Skolem functions. If the set of support is chosen as the clauses not satisfied by I', then this set of support will be a subset of the set S_F above and inferences are restricted to those that are relevant to the particular theorem. Of course, it is not necessary to test if $I \models C$ for clauses C; if one knows that A is satisfiable, one can choose S_F as the set of support.

The *semantic resolution* strategy is like the set-of-support resolution, but requires that when two clauses C_1 and C_2 resolve, at least one of them must not be satisfied by a specified interpretation I. Some interpretations permit the test $I \models C$ to be carried out; this is possible, for example, if I has a finite domain. Using such a semantic definition of the set of support strategy further restricts the set of possible resolutions over the set of support strategy while retaining completeness.

Other refinements of resolution include ordered resolution, which orders the literals of a clause, and requires that the subsets of resolution include a maximal literal in their respective clauses. Unit resolution requires all resolutions to be unit resolutions, and is not complete. Input resolution requires all resolutions to involve a clause from S, and this is not complete, either. Unit resulting (UR) resolution is like unit resolution, but has larger inference steps. This is also not complete, but works well surprisingly often. Locking resolution attaches indices to literals, and uses these to order the literals in a clause and decide which literals have to belong to the subsets of resolution. Ancestry-filter form resolution imposes a kind of linear format on resolution proofs. These strategies are both complete. Semantic resolution is compatible with some ordering refinements, that is, the two strategies together are still complete.

It is interesting that resolution is complete for *logical consequences*, in the following sense: If S is a set of clauses, and C is a clause such that $S \models C$, that is, C is a logical consequence of S, then there is a clause D derivable by resolution such that D subsumes C.

Another resolution refinement that is useful sometimes is *splitting*. If C is a clause and $C \equiv C_1 \cup C_2$, where C_1 and C_2 have no common variables, then $S \cup \{C\}$ is unsatisfiable iff $S \cup \{C_1\}$ is unsatisfiable and $S \cup \{C_2\}$ is unsatisfiable. The effect of this is to reduce the problem of testing unsatisfiability of $S \cup \{C\}$ to two simpler problems. A typical example of such a clause C is a ground clause with two or more literals.

There is a special class of clauses called *Horn clauses* for which specialized theorem proving strategies are complete. A *Horn clause* is a clause that has at most one positive literal. Such clauses have found tremendous application in logic programming languages. If S is a set of Horn clauses, then unit resolution is complete, as is input resolution.

Other strategies

There are a number of other strategies which apply to sets S of clauses, but do not use resolution. One of the most notable is *model elimination* [162], which constructs *chains* of literals and has some similarities to the DPLL procedure. Model elimination also specifies the order in which literals of a clause will "resolve away". There are also a number of *connection methods* [28, 158], which operate by constructing links between complementary literals in different clauses, and creating structures containing more than one clause linked together. In addition, there are a number of *instance-based* strategies, which create a set T of ground instances of S and test T for unsatisfiability using a DPLL-like procedure. Such instance-based methods can be much more efficient than resolution on certain kinds of clause sets, namely, those that are highly non-Horn but do not involve deep term structure.

Furthermore, there are a number of strategies that do not use clause form at all. These include the semantic tableau methods, which work backwards from a formula and construct a tree of possibilities; Andrews' matings method, which is suitable for higher order logic and has obtained some impressive proofs automatically; natural deduction methods; and sequent style systems. Tableau systems have found substantial application in automated deduction, and many of these are even adapted to formulas in clause form; for a survey see [106].

Evaluating strategies

In general, we feel that qualities that need to be considered when evaluating a strategy are not only *completeness* but also *propositional efficiency*, *goal-sensitivity* and *use of semantics*. By propositional efficiency is meant the degree to which the efficiency of the method on propositional problems compares with DPLL; most strategies do poorly in this respect. By goal-sensitivity is meant the degree to which the method permits one to concentrate on inferences related to the particular clauses coming from the negation of the theorem (the set S_F discussed above). When there are many, many input clauses, goal sensitivity is crucial. By use of semantics is meant whether the method can take advantage of natural semantics that may be provided with the problem statement in its search for a proof. An early prover that did use semantics in this way was the geometry prover of Gelernter et al. [94]. Note that model elimination and set of support strategies are goal-sensitive but apparently not propositionally efficient. Semantic resolution is goal-sensitive and can use natural semantics, but is not propositionally efficient, either. Some instance-based strategies are goal-sensitive and use natural semantics and are propositionally efficient, but may have to resort to exhaustive enumeration of ground terms instead of unification in order to instantiate clauses. A further issue is to what extent various methods permit the incorporation of efficient equality techniques, which varies a lot from method to method. Therefore there are some interesting problems involved in combining as many of these desirable features as possible. And for strategies involving extensive human interaction, the criteria for evaluation are considerably different.

1.3.3 Equality

When proving theorems involving equations, one obtains many irrelevant terms. For example, if one has the equations $x + 0 = x$ and $x * 1 = x$, and addition and multiplication are commutative and associative, then one obtains many terms identical to x, such as $1 * x * 1 * 1 + 0$. For products of two or three variables or constants, the situation becomes much worse. It is imperative to find a way to get rid of all of these equivalent terms. For this purpose, specialized methods have been developed to handle equality.

As examples of mathematical structures where such equations arise, for groups and monoids the group operation is associative with an identity, and for abelian groups the group operation is associative and commutative. Rings and fields also have an associative and commutative addition operator with an identity and another multiplication operator that is typically associative. For Boolean algebras, the multiplication operation is also idempotent. For example, set union and intersection are associative, commutative, and idempotent. Lattices have similar properties. Such equations and structures typically arise when axiomatizing integers, reals, complex numbers, matrices, and other mathematical objects.

The most straightforward method of handling equality is to use a general first-order resolution theorem prover together with the *equality axioms*, which are the following (assuming free variables are implicitly universally quantified):

$x = x$,
$x = y \rightarrow y = x$,
$x = y \wedge y = z \rightarrow x = z$,
$x_1 = y_1 \wedge x_2 = y_2 \wedge \cdots \wedge x_n = y_n \rightarrow f(x_1 \ldots x_n) = f(y_1 \ldots y_n)$
for all function symbols f,
$x_1 = y_1 \wedge x_2 = y_2 \wedge \cdots \wedge x_n = y_n \wedge P(x_1 \ldots x_n) \rightarrow P(y_1 \ldots y_n)$
for all predicate symbols P

Let **Eq** refer to this set of equality axioms. The approach of using **Eq** explicitly leads to many inefficiencies, as noted above, although in some cases it works reasonably well.

Another approach to equality is the *modification method* of Brand [40, 19]. In this approach, a set S of clauses is transformed into another set S' with the following property: $S \cup \mathbf{Eq}$ is unsatisfiable iff $S' \cup \{x = x\}$ is unsatisfiable. Thus this transformation avoids the need for the equality axioms, except for $\{x = x\}$. This approach often works a little better than using **Eq** explicitly.

Contexts

In order to discuss other inference rules for equality, some terminology is needed. A *context* is a term with occurrences of $\Box$ in it. For example, $f(\Box, g(a, \Box))$ is a context. A $\Box$ by itself is also a context. One can also have literals and clauses with $\Box$ in them, and they are also called contexts. If n is an integer, then an *n-context* is a term with n occurrences of $\Box$. If t is an n-context and $m \leqslant n$, then $t[t_1, \ldots, t_m]$ represents t with the leftmost m occurrences of $\Box$ replaced by the terms $t_1, \ldots, t_m$, respectively. Thus, for example, $f(\Box, b, \Box)$ is a 2-context, and $f(\Box, b, \Box)[g(c)]$ is $f(g(c), b, \Box)$. Also, $f(\Box, b, \Box)[g(c)][a]$ is $f(g(c), b, a)$. In general, if r is an n-context and $m \leqslant n$ and the terms s_i are 0-contexts, then $r[s_1, \ldots, s_n] \equiv r[s_1][s_2] \ldots [s_n]$. However, $f(\Box, b, \Box)[g(\Box)]$ is $f(g(\Box), b, \Box)$, so $f(\Box, b, \Box)[g(\Box)][a]$ is $f(g(a), b, \Box)$. In general, if r is a k-context for $k \geqslant 1$ and s is an n-context for $n \geqslant 1$, then $r[s][t] \equiv r[s[t]]$, by a simple argument (both replace the leftmost $\Box$ in $r[s]$ by t).

Termination orderings on terms

It is necessary to discuss partial orderings on terms in order to explain inference rules for equality. Partial orderings give a precise definition of the complexity of a term, so that $s > t$ means that the term s is more complex than t in some sense, and replacing s by t makes a clause simpler. A partial ordering $>$ is *well-founded* if there are no infinite sequences x_i of elements such that $x_i > x_{i+1}$ for all $i \geqslant 0$. A *termination ordering* on terms is a partial ordering $>$ which is well founded and satisfies the *full invariance property*, that is, if $s > t$ and Θ is a substitution then $s\Theta > t\Theta$, and also satisfies the *replacement property*, that is, $s > t$ implies $r[s] > r[t]$ for all 1-contexts r.

Note that if $s > t$ and $>$ is a termination ordering, then all variables in t appear also in s. For example, if $f(x) > g(x, y)$, then by full invariance $f(x) > g(x, f(x))$, and by replacement $g(x, f(x)) > g(x, g(x, f(x)))$, etc., giving an infinite descending sequence of terms.

The concept of a *multiset* is often useful to show termination. Informally, a multiset is a set in which an element can occur more than once. Formally, a multiset S is

a function from some underlying domain D to the non-negative integers. It is said to be finite if $\{x\colon\ S(x) > 0\}$ is finite. One writes $x \in S$ if $S(x) > 0$. $S(x)$ is called the *multiplicity* of x in S; this represents the number of times x appears in S. If S and T are multisets then $S \cup T$ is defined by $(S \cup T)(x) = S(x) + T(x)$ for all x. A partial ordering $>$ on D can be extended to a partial ordering $\gg$ on multisets in the following way: One writes $S \gg T$ if there is some multiset V such that $S = S' \cup V$ and $T = T' \cup V$ and S' is nonempty and for all t in T' there is an s in S' such that $s > t$. This relation can be computed reasonably fast by deleting common elements from S and T as long as possible, then testing if the specified relation between S' and T' holds. The idea is that a multiset becomes smaller if an element is replaced by any number of smaller elements. Thus $\{3, 4, 4\} \gg \{2, 2, 2, 2, 1, 4, 4\}$ since 3 has been replaced by 2, 2, 2, 2, 1. This operation can be repeated any number of times, still yielding a smaller multiset; in fact, the relation $\gg$ can be defined in this way as the smallest transitive relation having this property [75]. One can show that if $>$ is well founded, so is $\gg$. For a comparison with other definitions of multiset ordering, see [131].

We now give some examples of termination orderings. The simplest kind of termination orderings are those that are based on size. Recall that $\|s\|$ is the symbol size (number of symbol occurrences) of a term s. One can then define $>$ so that $s > t$ if for all Θ making $s\Theta$ and $t\Theta$ ground terms, $\|s\Theta\| > \|t\Theta\|$. For example, $f(x, y) > g(y)$ in this ordering, but it is not true that $h(x, a, b) > f(x, x)$ because x could be replaced by a large term. This termination ordering is computable; $s > t$ iff $\|s\| > \|t\|$ and no variable occurs more times in t than s.

More powerful techniques are needed to get some more interesting termination orderings. One of the most remarkable results in this area is a theorem of Dershowitz [75] about simplification orderings, that gives a general technique for showing that an ordering is a termination ordering. Before his theorem, each ordering had to be shown well founded separately, and this was often difficult. This theorem makes use of simplification orderings.

Definition 1.3.5. *A partial ordering $>$ on terms is a* simplification ordering *if it satisfies the replacement property, that is, for* 1*-contexts r, $s > t$ implies $r[s] > r[t]$, and has the* subterm property, *that is, $s > t$ if t is a proper subterm of s. Also, if there are function symbols f with variable arity, it is required that $f(\ldots s \ldots) > f(\ldots\ldots)$ for all such f.*

Theorem 1.3.6. *All simplification orderings are well founded.*

Proof. Based on Kruskal's tree theorem [148], which says that in any infinite sequence $t_1, t_2, t_3, \ldots$ of terms, there are natural numbers i and j with $i < j$ such that t_i is embedded in t_j in a certain sense. It turns out that if t_i is embedded in t_j then $t_j \geqslant t_i$ for any simplification ordering $>$. □

The *recursive path ordering* is one of the simplest simplification orderings. This ordering is defined in terms of a *precedence* ordering on function symbols, which is a partial ordering on the function symbols. One writes $f < g$ to indicate that f is less than g in the precedence relation on function symbols. The recursive path ordering will

be presented as a complete set of inference rules that may be used to construct proofs of $s > t$. That is, if $s > t$ then there is a proof of this in the system. Also, by using the inference rules backwards in a goal-directed manner, it is possible to construct a reasonably efficient decision procedure for statements of the form $s > t$. Recall that if $>$ is an ordering, then $\gg$ is the extension of this ordering to multisets. The ordering we present is somewhat weaker than that usually given in the literature.

$$\frac{f = g \qquad \{s_1 \ldots s_m\} \gg \{t_1 \ldots t_n\}}{f(s_1 \ldots s_m) > g(t_1 \ldots t_n)}$$

$$\frac{s_i \geqslant t}{f(s_1 \ldots s_m) > t}$$

$$\frac{true}{s \geqslant s}$$

$$\frac{f > g \qquad f(s_1 \ldots s_m) > t_i \text{ all } i}{f(s_1 \ldots s_m) > g(t_1 \ldots t_n)}$$

For example, suppose $* > +$. Then one can show that $x * (y + z) > x * y + x * z$ as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{true}{y \geqslant y}}{y + z > y}}{\{x, y + z\} \gg \{x, y\}}}{x * (y + z) > x * y} \quad \dfrac{\dfrac{\dfrac{\dfrac{true}{y \geqslant y}}{y + z > z}}{\{x, y + z\} \gg \{x, z\}}}{x * (y + z) > x * z} \quad * > +}{x * (y + z) > x * y + x * z}$$

For some purposes, it is necessary to modify this ordering so that subterms are considered lexicographically. In general, if $>$ is an ordering, then the lexicographic extension $>_{lex}$ of $>$ to tuples is defined as follows:

$$\frac{s_1 > t_1}{(s_1 \ldots s_m) >_{lex} (t_1 \ldots t_n)}$$

$$\frac{s_1 = t_1 \qquad (s_2 \ldots s_m) >_{lex} (t_2 \ldots t_n)}{(s_1 \ldots s_m) >_{lex} (t_1 \ldots t_n)}$$

$$\frac{true}{(s_1 \ldots s_m) >_{lex} ()}$$

One can show that if $>$ is well founded, then so is its extension $>_{lex}$ to bounded length tuples. This lexicographic treatment of subterms is the idea of the lexicographic path ordering of Kamin and Levy [136]. This ordering is defined by the following inference rules:

$$\frac{f = g \qquad (s_1 \ldots s_m) >_{lex} (t_1 \ldots t_n) \qquad f(s_1 \ldots s_m) > t_j, \text{ all } j \geqslant 2}{f(s_1 \ldots s_m) > g(t_1 \ldots t_n)}$$

$$\frac{s_i \geqslant t}{f(s_1 \ldots s_m) > t}$$

$$\frac{\dfrac{}{s \geqslant s}\ true \qquad f > g \qquad f(s_1 \ldots s_m) > t_i \text{ all } i}{f(s_1 \ldots s_m) > g(t_1 \ldots t_n)}$$

In the first inference rule, it is not necessary to test $f(s_1 \ldots s_m) > t_1$ since $(s_1 \ldots s_m) >_{lex} (t_1 \ldots t_n)$ implies $s_1 \geqslant t_1$ hence $f(s_1 \ldots s_m) > t_1$. One can show that this ordering is a simplification ordering for systems having fixed arity function symbols. This ordering has the useful property that $f(f(x, y), z) >_{lex} f(x, f(y, z))$; informally, the reason for this is that the terms have the same size, but the first subterm $f(x, y)$ of $f(f(x, y), z)$ is always larger than the first subterm x of $f(x, f(y, z))$.

The first orderings that could be classified as recursive path orderings were those of Plaisted [208, 207]. A large number of other similar orderings have been developed since the ones mentioned above, for example the *dependency pair* method [7] and its recent automatic versions [120, 98].

Paramodulation

Above, we saw that the equality axioms **Eq** can be used to prove theorems involving equality, and that Brand's modification method is another approach that avoids the need for the equality axioms. A better approach in most cases is to use the *paramodulation rule* [228, 193] defined as follows:

$$\frac{C[t], r = s \vee D, r \text{ and } t \text{ are unifiable, } t \text{ is not a variable, } \mathbf{Unify}(r, t) = \theta}{C\theta[s\theta] \vee D\theta}$$

Here $C[t]$ is a clause containing a subterm t, C is a context, and t is a non-variable term. Also, $C\theta[s\theta]$ is the clause $(C[t])\theta$ with $s\theta$ replacing the specified occurrence of $t\theta$. Also, $r = s \vee D$ is another clause having a literal $r = s$ whose predicate is equality and remaining literals D, which can be empty. To understand this rule, consider that $r\theta = s\theta$ is an instance of $r = s$, and $r\theta$ and $t\theta$ are identical. If $D\theta$ is false, then $r\theta = s\theta$ must be true, so it is possible to replace $r\theta$ in $(C[t])\theta$ by $s\theta$ if $D\theta$ is false. Thus $C\theta[s\theta] \vee D\theta$ is inferred. It is assumed as usual that variables in $C[t]$ or in $r = s \vee D$ are renamed if necessary to insure that these clauses have no common variables before performing paramodulation. The clause $C[t]$ is said to be paramodulated *into*. It is also possible to paramodulate in the other direction, that is, the equation $r = s$ can be used in either direction.

For example, the clause $P(g(a)) \vee Q(b)$ is a paramodulant of $P(f(x))$ and $(f(a) = g(a)) \vee Q(b)$. Brand [40] showed that if **Eq** is the set of equality axioms given above and S is a set of clauses, then $S \cup \mathbf{Eq}$ is unsatisfiable iff there is a proof of the empty clause from $S \cup \{x = x\}$ using resolution and paramodulation as inference rules. Thus, paramodulation allows us to dispense with all the equality axioms except $x = x$.

Some more recent proofs of the completeness of resolution and paramodulation [125] show the completeness of restricted versions of paramodulation which considerably reduce the search space. In particular, it is possible to restrict this rule so that it is not performed if $s\theta > r\theta$, where $>$ is a termination ordering fixed in advance. So if one has an equation $r = s$, and $r > s$, then this equation can only be used to replace instances of r by instances of s. If $s > r$, then this equation can only be used

in the reverse direction. The effect of this is to constrain paramodulation so that "big" terms are replaced by "smaller" ones, considerably improving its efficiency. It would be a disaster to allow paramodulation to replace x by $x * 1$, for example. Another complete refinement of ordered paramodulation is that paramodulation only needs to be done into the "large" side of an equation. If the subterm t of $C[t]$ occurs in an equation $u = v$ or $v = u$ of $C[t]$, and $u > v$, where $>$ is the termination ordering being used, then the paramodulation need not be done if the specified occurrence of t is in v. Some early versions of paramodulation required the use of the functionally reflexive axioms of the form $f(x_1, \ldots, x_n) = f(x_1, \ldots, x_n)$, but this is now known not to be necessary. When D is empty, paramodulation is similar to "narrowing", which has been much studied in the context of logic programming and term rewriting. Recently, a more refined approach to the completeness proof of resolution and paramodulation has been found [16, 17] which permits greater control over the equality strategy. This approach also permits one to devise resolution strategies that have a greater control over the order in which literals are resolved away.

Demodulation

Similar to paramodulation is the rewriting or "demodulation" rule, which is essentially a method of simplification.

$$\frac{C[t], r = s, r\theta \equiv t, r\theta > s\theta}{C[s\theta]}.$$

Here $C[t]$ is a clause (so C is a 1-context) containing a non-variable term t, $r = s$ is a unit clause, and $>$ is the termination ordering that is fixed in advance. It is assumed that variables are renamed so that $C[t]$ and $r = s$ have no common variables before this rule is applied. The clause $C[s\theta]$ is called a *demodulant* of $C[t]$ and $r = s$. Similarly, $C[s\theta]$ is a demodulant of $C[t]$ and $s = r$, if $r\theta > s\theta$. Thus an equation can be used in either direction, if the ordering condition is satisfied.

As an example, given the equation $x * 1 = x$ and assuming $x * 1 > x$ and given a clause $C[f(a) * 1]$ having a subterm of the form $f(a) * 1$, this clause can be simplified to $C[f(a)]$, replacing the occurrence of $f(a) * 1$ in C by $f(a)$.

To justify the demodulation rule, the instance $r\theta = s\theta$ of the equation $r = s$ can be inferred because free variables are implicitly universally quantified. This makes it possible to replace $r\theta$ in C by $s\theta$, and vice versa. But $r\theta$ is t, so t can be replaced by $s\theta$.

Not only is the demodulant $C[s\theta]$ inferred, but the original clause $C[t]$ is typically deleted. Thus, in contrast to resolution and paramodulation, demodulation replaces clauses by simpler clauses. This can be a considerable aid in reducing the number of generated clauses. This also makes mechanical theorem proving closer to human reasoning.

The reason for specifying that $s\theta$ is simpler than $r\theta$ is not only the intuitive desire to simplify clauses, but also to ensure that demodulation terminates. For example, there is no termination ordering in which $x * y > y * x$, since then the clause $a * b = c$ could demodulate using the equation $x * y = y * x$ to $b * a = c$ and then to $a * b = c$ and so on indefinitely. Such an ordering $>$ could not be a termination ordering, since it

violates the well-foundedness condition. However, for many termination orderings $>$, $x * 1 > x$, and thus the clauses $P(x * 1)$ and $x * 1 = x$ have $P(x)$ as a demodulant if some such ordering is being used.

Resolution with ordered paramodulation and demodulation is still complete if paramodulation and demodulation are done with respect to the same simplification ordering during the proof process [125]. Demodulation is essential in practice, for without it one can generate expressions like $x * 1 * 1 * 1$ that clutter up the search space. Some complete refinements of paramodulation also restrict which literals can be paramodulated into, which must be the "largest" literals in the clause in a sense. Such refinements are typically used with resolution refinements that also restrict subsets of resolution to contain "large" literals in a clause. Another recent development is *basic paramodulation*, which restricts the positions in a term into which paramodulation can be done [18, 194]; this refinement was used in McCune's proof of the Robbins problem [176].

1.3.4 Term Rewriting Systems

A beautiful theory of *term-rewriting systems* has been developed to handle proofs involving *equational systems*; these are theorems of the form $E \models e$ where E is a collection of equations and e is an equation. For such systems, term-rewriting techniques often lead to very efficient proofs. The Robbins problem was of this form, for example.

An *equational system* is a set of equations. Often one is interested in knowing if an equation follows logically from the given set. For example, given the equations $x + y = y + x$, $(x + y) + z = x + (y + z)$, and $-(-(x + y) + -(x + -y)) = x$, one might want to know if the equation $-(-x + y) + -(-x + -y) = x$ is a logical consequence. As another example, one might want to know whether $x * y = y * x$ in a group in which $x^2 = e$ for all x. Such systems are of interest in theorem proving, programming languages, and other areas. Common data structures like lists and stacks can often be described by such sets of equations. In addition, a functional program is essentially a set of equations, typically with higher order functions, and the execution of a program is then a kind of equational reasoning. In fact, some programming languages based on term rewriting have been implemented, and can execute several tens of millions of rewrites per second [72]. Another language based on rewriting is MAUDE [119]. Rewriting techniques have also been used to detect flaws in security protocols and prove properties of such protocols [129]. Systems for mechanising such proofs on a computer are becoming more and more powerful. The Waldmeister system [92] is particularly effective for proofs involving equations and rewriting. The area of rewriting was largely originated by the work of Knuth and Bendix [144]. For a discussion of term-rewriting techniques, see [76, 11, 77, 199, 256].

Syntax of equational systems

A term u is said to be a *subterm* of t if u is t or if t is $f(t_1, \ldots, t_n)$ and u is a subterm of t_i for some i. An *equation* is an expression of the form $s = t$ where s and t are terms. An *equational system* is a set of equations. We will generally consider only unsorted equational systems, for simplicity The letter E will be used to refer to equational systems.

We give a set of inference rules for deriving consequences of equations.

$$\frac{t = u}{t\theta = u\theta}$$

$$\frac{t = u}{u = t}$$

$$\frac{t = u}{f(\ldots t \ldots) = f(\ldots u \ldots)}$$

$$\frac{t = u \qquad u = v}{t = v}$$

$$\frac{true}{t = t}$$

The following result is due to Birkhoff [30]:

Theorem 1.3.7. *If E is a set of equations then $E \models r = s$ iff $r = s$ is derivable from E using these rules.*

This result can be stated in an equivalent way. Namely, $E \models r = s$ iff there is a finite sequence $u_1, u_2, \ldots, u_n$ of terms such that r is u_1 and s is u_n and for all i, u_{i+1} is obtained from u_i by replacing a subterm t of u_i by a term u, where the equation $t = u$ or the equation $u = t$ is an instance of an equation in E.

This gives a method for deriving logical consequences of sets of equations. However, it is inefficient. Therefore it is of interest to find restrictions of these inference rules that are still capable of deriving all equational consequences of an equational system. This is the motivation for the theory of term-rewriting systems.

Term rewriting

The idea of a term rewriting system is to orient an equation $r = s$ into a rule $r \to s$ indicating that instances of r may be replaced by instances of s but not vice versa. Often this is done in such a way as to replace terms by simpler terms, where the definition of what is simple may be fairly subtle. However, as a first approximation, smaller terms are typically simpler. The equation $x + 0 = x$ then would typically be oriented into the rule $x + 0 \to x$. This reduces the generation of terms like $((x + 0) + 0) + 0$ which can appear in proofs if no such directionality is applied. The study of term rewriting systems is concerned with how to orient rules and what conditions guarantee that the resulting systems have the same computational power as the equational systems they came from.

Terminology

In this section, variables r, s, t, u refer to *terms* and $\to$ is a relation over terms. Thus the discussion is at a higher level than earlier.

A *term-rewriting system* R is a set of rules of the form $r \to s$, where r and s are terms. It is common to require that any variable that appears in s must also appear in r. It is also common to require that r is not a variable. The *rewrite relation* $\to_R$ is

defined by the following inference rules:

$$\frac{r \to s \qquad \rho \text{ a substitution}}{r\rho \to s\rho}$$

$$\frac{r \to s}{f(\ldots r \ldots) \to f(\ldots s \ldots)}$$

$$\frac{true}{r \to^* r}$$

$$\frac{r \to s}{r \to^* s}$$

$$\frac{r \to^* s \qquad s \to^* t}{r \to^* t}$$

$$\frac{r \to s}{r \leftrightarrow s}$$

$$\frac{s \to r}{r \leftrightarrow s}$$

$$\frac{true}{r \leftrightarrow^* r}$$

$$\frac{r \leftrightarrow s}{r \leftrightarrow^* s}$$

$$\frac{r \leftrightarrow^* s \qquad s \leftrightarrow^* t}{r \leftrightarrow^* t}$$

The notation $\vdash_r$ indicates derivability using these rules. The r subscript refers to "rewriting" (not to the term r). A set R of rules may be thought of as a set of logical axioms. Writing $s \to t$ is in R, indicates that $s \to t$ is such an axiom. Writing $R \vdash_r s \to t$ indicates that $s \to t$ may refer to a rewrite relation not included in R. Often $s \to_R t$ is used as an abbreviation for $R \vdash_r s \to t$, and sometimes the subscript R is dropped. Similarly, $\to_R^*$ is defined in terms of derivability from R. Note that the relation $\to_R^*$ is the reflexive transitive closure of $\to_R$. Thus $r \to_R^* s$ if there is a sequence $r_1, r_2, \ldots, r_n$ such that r_1 is r, r_n is s, and $r_i \to_R r_{i+1}$ for all i. Such a sequence is called a *rewrite sequence* from r to s, or a *derivation* from r to s. Note that $r \to_R^* r$ for all r and R. A term r is *reducible* if there is a term s such that $r \to s$, otherwise r is *irreducible*. If $r \to_R^* s$ and s is irreducible then s is called a *normal form* of r.

For example, given the system $R = \{x + 0 \to x, 0 + x \to x\}$, the term $0 + (y + 0)$ rewrites in two ways; $0 + (y + 0) \to 0 + y$ and $0 + (y + 0) \to y + 0$. Applying rewriting again, one obtains $0 + (y + 0) \to^* y$. In this case, y is a normal form of $0 + (y + 0)$, since y cannot be further rewritten. Computationally, rewriting a term s proceeds by finding a subterm t of s, called a *redex*, such that t is an instance of the left-hand side of some rule in R, and replacing t by the corresponding instance of the right-hand side of the rule. For example, $0 + (y + 0)$ is an instance of the left-hand side $0 + x$ of the rule $0 + x \to x$. The corresponding instance of the right-hand side x of this rule is $y + 0$, so $0 + (y + 0)$ is replaced by $y + 0$. This approach assumes that all variables on the right-hand side appear also on the left-hand side.

We now relate rewriting to equational theories. From the above rules, $r \leftrightarrow s$ if $r \rightarrow s$ or $s \rightarrow r$, and $\leftrightarrow^*$ is the reflexive transitive closure of $\leftrightarrow$. Thus $r \leftrightarrow^* s$ if there is a sequence $r_1, r_2, \ldots, r_n$ such that r_1 is r, r_n is s, and $r_i \leftrightarrow r_{i+1}$ for all i. Suppose R is a term rewriting system $\{r_1 \rightarrow s_1, \ldots, r_n \rightarrow s_n\}$. Define $R^=$ to be the associated equational system $\{r_1 = s_1, \ldots, r_n = s_n\}$. Also, $t =_R u$ is defined as $R^= \models t = u$, that is, the equation $t = u$ is a logical consequence of the associated equational system. The relation $=_R$ is thus the smallest congruence relation generated by R, in algebraic terms. The relation $=_R$ is defined semantically, and the relation $\rightarrow^*$ is defined syntactically. It is useful to find relationships between these two concepts in order to be able to compute properties of $=_R$ and to find complete restrictions of the inference rules of Birkhoff's theorem. Note that by Birkhoff's theorem, $R^= \models t = u$ iff $t \leftrightarrow^*_R u$. This is already a connection between the two concepts. However, the fact that rewriting can go in both directions in the derivation for $t \leftrightarrow^*_R u$ is a disadvantage. What we will show is that if R has certain properties, some of them decidable, then $t =_R u$ iff any normal form of t is the same as any normal form of u. This permits us to decide if $t =_R u$ by rewriting t and u to any normal form and checking if these are identical.

1.3.5 Confluence and Termination Properties

We now present some properties of term rewriting systems R. Equivalently, these can be thought of as properties of the rewrite relation $\rightarrow_R$. For terms s and t, $s \downarrow t$ means that there is a term u such that $s \rightarrow^* u$ and $t \rightarrow^* u$. Also, $s \uparrow t$ means that there is a term r such that $r \rightarrow^* s$ and $r \rightarrow^* t$. R is said to be *confluent* if for all terms s and t, $s \uparrow t$ implies $s \downarrow t$. The meaning of this is that any two rewrite sequences from a given term, can always be "brought together". Sometimes one is also interested in *ground confluence*. R is said to be ground confluent if for all ground terms r, if $r \rightarrow^* s$ and $r \rightarrow^* t$ then $s \downarrow t$. Most research in term rewriting systems concentrates on confluent systems.

A term rewriting system R (alternatively, a rewrite relation $\rightarrow$) has the *Church–Rosser property* if for all terms s and t, $s \leftrightarrow^* t$ iff $s \downarrow t$.

Theorem 1.3.8. (*See [192].*) *A term rewriting system R has the Church–Rosser property iff R is confluent.*

Since $s \leftrightarrow^* t$ iff $s =_R t$, this theorem connects the equational theory of R with rewriting. In order to decide if $s =_R t$ for confluent R it is only necessary to see if s and t rewrite to a common term.

Two term rewriting systems are said to be *equivalent* if their associated equational theories are equivalent (have the same logical consequences).

Definition 1.3.9. *A term rewriting system is* terminating (*strongly normalizing*) *if it has no infinite rewrite sequences. Informally, this means that the rewriting process, applied to a term, will eventually stop, no matter how the rewrite rules are applied.*

One desires all rewrite sequences to stop in order to guarantee that no matter how the rewriting is done, it will eventually terminate. An example of a terminating system

is $\{g(x) \rightarrow f(x), f(x) \rightarrow x\}$. The first rule changes g's to f's and so can only be applied as many times as there are g's. The second rule reduces the size and so it can only be applied as many times as the size of a term. An example of a nonterminating system is $\{x \rightarrow f(x)\}$. It can be difficult to determine if a system is terminating. The intuitive idea is that a system terminates if each rule makes a term simpler in some sense. However, the definition of simplicity is not always related to size. It can be that a term becomes simpler even if it becomes larger. In fact, it is not even partially decidable whether a term rewriting system is terminating [128]. Termination orderings are often used to prove that term rewriting systems are terminating. Recall the definition of termination ordering from Section 1.3.3.

Theorem 1.3.10. *Suppose R is a term rewriting system and $>$ is a termination ordering and for all rules $r \rightarrow s$ in R, $r > s$. Then R is terminating.*

This result can be extended to quasi-orderings, which are relations that are reflexive and transitive, but the above result should be enough to give an idea of the proof methods used. Many termination orderings are known; some will be discussed in Section 1.3.5. The orderings of interest are computable orderings, that is, it is decidable whether $r > s$ given terms r and s.

Note that if R is terminating, it is always possible to find a normal form of a term by any rewrite sequence continued long enough. However there can be more than one normal form. If R is terminating and confluent, there is exactly one normal form for every term. This gives a decision procedure for the equational theory, since for terms r and s, $r =_R s$ iff $r \leftrightarrow_R^* s$ (by Birkhoff's theorem) iff $r \downarrow s$ (by confluence) iff r and s have the same normal form (by termination). This gives us a directed form of theorem proving in such an equational theory. A term rewriting system which is both terminating and confluent is called *canonical*. Some authors use the term *convergent* for such systems [76]. Many such systems are known. Systems that are not terminating may still be *globally finite*, which means that for every term s there are finitely many terms t such that $s \rightarrow^* t$. For a discussion of global finiteness, see [105].

We have indicated how termination is shown; more will be presented in Section 1.3.5. However, we have not shown how to prove confluence. As stated, this looks like a difficult property. However, it turns out that if R is terminating, confluence is decidable, from Newman's lemma [192], given below. If R is not terminating, there are some methods that can still be used to prove confluence. This is interesting, even though in that case one does not get a decision procedure by rewriting to normal form, since it allows some flexibility in the rewriting procedure.

Definition 1.3.11. *A term rewriting system is* locally confluent (weakly confluent) *if for all terms r, s, and t, if $r \rightarrow s$ and $r \rightarrow t$ then $s \downarrow t$.*

Theorem 1.3.12 *(Newman's lemma). If R is locally confluent and terminating then R is confluent.*

It turns out that one can test whether R is locally confluent using *critical pairs* [144], so that local confluence is decidable for terminating systems. Also, if R is not locally confluent, it can sometimes be made so by computing critical pairs between

rewrite rules in R and using these critical pairs to add new rewrite rules to R until the process stops. This process is known as *completion* and was introduced by Knuth and Bendix [144]. Completion can also be seen as adding equations to a set of rewrite rules by ordered paramodulation and demodulation, deleting new equations that are instances of existing ones or that are instances of $x = x$. These new equations are then oriented into rewrite rules and the process continues. This process may terminate with a finite canonical term rewriting system or it may continue forever. It may also fail by generating an equation that cannot be oriented into a rewrite rule. One can still use *ordered rewriting* on such equations so that they function much as a term rewriting system [61]. When completion does not terminate, and even if it fails, it is still possible to use a modified version of the completion procedure as a semidecision procedure for the associated equational theory using the so-called *unfailing completion* [14, 15] which in the limit produces a ground confluent term rewriting system. In fact, Huet proved earlier [126] that if the original completion procedure does not fail, it provides a semidecision procedure for the associated equational theory.

Termination orderings

We give techniques to show that a term rewriting system is terminating. These all make use of well founded partial orderings on terms having the property that if $s \to t$ then $s > t$. If such an ordering exists, then a rewriting system is terminating since infinite reduction sequences correspond to infinite descending sequences of terms in the ordering. Recall from Section 1.3.3 that a termination ordering is a well-founded ordering that has the full invariance and replacement properties.

The termination ordering based on size was discussed in Section 1.3.3. Unfortunately, this ordering is too weak to handle many interesting systems such as those containing the rule $x * (y + z) \to x * y + x * z$, since the right-hand side is bigger than the left-hand side and has more occurrences of x. This ordering can be modified to weigh different symbols differently; the definition of $\|s\|$ can be modified to be a weighted sum of the number of occurrences of the symbols. The ordering of Knuth and Bendix [144] is more refined and is able to show that systems containing the rule $(x * y) * z \to x * (y * z)$ terminate.

Another class of termination orderings are the polynomial orderings suggested by Lankford [149, 150]. For these, each function and constant symbol is interpreted as a polynomial with integer coefficients and terms are ordered by the functions associated with them.

The recursive path ordering was discussed in Section 1.3.3. In order to handle the associativity rule $(x * y) * z \to x * (y * z)$ it is necessary to modify the ordering so that subterms are considered lexicographically. This lexicographic treatment of subterms is the idea of the lexicographic path ordering of Kamin and Levy [136]. Using this ordering, one can prove the termination of Ackermann's function. There are also many orderings intermediate between the recursive path ordering and the lexicographic path ordering; these are known as orderings with "status". The idea of status is that for some function symbols, when $f(s_1 \ldots s_m)$ and $f(t_1 \ldots t_n)$ are compared, the subterms s_i and t_i are compared using the multiset ordering. For other function symbols, the subterms are compared using the lexicographic ordering. For other function symbols, the subterms are compared using the lexicographic ordering in reverse, that is, from right to left; this is equivalent to reversing the lists and then applying

the lexicographic ordering. One can show that all such versions of the orderings are simplification orderings, for function symbols of bounded arity.

There are also many other orderings known that are similar to the above ones, such as the recursive decomposition ordering [132] and others; for some surveys see [75, 244]. In practice, *quasi-orderings* are often used to prove termination. A relation is a quasi-ordering if it is reflexive and transitive. A quasi-ordering is often written as $\geqslant$. Thus $x \geqslant x$ for all x, and if $x \geqslant y$ and $y \geqslant z$ then $x \geqslant z$. It is possible that $x \geqslant y$ and $y \geqslant x$ even if x and y are distinct; then one writes $x \approx y$ indicating that such x and y are in some sense "equivalent" in the ordering. One writes $x > y$ if $x \geqslant y$ but not $y \geqslant x$, for a quasi-ordering $\geqslant$. The relation $>$ is called the *strict part* of the quasi-ordering $\geqslant$. Note that the strict part of a quasi-ordering is a partial ordering. The multiset extension of a quasi-ordering is defined in a manner similar to the multiset extension of a partial ordering [131, 75].

Definition 1.3.13. *A quasi-ordering $\geqslant$ on terms satisfies the* replacement property (*is* monotonic) *if $s \geqslant t$ implies $f(\ldots s \ldots) \geqslant f(\ldots t \ldots)$. Note that it is possible to have $s > t$ and $f(\ldots s \ldots) \approx f(\ldots t \ldots)$.*

Definition 1.3.14. *A quasi-ordering $\geqslant$ is a* quasi-simplification ordering *if $f(\ldots t \ldots) \geqslant t$ for all terms and if $f(\ldots t \ldots) \geqslant f(\ldots\ldots)$ for all terms and all function symbols f of variable arity, and if the ordering satisfies the replacement property.*

Definition 1.3.15. *A quasi-ordering $\geqslant$ satisfies the* full invariance property (*see Section 1.3.5*) *if $s > t$ implies $s\Theta > t\Theta$ for all s, t, Θ.*

Theorem 1.3.16. (*See Dershowitz [74].*) *For terms over a finite set of function symbols, all quasi-simplification orderings have strict parts which are well founded.*

Proof. Using Kruskal's tree theorem [148]. □

Theorem 1.3.17. *Suppose R is a term rewriting system and $\geqslant$ is a quasi-simplification ordering which satisfies the full invariance property. Suppose that for all rules $l \to r$ in R, $l > r$. Then R is terminating.*

Actually, a version of the recursive path ordering adapted to quasi-orderings is known as the recursive path ordering in the literature. The idea is that terms that are identical up to permutation of arguments, are equivalent. There are a number of different orderings like the recursive path ordering.

Some decidability results about termination are known. In general, it is undecidable whether a system R is terminating [128]; however, for ground systems, that is, systems in which left and right-hand sides of rules are ground terms, termination is decidable [128]. For non-ground systems, termination of even one rule systems has been shown to be undecidable [63]. However, automatic tools have been developed that are very effective at either proving a system to be terminating or showing that it is not terminating, or finding an orientation of a set of equations that is terminating [120, 82, 145, 98]. In fact, one such system [145] from [91] was able to find an automatic

proof of termination of a system for which the termination proof was the main result of a couple of published papers.

A number of relationships between termination orderings and large ordinals have been found; this is only natural since any well-founded ordering corresponds to some ordinal. It is interesting that the recursive path ordering and other orderings provide intuitive and useful descriptions of large ordinals. For a discussion of this, see [75] and [73].

There has also been some work on modular properties of termination; for example, if one knows that R_1 and R_2 terminate, what can be said about the termination of $R_1 \cup R_2$ under certain conditions? For a few examples of works along this line, see [258, 259, 182].

1.3.6 Equational Rewriting

There are two motivations for equational rewriting. The first is that some rules are nonterminating and cannot be used with a conventional term rewriting system. One example is the commutative axiom $x + y = y + x$ which is nonterminating no matter how it is oriented into a rewrite rule. The second reason is that if an operator like $+$ is associative and commutative then there are many equivalent ways to represent terms like $a + b + c + d$. This imposes a burden in storage and time on a theorem prover or term rewriting system. Equational rewriting permits us to treat some axioms, like $x + y = y + x$, in a special way, avoiding problems with termination. It also permits us to avoid explicitly representing many equivalent forms of a term. The cost is a more complicated rewriting relation, more difficult termination proofs, and a more complicated completion procedure. Indeed, significant developments are still occurring in these areas, to attempt to deal with the problems involved. In equational rewriting, some equations are converted into rewrite rules R and others are treated as equations E. Typically, rules that terminate are placed in R and rules for which termination is difficult are placed in E, especially if E unification algorithms are known.

The general idea is to consider E-equivalence classes of terms instead of single terms. The E-equivalence classes consist of terms that are provably equal under E. For example, if E includes associative and commutative axioms for $+$, then the terms $(a+b)+c$, $a+(b+c)$, $c+(b+a)$, etc., will all be in the same E-equivalence class. Recall that $s =_E t$ if $E \models s = t$, that is, t can be obtained from s by replacing subterms using E. Note that $=_E$ is an equivalence relation. Usually some representation of the whole equivalence class is used; thus it is not necessary to store all the different terms in the class. This is a considerable savings in storage and time for term rewriting and theorem proving systems.

It is necessary to define a rewriting relation on E-equivalence classes of terms. If s is a term, let $[s]_E$ be its E-equivalence class, that is, the set of terms E-equivalent to s. The simplest approach is to say that $[s]_E \rightarrow [t]_E$ if $s \rightarrow t$. Retracting this back to individual terms, one writes $u \rightarrow_{R/E} v$ if there are terms s and t such that $u =_E s$ and $v =_E t$ and $s \rightarrow_R t$. This system R/E is called a *class rewriting system*. However, R/E rewriting turns out to be difficult to compute, since it requires searching through all terms E-equivalent to u. A computationally simpler idea is to say that $u \rightarrow v$ if u has a subterm s such that $s =_E s'$ and $s' \rightarrow_R t$ and v is u with s replaced by t. In this case one writes that $u \rightarrow_{R,E} v$. This system R, E is called the *extended rewrite*

system for R modulo E. Note that rules with E-equivalent left-hand sides need not be kept. The R, E rewrite relation only requires using the equational theory on the chosen redex s instead of the whole term, to match s with the left-hand side of some rule. Such E-matching is often (but not always, see [116]) easy enough computationally to make R, E rewriting much more efficient than R/E rewriting. Unfortunately, $\rightarrow_{R/E}$ has better logical properties for deciding $R \cup E$ equivalence. So the theory of equational rewriting is largely concerned with finding connections between these two rewriting relations.

Consider the systems R/E and R, E where R is $\{a * b \rightarrow d\}$ and E consists of the associative and commutative axioms for $*$. Suppose s is $(a * c) * b$ and t is $c * d$. Then $s \rightarrow_{R/E} t$ since s is E-equivalent to $c * (a * b)$. However, it is not true that $s \rightarrow_{R,E} t$ since there is no subterm of s that is E-equivalent to $a * b$. Suppose s is $(b * a) * c$. Then $s \rightarrow_{R,E} d * c$ since $b * a$ is E-equivalent to $a * b$.

Note that if E equivalence classes are nontrivial then it is impossible for class rewriting to be confluent in the traditional sense (since any term E-equivalent to a normal form will also be a normal form of a term). So it is necessary to modify the definition to allow E-equivalent normal forms. We want to capture the property that class rewriting is confluent when considered as a rewrite relation on equivalence classes. More precisely, R/E is (*class*) *confluent* if for any term t, if $t \rightarrow^*_{R/E} u$ and $t \rightarrow^*_{R/E} v$ then there are E-equivalent terms u' and v' such that $u \rightarrow^*_{R/E} u'$ and $v \rightarrow^*_{R/E} v'$. This implies that R/E is confluent and hence Church–Rosser, considered as a rewrite relation on E-equivalence classes. If R/E is class confluent and terminating then a term may have more than one normal form, but all of them will be E-equivalent. Furthermore, if R/E is class confluent and terminating, then any $R^= \cup E$ equivalent terms can be reduced to E equivalent terms by rewriting. Then an E-equivalence procedure can be used to decide $R^= \cup E$ equivalence, if there is one. Note that E-equivalent rules need not both be kept, for this method.

R is said to be *Church–Rosser modulo* E if any two $R^= \cup E$-equivalent terms can be R, E rewritten to E-equivalent terms. This is not the same as saying that R/E is Church–Rosser, considered as a rewrite system on E-equivalence classes; in fact, it is a stronger property. Note that R, E rewriting is a subset of R/E rewriting, so if R/E is terminating, so is R, E. If R/E is terminating and R is Church–Rosser modulo E then R, E rewriting is also terminating and $R^= \cup E$-equality is decidable if E-equality is. Also, the computationally simpler R, E rewriting can be used to decide the equational theory. But Church–Rosser modulo E is not a local property; in fact it is undecidable in general. Therefore one desires decidable sufficient conditions for it. This is the contribution of Jouannaud and Kirchner [130], using confluence and "coherence". The idea of coherence is that there should be some similarity in the way all elements of an E-equivalence class rewrite. Their conditions involve critical pairs between rules and equations and E-unification procedures.

Another approach is to add new rules to R to obtain a logically equivalent system R'/E; that is, $R^= \cup E$ and $R'^= \cup E$ have the same logical consequences (i.e., they are equivalent), but R', E rewriting is the same as R/E rewriting. Therefore it is possible to use the computationally simpler R', E rewriting to decide the equality theory of R/E. This is done for associative–commutative operators by Peterson and Stickel [205]. In this case, confluence can be decided by methods simpler than those of Jouannaud and Kirchner. Termination for equational rewriting systems is tricky to

decide; this will be discussed later. Another topic is completion for equational rewriting, adding rules to convert an equational rewriting system into a logically equivalent equational rewriting system with desired confluence properties. This is discussed by Peterson and Stickel [205] and also by Jouannaud and Kirchner [130]; for earlier work along this line see [151, 152].

AC rewriting

We now consider the special case of rewriting relative to the associative and commutative axioms $E = \{f(x, y) = f(y, x), f(f(x, y), z) = f(x, f(y, z))\}$ for a function symbol f. Special efficient methods exist for this case. One idea is to modify the term structure so that R, E rewriting can be used rather than R/E rewriting. This is done by *flattening*, that is a term $f(s_1, f(s_2, \ldots, f(s_{n-1}, s_n)\ldots))$, where none of the s_i have f as a top-level function symbol, is represented as $f(s_1, s_2, \ldots, s_n)$. Here f is a vary-adic symbol, which can take a variable number of arguments. Similarly, $f(f(s_1, s_2), s_3)$ is represented as $f(s_1, s_2, s_3)$. This represents all terms that are equivalent up to the associative equation $f(f(x, y), z) = f(x, f(y, z))$ by the same term. Also, terms that are equivalent up to permutation of arguments of f are also considered as identical. This means that each E-equivalence class is represented by a single term. This also means that all members of a given E-equivalence class have the same term structure, making R, E rewriting seem more of a possibility. Note however that the subterm structure has been changed; $f(s_1, s_2)$ is a subterm of $f(f(s_1, s_2), s_3)$ but there is no corresponding subterm of $f(s_1, s_2, s_3)$. This means that R, E rewriting does not simulate R/E rewriting on the original system. For example, consider the systems R/E and R, E where R is $\{a * b \to d\}$ and E consists of the associative and commutative axioms for $*$. Suppose s is $(a * b) * c$ and t is $d * c$. Then $s \to_{R/E} t$; in fact, $s \to_{R,E} t$. However, if one flattens the terms, then s becomes $*(a, b, c)$ and s no longer rewrites to t since the subterm $a * b$ has disappeared.

To overcome this, one adds *extensions* to rewrite rules to simulate their effect on flattened terms. The extension of the rule $\{a * b \to d\}$ is $\{*(x, a, b) \to *(x, d)\}$, where x is a new variable. With this extended rule, $*(a, b, c)$ rewrites to $d * c$. The general idea, then, is to flatten terms, and extend R by adding extensions of rewrite rules to it. Then, extended rewriting on flattened terms using the extended R is equivalent to class rewriting on the original R. Formally, suppose s and t are terms and s' and t' are their flattened forms. Suppose R is a term rewriting system and R' is R with the extensions added. Suppose E is associativity and commutativity. Then $s \to_{R/E} t$ iff $s' \to_{R',E} t'$. The extended R is obtained by adding, for each rule of the form $f(r_1, r_2, \ldots, r_n) \to s$ where f is associative and commutative, an extended rule of the form $f(x, r_1, r_2, \ldots, r_n) \to f(x, s)$, where x is a new variable. The original rule is also retained. This idea does not always work on other equational theories, however. Note that some kind of associative–commutative matching is needed for extended rewriting. This can be fairly expensive, since there are so many permutations to consider, but it is fairly straightforward to implement. Completion relative to associativity and commutativity can be done with the flattened representation; a method for this is given in [205]. This method requires associative–commutative unification (see Section 1.3.6).

Other sets of equations

The general topic of completion for other equational theories was addressed by Jouannaud and Kirchner in [130]. Earlier work along these lines was done by Lankford, as mentioned above. Such completion procedures may use E-unification. Also, they may distinguish rules with linear left-hand sides from other rules. (A term is *linear* if no variable appears more than once.)

AC termination orderings

We now consider termination orderings for special equational theories E. The problem is that E-equivalent terms are identified when doing equational rewriting, so that all E-equivalent terms have to be considered the same by the ordering. Equational rewriting causes considerable problems for the recursive path ordering and similar orderings. For example, consider the associative–commutative equations E. One can represent E-equivalence classes by flattened terms, as mentioned above. However, applying the recursive path ordering to such terms violates monotonicity. Suppose $* > +$ and $*$ is associative–commutative. Then $x*(y+z) > x*y+x*z$. By monotonicity, one should have $u*x*(y+z) > u*(x*y+x*z)$. In fact, this fails; the term on the right is larger in the recursive path ordering. A number of attempts have been made to overcome this. The first was the associative path ordering of Dershowitz, Hsiang, Josephson, and Plaisted [78], developed by the last author. This ordering applied to transformed terms, in which big operators like $*$ were pushed inside small operators like $+$. The ordering was not originally extended to non-ground terms, but it seems that it would be fairly simple to do so using the fact that a variable is smaller than any term properly containing it. A simpler approach to extending this ordering to non-ground terms was given later by Plaisted [209], and then further developed in Bachmair and Plaisted [12], but this requires certain conditions on the precedence. This work was generalized by Bachmair and Dershowitz [13] using the idea of "commutation" between two term rewriting systems. Later, Kapur [139] devised a fully general associative termination ordering that applies to non-ground terms, but may be hard to compute. Work in this area has continued since that time [146]. Another issue is the incorporation of status in such orderings, such as left-to-right, right-to-left, or multiset, for various function symbols. E-termination orderings for other equational theories may be even more complicated than for associativity and commutativity.

Congruence closure

Suppose one wants to determine whether $E \models s = t$ where E is a set (conjunction) of ground equations and s and t are ground terms. For example, one may want to decide whether $\{f^5(c) = c,\ f^3(c) = c\} \models f(c) = c$. This is a case in which rewriting techniques apply but another method is more efficient. The method is called *congruence closure* [191]; for some efficient implementations and data structures see [81]. The idea of congruence closure is essentially to use equality axioms, but restricted to terms that appear in E, including its subterms. For the above problem, the following is a derivation of $f(c) = c$, identifying equations $u = v$ and $v = u$:

1. $f^5(c) = c$ (given).
2. $f^3(c) = c$ (given).

3. $f^4(c) = f(c)$ (2, using equality replacement).
4. $f^5(c) = f^2(c)$ (3, using equality replacement).
5. $f^2(c) = c$ (1, 4, transitivity).
6. $f^3(c) = f(c)$ (5, using equality replacement).
7. $f(c) = c$ (2, 6, transitivity).

One can show that this approach is complete.

E-unification algorithms

When the set of axioms in a theorem to be proved includes a set E of equations, it is often better to use specialized methods than general theorem proving techniques. For example, if the binary infix operator $*$ is associative and commutative, many equivalent terms $x * (y * z)$, $y * (x * z)$, $y * (z * x)$, etc. may be generated. These cannot be eliminated by rewriting since none is simpler than the others. Even the idea of using unorderable equations as rewrite rules when the applied instance is orderable, will not help. One approach to this problem is to incorporate a general E-unification algorithm into the theorem prover. Plotkin [214] first discussed this general concept and showed its completeness in the context of theorem proving. With E unification built into a prover, only one representative of each E-equivalence class need be kept, significantly reducing the number of formulas retained. E-unification is also known as semantic unification, which may be a misnomer since no semantics (interpretation) is really involved. The general idea is that if E is a set of equations, an E-unifier of two terms s and t is a substitution Θ such that $E \models s\Theta = t\Theta$, and a most general E-unifier is an E-unifier that is as general as possible in a certain technical sense relative to the theory E. Many unification algorithms for various sets of equations have been developed [239, 9]. For some theories, there may be at most one most general E-unifier, and for others, there may be more than one, or even infinitely many, most general E-unifiers.

An important special case, already mentioned above in the context of term-rewriting, is associative–commutative (AC) unification. In this case, if two terms are E-unifiable, then there are at most finitely many most general E-unifiers, and there are algorithms to find them that are usually efficient in practice. The well-known algorithm of [251] essentially involves solving Diophantine equations and finding a basis for the set of solutions and finding combinations of basis vectors in which all variables are present. This can sometimes be very time consuming; the time to perform AC-unification can be double exponential in the sizes of the terms being unified [137]. Domenjoud [80] showed that the two terms $x + x + x + x$ and $y_1 + y_2 + y_3 + y_4$ have more than 34 billion different AC unifiers. Perhaps AC unification algorithm is artificially adding complexity to theorem proving, or perhaps the problem of theorem proving in the presence of AC axioms is really hard, and the difficulty of the AC unification simply reveals that. There may be ways of reducing the work involved in AC unification. For example, one might consider resource bounded AC unification, that is, finding all unifiers within some size bound. This might reduce the number of unifiers in cases where many of them are very large. Another idea is to consider "optional

variables", that is, variables that may or may not be present. If x is not present in the product $x * y$ then this product is equivalent to y. This is essentially equivalent to introducing a new identity operator, and greatly reduces the number of AC unifiers. This approach has been studied by Domenjoud [79]. This permits one to represent a large number of solutions compactly, but requires one to keep track of optionality conditions.

Rule-based unification

Unification can be viewed as equation solving, and therefore is part of theorem proving or possibly logic programming. This approach to unification permits conceptual simplicity and also is convenient for theoretical investigations. For example, unifying two literals $P(s_1, s_2, \ldots, s_n)$ and $P(t_1, t_2, \ldots, t_n)$ can be viewed as solving the set of equations $\{s_1 = t_1, s_2 = t_2, \ldots, s_n = t_n\}$. Unification can be expressed as a collection of rules operating on such sets of equations to either obtain a most general unifier or detect non-unifiability. For example, one rule replaces an equation $f(u_1, u_2, \ldots, u_n) = f(v_1, v_2, \ldots, v_n)$ by the set of equations $\{u_1 = v_1, u_2 = v_2, \ldots, u_n = v_n\}$. Another rule detects non-unifiability if there is an equation of the form $f(\ldots) = g(\ldots)$ for distinct f and g. Another rule detects non-unifiability if there is an equation of the form $x = t$ where t is a term properly containing x. With a few more such rules, one can obtain a simple unification algorithm that will terminate with a set of equations representing a most general unifier. For example, the set of equations $\{x = f(a), y = g(f(a))\}$ would represent the substitution $\{x \leftarrow f(a), y \leftarrow g(f(a))\}$. This approach has also been extended to E-unification for various equational theories E. For a survey of this approach, see [133].

1.3.7 Other Logics

Up to now, we have considered theorem proving in general first-order logic. However, there are many more specialized logics for which more efficient methods exist. Such logics fix the domain of the interpretation, such as to the reals or integers, and also the interpretations of some of the symbols, such as "+" and "∗". Examples of theories considered include Presburger arithmetic, the first-order theory of natural numbers with addition [200], Euclidean and non-Euclidean geometry [272, 55], inequalities involving real polynomials (for which Tarski first gave a decision procedure) [52], ground equalities and inequalities, for which congruence closure [191] is an efficient decision procedure, modal logic, temporal logic, and many more specialized logics. Theorem proving for ground formulas of first-order logic is also known as *satisfiability modulo theories* (SMT) in the literature. Description logics [8], discussed in Chapter 3 of this Handbook, are sublanguages of first-order logic, with extensions, that often have efficient decision procedures and have applications to the semantic web. Specialized logics are often built into provers or logic programming systems using *constraints* [33]. The idea of using constraints in theorem proving has been around for some time [143]. Another specialized area is that of computing polynomial ideals, for which efficient methods have been developed [44]. An approach to combining decision procedures was given in [190] and there has been continued interest in the combination of decision procedures since that time.

Higher-order logic

In addition to the logics mentioned above, there are more general logics to consider, including higher-order logics. Such logics permit quantification over functions and predicates, as well as variables. The HOL prover [101] uses higher-order logic and permits users to give considerable guidance in the search for a proof. Andrews' TPS prover is more automatic, and has obtained some impressive proofs fully automatically, including Cantor's theorem that the powerset of a set has more elements than the set. The TPS prover was greatly aided by a breadth-first method of instantiating matings described in [31]. In general, higher-order logic often permits a more natural formulation of a theorem than first-order logic, and shorter proofs, in addition to being more expressive. But of course the price is that the theorem prover is more complicated; in particular, higher-order unification is considerably more complex than first-order unification.

Mathematical induction

Without going to a full higher-order logic, one can still obtain a considerable increase in power by adding mathematical induction to a first-order prover. The mathematical induction schema is the following one:

$$\frac{\forall y[[\forall x((x < y) \rightarrow P(x))] \rightarrow P(y)]}{\forall y P(y)}.$$

Here $<$ is a well-founded ordering. Specializing this to the usual ordering on the integers, one obtains the following Peano induction schema:

$$\frac{P(0), \forall x(P(x) \rightarrow P(x+1))}{\forall x P(x)}.$$

With such inference rules, one can, for example, prove that addition and multiplication are associative and commutative, given their straightforward definitions. Both of these induction schemas are second-order, because the predicate P is implicitly universally quantified. The problem in using these schemas in an automatic theorem prover is in instantiating P. Once this is done, the induction schema can often be proved by first-order techniques. One way to adapt a first-order prover to perform mathematical induction, then, is simply to permit a human to instantiate P. The problem of instantiating P is similar to the problem of finding loop invariants for program verification.

By instantiating P is meant replacing $P(y)$ in the above formula by $A[y]$ for some first-order formula A containing the variable y. Equivalently, this means instantiating P to the function $\lambda z.A[z]$. When this is done, the first schema above becomes

$$\frac{\forall y[[\forall x((x < y) \rightarrow A[x])] \rightarrow A[y]]}{\forall y A[y]}.$$

Note that the hypothesis and conclusion are now first-order formulas. This instantiated induction schema can then be given to a first-order prover. One way to do this is to have the prover prove the formula $\forall y[[\forall x((x < y) \rightarrow A[x])] \rightarrow A[y]]$, and then conclude $\forall y A[y]$. Another approach is to add the first-order formula $\{\forall y[[\forall x((x < y) \rightarrow A[x])] \rightarrow A[y]]\} \rightarrow \{\forall y A[y]\}$ to the set of axioms. Both approaches are facilitated by using a structure-preserving translation of these formulas to clause form,

in which the formula $A[y]$ is defined to be equivalent to $P(y)$ for a new predicate symbol P.

A number of semi-automatic techniques for finding such a formula A and choosing the ordering $<$ have been developed. One of them is the following: To prove that for all finite ground terms t, $A[t]$, first prove $A[c]$ for all constant symbols c, and then for each function symbol f of arity n prove that $A[t_1] \wedge A[t_2] \wedge \cdots \wedge A[t_n] \rightarrow A[f(t_1, t_2, \ldots, t_n)]$. This is known as *structural induction* and is often reasonably effective.

A common case when an induction proof may be necessary is when the prover is not able to prove the formula $\forall x A[x]$, but the formulas $A[t]$ are separately provable for all ground terms t. Analogously, it may not be possible to prove that $\forall x(\text{natural\_number}(x) \rightarrow A[x])$, but one may be able to prove $A[0]$, $A[1]$, $A[2]$, ... individually. In such a case, it is reasonable to try to prove $\forall x A[x]$ by induction, instantiating $P(x)$ in the above schema to $A[x]$. However, this still does not specify which ordering $<$ to use. For this, it can be useful to detect how long it takes to prove the $A[t]$ individually. For example, if the time to prove $A[n]$ for natural number n is proportional to n, then one may want to try the usual (size) ordering on natural numbers. If $A[n]$ is easy to prove for all even n but for odd n, the time is proportional to n, then one may try to prove the even case directly without induction and the odd case by induction, using the usual ordering on natural numbers.

The Boyer–Moore prover NqTHM [38, 36] has mathematical induction techniques built in, and many difficult proofs have been done on it, generally with substantial human guidance. For example, correctness of AMD Athlon's elementary floating point operations, and parts of IBM Power 5 and other processors have been proved on it. ACL2 [142, 141] is a software system built on Common Lisp related to NqTHM that is intended to be an industrial strength version of NqTHM, mainly for the purpose of software and hardware verification. Boyer, Kaufmann, and Moore won the ACM Software System Award in 2005 for these provers. A number of other provers also have automatic or semi-automatic induction proof techniques. Rippling [47] is a technique originally developed for mathematical induction but which also has applications to summing series and general equational reasoning. The *ground reducibility* property is also often used for induction proofs, and has applications to showing the completeness of algebraic specifications [134]. A term is *ground reducible* by a term rewriting system R if all its ground instances are reducible by R. This property was first shown decidable in [210], with another proof soon after in [138]. It was shown to be exponential time complete by Comon and Jacquemard [60]. However, closely related versions of this problem are undecidable. Recently Kapur and Subramaniam [140] described a class of inductive theorems for which validity is decidable, and this work was extended by Giesl and Kapur [97]. Bundy has written an excellent survey of inductive theorem proving [46] and the same handbook also has a survey of the so-called *inductionless induction* technique, which is based on completion of term-rewriting systems [59]; see also [127].

Set theory

Since most of mathematics can be expressed in terms of set theory, it is logical to develop theorem proving methods that apply directly to theorems expressed in set theory. Second-order provers do this implicitly. First-order provers can be used for set

theory as well; Zermelo–Fraenkel set theory consists of an infinite set of first-order axioms, and so one again has the problem of instantiating the axiom schemas so that a first-order prover can be used. There is another version of set theory known as von Neumann–Bernays–Gödel set theory [37] which is already expressed in first-order logic. Quite a bit of work has been done on this version of set theory as applied to automated deduction problems. Unfortunately, this version of set theory is somewhat cumbersome for a human or for a machine. Still, some mathematicians have an interest in this approach. There are also a number of systems in which humans can construct proofs in set theory, such as Mizar [260] and others [26, 219]. In fact, there is an entire project (the QED project) devoted to computer-aided translation of mathematical proofs into completely formalized proofs [218].

It is interesting to note in this respect that many set theory proofs that are simple for a human are very hard for resolution and other clause-based theorem provers. This includes theorems about the associativity of union and intersection, for example. In this area, it seems worthwhile to incorporate more of the simple definitional replacement approach used by humans into clause-based theorem provers.

As an example of the problem, suppose that it is desired to prove that $\forall x((x \cap x) = x)$ from the axioms of set theory. A human would typically prove this by noting that $(x \cap x) = x$ is equivalent to $((x \cap x) \subseteq x) \wedge (x \subseteq (x \cap x))$, then observe that $A \subseteq B$ is equivalent to $\forall y((y \in A) \rightarrow (y \in B))$, and finally observe that $y \in (x \cap x)$ is equivalent to $(y \in x) \wedge (y \in x)$. After applying all of these equivalences to the original theorem, a human would observe that the result is a tautology, thus proving the theorem.

But for a resolution theorem prover, the situation is not so simple. The axioms needed for this proof are

$$(x = y) \leftrightarrow [(x \subseteq y) \wedge (y \subseteq x)],$$
$$(x \subseteq y) \leftrightarrow \forall z((z \in x) \rightarrow (z \in y)),$$
$$(z \in (x \cap y)) \leftrightarrow [(z \in x) \wedge (z \in y)].$$

When these are all translated into clause form and Skolemized, the intuition of replacing a formula by its definition gets lost in a mass of Skolem functions, and a resolution prover has a much harder time. This particular example may be easy enough for a resolution prover to obtain, but other examples that are easy for a human quickly become very difficult for a resolution theorem prover using the standard approach.

The problem is more general than set theory, and has to do with how definitions are treated by resolution theorem provers. One possible method to deal with this problem is to use "replacement rules" as described in [154]. This gives a considerable improvement in efficiency on many problems of this kind. Andrews' matings prover has a method of selectively instantiating definitions [32] that also helps on such problems in a higher-order context. The U-rules of OSHL also help significantly [184].

1.4 Applications of Automated Theorem Provers

Among theorem proving applications, we can distinguish between those applications that are truly automated, and those requiring some level of human intervention; be-

tween KR and non-KR applications; and between applications using classical first-order theorem provers and those that do not. In the latter category fall applications using theorem proving systems that do not support equality, or allow only restricted languages such as Horn clause logic, or supply inferential procedures beyond those of classical theorem proving.

These distinctions are not independent. In general, applications requiring human intervention have been only slightly used for KR; moreover, KR applications are more likely to use a restricted language, or to use special-purpose inferential procedures.

It should be noted that any theorem proving system that can solve the math problems that form a substantial part of the TPTP (Thousands of Problems for Theorem Provers) testbed [255] must be a classical first-order theorem prover that supports equality.

1.4.1 Applications Involving Human Intervention

Because theorem proving is in general intractable, the majority of applications of automated theorem provers require direction from human users in order to work. The intervention required can be extensive, e.g., the user may be required to supply lemmas to the proofs on which the automated theorem prover is working [84]. In the worst case, a user may be required to supply every step of a proof to an automated theorem prover; in this case, the automated theorem prover is functioning simply as a proof checker.

The need for human intervention has often limited the applicability of automated theorem provers to applications where reasoning can be done offline; that is, where the reasoner is not used as part of a real-time application. Even given this restriction, automated theorem provers have proved very valuable in a number of domains, including software development and verification of software and hardware.

Software development

An example of an application to software development is the Amphion system, which was developed by Stickel et al. [250] and uses the SNARK theorem prover [249]. It has been used by NASA to compose programs out of a library of FORTRAN-77 subroutines. The user of Amphion, who does not have to have any familiarity with either theorem proving or the library subroutines, gives a graphical specification; this specification is translated into a theorem of first-order logic; and SNARK provides a constructive proof of this theorem. This constructive proof is then translated into the application program in FORTRAN-77.

The NORA/HAMMR system [86] similarly determines what software components can be reused during program development. Each software component is associated with a *contract* written in a formal language which captures the essentials of the component's behavior. The system determines whether candidate components have compatible contracts and are thus potentially reusable; the proof of compatibility is carried out using an automated theorem prover, though with a fair amount of human guidance. Automated theorem provers used for NORA/HAMMR include Setheo [158], Spass [268, 269], and PROTEIN [24], a theorem prover based on Mark Stickel's PTTP [246, 248].

In the area of algorithm design and program analysis and optimization, KIDS (Kestrel Interactive Development System) [241] is a program derivation system that

uses automated theorem proving technology to facilitate the derivation of programs from high-level program specifications. The program specification is viewed as a goal, and rules of transformational development are viewed as axioms of the system. The system, guided by the user, searches to find the appropriate transformational rules, the application of which leads to the final program. Both Amphion and KIDS require relatively little intervention from the user once the initial specification is made; KIDS, for example, requires active interaction only for the algorithm design tactic.

Hardware and software verification

Formal verification of both hardware and software has been a particularly fruitful application of automated theorem provers. The need for verification of program correctness had been noted as far back as the early 1960s by McCarthy [172], who suggested approaching the problem by stating a theorem that a program had certain properties—and in particular, computed certain functions—and then using an automated theorem prover to prove this theorem. Verification of cryptographic protocols is another important subfield of this area.

The field of hardware verification can be traced back to the design of the first hardware description languages, e.g., ISP [27], and became active in the 1970s and 1980s, with the advent of VLSI design. (See, e.g, [22].) It gained further prominence after the discovery in 1994 [108] of the Pentium FDIV bug, a bug in the floating point unit of Pentium processors. It was caused by missing lookup table entries and led to incorrect results for some floating point division operators. The error was widespread, well-publicized, and costly to Intel, Pentium's manufacturer, since it was obliged to offer to replace all affected Pentium processors.

General-purpose automated theorem provers that have been commonly used for hardware and/or software verification include the following:

- The Boyer–Moore theorem provers NqTHM and ACL2 [36, 142] were inspired by McCarthy's first papers on the topic of verifying program correctness. As mentioned in the previous section, these award-winning theorem provers have been used for many verification applications.

- The Isabelle theorem prover [203, 197] can handle higher-order logics and temporal logics. Isabelle is thus especially well-suited for cases where program specifications are written in temporal or dynamic logic (as is frequently the case). It has also been used for verification of cryptographic protocols [242], which are frequently written in higher order and/or epistemic logics [49].

- OTTER has been used for a system that analyzes and detects attacks on security APIs (application programming interfaces) [273].

Special-purpose verification systems which build verification techniques on top of a theorem prover include the following:

- The PVS system [201] has been used by NASA's SPIDER (Scalable Processor-Independent Design for Enhanced Reliability) to verify SPIDER protocols [206].

- The KIV (Karlsruhe Interactive Verifier) has been used for a range of software verification applications, including validation of knowledge-based systems [84].

The underlying approach is similar to that of the KIDS and Amphion projects in that first, the user is required to enter a specification; second, the user is entering a specification of a modularized system, and the interactions between the modules; and third, the user works with the system to construct a proof of validity. More interaction between the user and the theorem prover seems to be required in this case, perhaps due to the increased complexity of the problem. KIV offers a number of techniques to reduce the burden on the user, including reuse of proofs and the generation of counterexamples.

1.4.2 Non-Interactive KR Applications of Automated Theorem Provers

McCarthy argued [171] for an AI system consisting of a set of axioms and an automated theorem prover to reason with those axioms. The first implementation of this vision came in the late 1960s with Cordell Green's question-answering system QA3 and planning system [103, 104]. Given a set of facts and a question, Green's question-answering system worked by resolving the (negated) question against the set of facts. Green's planning system used resolution theorem proving on a set of axioms representing facts about the world in order to make simple inferences about moving blocks in a simple blocks-world domain. In the late 1960s and early 1970s, SRI's Shakey project [195] attempted to use the planning system STRIPS [85] for robot motion planning; automated theorem proving was used to determine applicability of operators and differences between states [232]. The difficulties posed by the intractability of theorem proving became evident. (Shakey also faced other problems, including dealing with noisy sensors and incomplete knowledge. Moreover, the Shakey project does not actually count as a non-interactive application of automated theorem proving, since people could obviously change Shakey's environment while it acted. Nonetheless, projects like these underscored the importance of dealing effectively with theorem proving's essential intractability.)

In fact, there are today many fewer non-interactive than interactive applications of theorem proving, due to its computational complexity. Moreover, non-interactive applications will generally use carefully crafted heuristics that are tailored and fine-tuned to a particular domain or application. Without such heuristics, the theorem-proving program would not be able to handle the huge number of clauses generated. Finally, as mentioned above, non-interactive applications often use ATPs that are not general theorem provers with complete proof procedures. This is because completeness and generality often come at the price of efficiency.

Some of the most successful non-interactive ATP applications are based on two theorem provers developed by Mark Stickel at SRI, PTTP [246, 248] and SNARK [249]. PTTP attempts to retain as much as possible the efficiency of Prolog (see Section 1.4.4 below) while it remedies the ways in which Prolog fails as a general-purpose theorem prover, namely, its unsound unification algorithm, its incomplete search strategy, and its incomplete inference system. PTTP was used in SRI's TACITUS system [121, 124], a message understanding system for reports on equipment failure, naval operations, and terrorist activities. PTTP was used specifically to furnish minimal-cost abductive explanations. It is frequently necessary to perform abduction—that is, to posit a likely explanation—when processing text. For example, to understand the sentence "The Boston office called", one must understand that the construct of metonymy

(the use of a single characteristic to identify an entity of which it is an attribute) is being used, and that what is meant is *a person in the office* called. Thus, to understand the sentence we must posit an explanation of a person being in the office and making that call.

There are usually many possible explanations that can be posited for any particular phenomenon; thus, the problem arises of choosing the simplest non-trivial explanation. (One would not, for example, wish to posit an explanation consistent with an office actually being able to make a call.) TACITUS considers explanations of the form $P(a)$, where $\forall x\, P(x) \rightarrow Q(x)$ and $Q(a)$ are in the theory, and chooses the explanation that has minimal cost [247]. Every conjunct in the logical form of a sentence is given an assumability cost; this cost is passed back to antecedents in the Horn clause. Because of the way costs are propagated, the cost may be partly dependent on the length of the proofs of the literals in the explanation.

PTTP was also used in a central component of Stanford's Logic-Based Subsumption Architecture for robot control [1], which was used to program a Nomad-200 robot to travel to different rooms in a multi-story building. The system employed a multi-layered architecture; in each layer, PTTP was used to prove theorems from the given axioms. Goals were transmitted to layers below or to robot manipulators.

PTTP is fully automated; the user has no control over the search for solutions. In particular, each rule is used in its original form and in its contrapositive. In certain situations, such as stating principles about substituting equals, reasoning with a contrapositive form can lead to considerable inefficiency.

Stickel's successor theorem prover to PTTP, SNARK [249], gives users this control. It is more closely patterned after Otter; difficult theorems that are intractable for PTTP can be handled by SNARK. It was used as the reasoning component for SRI's participation in DARPA's High-Performance Knowledge Bases (HPKB) Project [58], which focused on constructing large knowledge bases in the domain of crisis management; and developing question-answering systems for querying these knowledge bases. SNARK was used primarily in SRI's question-answering portion of that system. SNARK, in contrast to what would have been possible with PTTP, allowed users to fine tune the question-answering system for HPKB, by crafting an ordering of predicates and clauses on which resolution would be performed. This ordering could be modified as the knowledge base was altered. Such strategies were necessary to get SNARK to work effectively given the large size of the HPKB knowledge base.

For its use in the HPKB project, SNARK had to be extended to handle temporal reasoning.

SNARK has also been used for consistency checking of semantic web ontologies [20].

Other general-purpose theorem provers have also been used for natural language applications, though on a smaller scale and for less mature applications. Otter has been used in PENG (Processable English) [236], a controlled natural language used for writing precise specifications. Specifications in PENG can be translated into first-order logic; Otter is then used to draw conclusions. As discussed in detail in Chapter 20, Bos and Markert [35] have used Vampire (as well as the Paradox model finder) to determine whether a hypothesis is entailed by some text.

The Cyc artificial intelligence project [157, 156, 169] is another widespread application of non-interactive automated theorem proving. The ultimate goal of Cyc is

the development of a comprehensive, encyclopedic knowledge base of commonsense facts, along with inference mechanisms for reasoning with that knowledge. Cyc contains an ontology giving taxonomic information about commonsense concepts, as well as assertions about the concepts.

Cyc's underlying language, CycL, allows expression of various constructs that go beyond first-order logic. Examples include:

- The concept of contexts [50]: one can state that something is true in a particular context as opposed to absolutely. (E.g., the statement that vampires are afraid of garlic is true in a mythological context, though not in real life.)
- Higher-order concepts. (E.g., one can state that if a relation is reflexive, symmetric, and transitive, it is an equivalence relation.)
- Exceptions. (E.g., one can say that except for Taiwan, all Chinese provinces are part of the People's Republic of China.)

The Cyc knowledge base is huge. Nevertheless, it has been successfully used in real-world applications, including HPKB. (Cyc currently has over 3 million assertions; at the time of its use in HPKB, it had over a million assertions.) Theorem proving in Cyc is incomplete but efficient, partly due to various special-purpose mechanisms for reasoning with its higher-order constructs. For example, Cyc's reasoner includes a special module for solving *disjointWith* queries that traverses the taxonomies in the knowledge base to determine whether two classes have an empty intersection.

Ramachandran et al. [221, 220] compared the performance of Cyc's reasoner with standard theorem provers. First, most of ResearchCyc's knowledge base[4] was translated into first-order logic. The translated sentences were then loaded into various theorem provers, namely, Vampire, E [235], Spass, and Otter. The installations of Vampire and Spass available to Ramachandran et al. did not have sufficient memory to load all assertions, necessitating performing the comparison of Cyc with these theorem provers on just 10 percent of ResearchCyc's knowledge base. On sample queries—e.g., "Babies can't be doctors", "If the U.S. bombs Iraq, someone is responsible", –Cyc proved to be considerably more efficient. For example, for the query about babies and doctors, Cyc took 0.04 seconds to answer the query, while Vampire took 847.6 seconds.

Ramachandran and his colleagues conjecture that the disparity in performance partly reflects the fact that Cyc's reasoner and the standard theorem provers have been designed for different sets of problems. General automated theorem provers have been designed to perform deep inference on small sets of axioms. If one looks at the problems in the TPTP database, they often have just a few dozen and rarely have more than a few hundred axioms. Cyc's reasoner, on the other hand, has been designed to perform relatively shallow inference on large sets of axioms.

It is also worth noting that the greatest disparity of inference time between Cyc and the other theorem provers occurred when Cyc was using a special purpose reasoning module. In that sense, of course, purists might argue that Cyc is not really doing

[4] ResearchCyc [169] contains the knowledge base open to the public for research; certain portions of Cyc itself are not open to the public. The knowledge base of ResearchCyc contains over a million assertions.

theorem proving faster than standard ATPs; rather, it is doing something that is functionally equivalent to theorem proving while ATPs are doing theorem proving, and it is doing that something much faster.

1.4.3 Exploiting Structure

Knowledge bases for real-world applications and commonsense reasoning often exhibit a modular-like structure, containing multiple sets of facts with relatively little connection to one another. For example, a knowledge base in the banking domain might contain sets of facts concerning loans, checking accounts, and investment instruments; moreover, these sets of facts might have little overlap with one another. In such a situation, reasoning would primarily take place within a module, rather than between modules. Reasoning between modules would take place—for example, one might want to reason about using automated payments from a checking account to pay off installments on a loan—but would be limited. One would expect that a theorem prover that takes advantage of this modularity would be more efficient: most of the time, it would be doing searches in reduced spaces, and it would produce fewer irrelevant resolvents.

A recent trend in automated reasoning focuses on exploiting structure of a knowledge base to improve performance. This section presents a detailed example of such an approach. Amir and McIlraith [2] have studied the ways in which a knowledge base can be automatically partitioned into loosely coupled clusters of domain knowledge, forming a network of subtheories. The subtheories in the network are linked via the literals they share in common. Inference is carried out within a subtheory; if a literal is inferred within one subtheory that links to another subtheory, it may be passed from the first to the second subtheory.

Consider, from [2], the following theory specifying the workings of an espresso machine, and the preparation of espresso and tea: (Note that while this example is propositional, the theory is first-order.)

(1) $\neg$ okpump $\vee\neg$ onpump $\vee$ water

(2) $\neg$ manfill $\vee$ water

(3) $\neg$ manfill $\vee\neg$ onpump

(4) manfill $\vee$ onpump

(5) $\neg$ water $\vee\neg$ okboiler $\vee\neg$ onboiler $\vee$ steam

(6) water $\vee\neg$ steam

(7) okboiler $\vee\neg$ steam

(8) onboiler $\vee\neg$ steam

(9) $\neg$ steam $\vee\neg$ cofee $\vee$ hotdrink

(10) coffee $\vee$ teabag

(11) $\neg$ steam $\vee\neg$ teabag $\vee$ hotdrink

Intuitively, this theory can be decomposed into three subtheories. The first, *A1*, containing axioms 1 through 4, regards water in the machine; it specifies the relations between manually filling the machine with water, having a working pump, and having water in the machine. The second, *A2*, containing axioms 5 through 8, regards getting steam; it specifies the relations between having water, a working boiler, the boiler switch turned on, and steam. The third, *A3*, containing axioms 9 through 11, regards getting a hot drink; it specifies the relation between having steam, having coffee, having a teabag, and having a hot drink.

In this partitioning, the literal *water* links *A1* and *A2*; the literal *steam* links *A2* and *A3*. One can reason with logical partitions using forward message-passing of linking literals. If one asserts *okpump*, and performs resolution on the clauses of *A1*, one obtains *water*. If one asserts *okboiler* and *onboiler* in *A2*, passes *water* from *A1* to *A2*, and performs resolution in *A2*, one obtains *steam*. If one passes *steam* to *A3* and performs resolution in *A3*, one obtains *hotdrink*.

In general, the complexity of this sort of reasoning depends on the number of partitions, the size of the partitions, the interconnectedness of the subtheory graph, and the number of literals linking subtheories. When partitioning the knowledge base, one wants to minimize these parameters to the extent possible. (Note that one cannot simultaneously minimize all parameters; as the number of partitions goes down, the size of at least some of the partitions goes up.)

McIlraith et al. [165] did some empirical studies on large parts of the Cyc database used for HPKB, comparing the results of the SNARK theorem prover with and without this partitioning strategy. SNARK plus (automatically-performed) partitioning performed considerably better than SNARK with no strategy, though it was comparable to SNARK plus set-of-support strategies. When partitioning was paired with another strategy like set-of-support, it outperformed combinations of strategies without partitioning.

Clustering to improve reasoning performance has also been explored by Hayes et al. [115]. In a similar spirit, there has been growing interest in modularization of ontologies from the Description Logic and Semantic Web communities [267, 223, 102]. Researchers have been investigating how such modularization affects the efficiency of reasoning (i.e., performing subsumption and classification, and performing consistency checks) over the ontologies.

1.4.4 Prolog

In terms of its use in working applications, the logic programming paradigm [147] represents an important success in automated theorem proving. Its main advantage is its efficiency; this makes it suitable for real-world applications. The most popular language for logic programming is Prolog [41].

What makes Prolog work so efficiently is a combination of the restricted form of first-order logic used, and the particular resolution and search strategies that are implemented. In the simplest case, a Prolog program consists of a set of Horn clauses; that is, either atomic formulas or implications of the form $(P_1 \wedge P_2 \wedge \cdots) \rightarrow P_0$, where the P_i's are all atomic formulas. This translates into having at most one literal in the consequence of any implication. The resolution strategy used is linear-input resolution: that is, for each resolvent, one of the parents is either in the initial database

or is an ancestor of the other parent. The search strategy used is backward-chaining; the reasoner backchains from the query or goal, against the sentences in the logic program.

The following are also true in the logic programming paradigm: there is a form of negation that is interpreted as negation-as-failure: that is, *not a* will be taken to be true if *a* cannot be proven; and the result of a logic program can depend on the ordering of its clauses and subgoals. Prolog implementations provide additional control mechanisms, including the cut and fail operators; the result is that few programs in Prolog are pure realizations of the declarative paradigm. Prolog also has an incomplete mechanism for unification, particularly of arithmetic expressions.

Prolog has been widely used in developing expert systems, especially in Europe and Japan, although languages such as Java and C++ have become more popular.

Examples of successful practical applications of logic programming include the HAPPS system for model house configuration [83] and the Munich Rent Advisor [90], which calculates the estimated fair rent for an apartment. (This is a rather complex operation that can take days to do by hand.) There has been special interest in the last decade on world-wide web applications of logic programming (see *Theory and Practice of Logic Programming*, vol. 1, no. 3).

What are the drawbacks to Prolog? Why is there continued interest in the significantly less efficient general theorem provers?

First, the restriction to Horn clause form is rather severe; one may not be able to express knowledge crucial for one's application. An implication whose conclusion is a disjunction is not expressible in Horn clause form. This means, for example, that one cannot represent a rule like

> *If you are diagnosed with high-blood pressure, you will either have to reduce your salt intake or take medication*

because that is most naturally represented as an implication with a disjunction in the consequent.

Second, Prolog's depth-first-search strategy is incomplete.

Third, because, in most current Prolog implementations, the results of a Prolog program depend crucially on the ordering of its clauses, and because it is difficult to predict how the negation-as-failure mechanism will interact with one's knowledge base and goal query, it may be difficult to predict a program's output.

Fourth, since Prolog does not support inference with equality, it cannot be used for mathematical theorem proving.

There has been interest in the logic programming community in addressing limitations or perceived drawbacks of Prolog. Disjunctive logic programming [6] allows clauses with a disjunction of literals in the consequent of a rule. Franconi et al. [88] discusses one application of disjunctive logic programming, the implementation of a clean-up procedure prior to processing census data.

The fact that logic programs may have unclear or ambiguous semantics has concerned researchers for decades. This has led to the development of answer set programming, discussed in detail in Chapter 7, in which logic programs are interpreted with the stable model semantics. Answer set programming has been used for many applications, including question answering, computational biology, and system validation.

1.5 Suitability of Logic for Knowledge Representation

The central tenet of logicist AI[5]—that knowledge is best represented using formal logic—has been debated as long as the field of knowledge representation has existed. Among logicist AI's strong advocates are John McCarthy [171, 175], Patrick Hayes [112, 114, 111], and Robert Moore [186]; critics of the logicist approach have included Yehoshua Bar-Hillel [21], Marvin Minsky [185], Drew McDermott [180], and Rodney Brooks [42]. (McDermott can be counted in both the logicist and anti-logicist camps, having advocated for and contributed to logicist AI [178, 181, 179] before losing faith in the enterprise.)

The crux of the debate is simply this: Logicists believe that first-order logic, along with its modifications, is a language particularly well suited to capture reasoning, due to its expressivity, its model-theoretic semantics, and its inferential power. Note [112] that it is not a particular syntax for which logicists argue; it is the notion of a formal, declarative semantics and methods of inference that are important. (See [95, 64, 233, 39] for examples of how AI logicism is used.) Anti-logicists have argued that the program, outside of textbook examples, is undesirable and infeasible. To paraphrase McDermott [180], You Don't Want To Do It, and You Can't Do It Anyway.

This handbook clearly approaches AI from a logicist point of view. It is nevertheless worthwhile examining the debate in detail. For it has not consisted merely of an ongoing sequence of arguments for and against a particular research approach. Rather, the arguments of the anti-logicists have proved quite beneficial for the logicist agenda. The critiques have often been recognized as valid within the logicist community; researchers have applied themselves to solving the underlying difficulties; and in the process have frequently founded productive subfields of logicist AI, such as nonmonotonic reasoning. Examining the debate puts into context the research in knowledge representation that is discussed in this Handbook.

1.5.1 Anti-logicist Arguments and Responses

In the nearly fifty years since McCarthy's Advice Taker paper first appeared [171], the criticisms against the logicist approach have been remarkably stable. Most of the arguments can be characterized under the following categories:

- Deductive reasoning is not enough.
- Deductive reasoning is too expensive.
- Writing down all the knowledge (the right way) is infeasible.
- Other approaches do it better and/or cheaper.

The argument: Deductive reasoning is not enough

McCarthy's original logicist proposal called for the formalization of a set of commonsense facts in first-order logic, along with an automated theorem prover to reason with

[5] The term *logicism* generally refers to the school of thought that mathematics can be reduced to logic [270], *logicists* to the proponents of logicism. Within the artificial intelligence community, however, a *logicist* refers to a proponent of logicist AI, as defined in this section [257].

those facts. He gave as an example the reasoning task of planning to get to the airport. McCarthy argued that starting out from facts about first, the location of oneself, one's car, and the airport; second, how these locations relate to one another; third, the feasibility of certain actions, such as walking and driving; fourth, the effects that actions had; and fifth, basic planning constructs, one could deduce that to get to the airport, one should walk to one's car and drive the car to the airport. There were, all together, just 15 axioms in this draft formalization.

Bar-Hillel argued:

> It sounds rather incredible that the machine could have arrived at its conclusion—which, in plain English, is "Walk from your desk to your car!"—by sound deduction! This conclusion surely could not possibly follow from the premise in any serious sense. Might it not be occasionally cheaper to call a taxi and have it take you over to the airport? Couldn't you decide to cancel your flight or to do a hundred other things?

The need for nonmonotonic reasoning

In part, Bar-Hillel was alluding to the many exceptions that could exist in any realistically complex situation. Indeed, it soon became apparent to AI researchers that exceptions exist for even simple situations and facts. The classic example is that of reasoning that a bird can fly. Birds typically can fly, although there are exceptions, such as penguins and birds whose wings are broken. If one wants to formalize a theory of bird flying, one cannot simply write

$$\forall x(Bird(x) \rightarrow Flies(x)) \tag{1.17}$$

because that would mean that all birds fly. That would be wrong, because it does not take penguins and broken-winged birds into account. One could instead write

$$\forall x(Bird(x) \wedge \neg Penguin(x) \wedge \neg Brokenwinged(x) \rightarrow Flies(x)) \tag{1.18}$$

which says that all birds fly, as long as they are not penguins or broken-winged, or better yet, from the representational point of view, the following three formulas:

$$\forall x(Bird(x) \wedge \neg Ab(x) \rightarrow Flies(x)), \tag{1.19}$$

$$\forall x(Penguin(x) \rightarrow Ab(x)), \tag{1.20}$$

$$\forall x(Brokenwinged(x) \rightarrow Ab(x)) \tag{1.21}$$

which say that birds fly unless they are abnormal, and that penguins and broken-winged birds are abnormal.

A formula in the style of (1.18) is difficult to write, since one needs to state all possible exceptions to bird flying in order to have a correct axiom. But even aside from the representational difficulties, there is a serious inferential problem. If one only knows that Tweety is a bird, one cannot use axiom (1.18) in a deductive proof. One needs to know as well that the second and third conjuncts on the left-hand side of the implication are true: that is, that Tweety is not a penguin and is not broken-winged. Something stronger than deduction is needed here; something that permits jumping to the conclusion that Tweety flies from the fact that Tweety is a bird and the

absence of any knowledge that would contradict this conclusion. This sort of default reasoning would be *nonmonotonic* in the set of axioms: adding further information (e.g., that Tweety is a penguin) could mean that one has to retract conclusions (that is, that Tweety flies).

The need for nonmonotonic reasoning was noted, as well, by Minsky [185]. At the time Minsky wrote his critique, early work on nonmonotonicity had already begun. Several years later, most of the major formal approaches to nonmonotonic reasoning had already been mapped out [173, 224, 181]. This validated both the logicist AI approach, since it demonstrated that formal systems could be used for default reasoning, and the anti-logicists, who had from the first argued that first-order logic was too weak for many reasoning tasks.

Nonmonotonicity and the anti-logicists

From the time they were first developed, nonmonotonic logics were seen as an essential logicist tool. It was expected that default reasoning would help deal with many KR difficulties, such as the frame problem, the problem of efficiently determining which things remain the same in a changing world. However, it turned out to be surprisingly difficult to develop nonmonotonic theories that entailed the expected conclusions. To solve the frame problem, for example, one needs to formalize the *principle of inertia*—that properties tend to persist over time. However, a naive formalization of this principle along the lines of [174] leads to the *multiple extension* problem; a phenomenon in which the theory supports several models, some of which are unintuitive. Hanks and McDermott [110] demonstrated a particular example of this, the Yale shooting problem. They wrote up a simple nonmonotonic theory containing some general facts about actions (that loading a gun causes the gun to be loaded, and that shooting a loaded gun at someone causes that individual to die), the principle of inertia, and a particular narrative (that a gun is loaded at one time, and shot at an individual a short time after). The expected conclusion, that the individual will die, did not hold. Instead, Hanks and McDermott got multiple extensions: the expected extension, in which the individual dies; and an unexpected extension, in which the individual survives, but the gun mysteriously becomes unloaded. The difficulty is that the principle of inertia can apply either to the gun remaining loaded or the individual remaining alive. Intuitively we expect the principle to be applied to the gun remaining loaded; however, there was nothing in Hank's and McDermott's theory to enforce that.

The Yale shooting problem was not hard to handle: solutions began appearing shortly after the problem became known. (See [160, 161, 238] for some early solutions.) Nonetheless, the fact that nonmonotonic logics could lead to unexpected conclusions for such simple problems was evidence to anti-logicists of the infeasibility of logicist AI. Indeed, it led McDermott to abandon logicist AI. Nonmonotonic logic was essentially useless, McDermott argued [180], claiming that it required one to know beforehand what conclusions one wanted to draw from a set of axioms, and to build that conclusion into the premises.

In contrast, what logicist AI learned from the Yale shooting problem was the importance of a good underlying representation. The difficulty with Hanks and McDermott's axiomatization was not that it was written in a nonmonotonic logic; it was that it was devoid of a concept of causation. The Yale shooting problem does not arise in an axiomatization based on a sound theory of causation [243, 187, 237].

From today's perspective, the Yale shooting scenario is rather trivial. Over the last ten years, research related to the frame problem has concentrated on more elaborate kinds of action domains—those that include actions with indirect effects, nondeterministic actions, and interacting concurrently executed actions. Efficient implementations of such advanced forms of nonmonotonic reasoning have been used in serious industrial applications, such as the design of a decision support system for the Space Shuttle [198].

The current state of research on nonmonotonic reasoning and the frame problem is described in Chapters 6, 7, and 16–20 of this Handbook.

The need for abduction and induction

Anti-logicists have pointed out that not all commonsense reasoning is deductive. Two important examples of non-deductive reasoning are *abduction*, explaining the cause of a phenomenon, and *induction*, reasoning from specific instances of a class to the entire class. Abduction, in particular, is important for both expert and commonsense reasoning. Diagnosis is a form of abduction; understanding natural language requires abduction as well [122].

Some philosophers of science [215, 117, 118] have suggested that abduction can be grounded in deduction. The idea is to hypothesize or guess an explanation for a particular phenomenon, and then try to justify this guess using deduction. A well-known example of this approach is known as the deductive-nomological hypothesis.

McDermott [180] has argued against such attempts, pointing out what has been noted by philosophers of science [234]: the approach is overly simplistic, can justify trivial explanations, and can support multiple explanations without offering a way of choosing among candidates. But he was tilting at a straw man. In fact, the small part of logicist AI that has focused on abduction has been considerably more sophisticated in its approach. As discussed in the previous section, Hobbs, Stickel, and others have used theorem proving technology to support abductive reasoning [247, 122], but they do it by carefully examining the structure of the generated proofs, and the particular context in which the explanandum occurs. There is a deliberate and considered approach toward choosing among multiple explanations and toward filtering out trivial explanations.

There is also growing interest in *inductive logic programming* [189]. This field uses machine learning techniques to construct a logic program that entails all the positive and none of the negative examples of a given set of examples.

The argument: Deductive reasoning is too expensive

> The decisive question [is] how a machine, even assuming it will have somehow countless millions of facts stored in its memory, will be able to pick out those facts which will serve as premises for its deduction.
>
> *Yehoshua Bar-Hillel [21]*

When McCarthy first presented his Advice Taker paper and Bar-Hillel made the above remark, automated theorem proving technology was in its infancy: resolution theorem proving was still several years away from being invented. But even with relatively advanced theorem proving techniques, Bar-Hillel's point remains. General

automated theorem proving programs frequently cannot handle theories with several hundred axioms, let alone several million.

This point has in fact shaped much of the AI logicist research agenda. The research has progressed along several fronts. There has been a large effort to make general theorem proving more efficient (this is discussed at length in Section 1.3); special-purpose reasoning techniques have been developed, e.g., by the description logic community [11] as well as by Cyc (see Section 1.4.2) to determine subsumption and disjointness of classes; and logic programming techniques (for both Prolog (see Section 1.4.4) and answer set programming (see Chapter 7)) have been developed so that relatively efficient inferences can be carried out under certain restricted assumptions. The HPKB project and Cyc demonstrate that at least in some circumstances, inference is practical even with massively large knowledge bases.

The argument: Writing down all the knowledge (the right way) is infeasible

> Just constructing a knowledge base is a major intellectual research problem ... The problem of finding suitable axioms—the problem of "stating the facts" in terms of always-correct, logical, assumptions—is very much harder than is generally believed.
>
> *Marvin Minsky [185].*

The problem is in fact much greater than Minsky realized, although it has taken AI logicists a while to realize the severity of the underlying issues. At the time that Minsky wrote his paper, his critique on this point was not universally appreciated by proponents of AI logicism. The sense one gets from reading the papers of Pat Hayes [113, 114, 111],[6] for example, is one of confidence and optimism. Hayes decried the paucity of existing domain formalizations, but at the time seemed to believe that creating the formalizations could be done as long as enough people actually sat down to write the axioms. He proposed, for the subfield of naive physics, that a committee be formed, that the body of commonsense knowledge about the physical world be divided into clusters, with clusters assigned to different committee members, who would occasionally meet in order to integrate their theories.

But there never was a concerted effort to formalize naive physics. Although there have been some attempts to formalize knowledge of various domains (see, e.g., [123], and the proceedings of the various symposia on Logical Formalizations of Commonsense Knowledge), most research in knowledge representation remains at the metalevel. The result, as Davis [65] has pointed out, is that at this point constructing a theory that can reason correctly about simple tasks like staking plants in a garden is beyond our capability.

What makes it so difficult to write down the necessary knowledge? It is not, certainly, merely the writing down of millions of facts. The Cyc knowledge base, as discussed in Section 1.4, has over 3 million assertions. But that knowledge base is still missing the necessary information to reason about staking plants in a garden, cracking eggs into a bowl, or many other challenge problems in commonsense reasoning and knowledge representation [183]. Size alone will not solve the problem. That is why attempts to use various web-based technologies to gather vast amounts of knowledge [170] are irrelevant to this critique of the logicist approach.

[6] Although [111] was published in the 1980s, a preliminary version was first written in the late 1970s.

Rather, formalizing domains in logic is difficult for at least the following reasons:

- First, it is difficult to become aware of all our implicit knowledge; that is, to make this knowledge explicit, even in English or any other natural language. The careful examination of many domains or non-trivial commonsense reasoning problems makes this point clear. For example, reasoning about how and whether to organize the giving of a surprise birthday present [188] involves reasoning about the factors that cause a person to be surprised, how surprises can be foiled, joint planning, cooperation, and the importance of correct timing. The knowledge involved is complex and needs to be carefully teased out of the mass of social protocols that unknowingly govern our behavior.

- Second, as Davis [65] has pointed out, there is some knowledge that is difficult to express in any language. Davis gives the example of reasoning about a screw. Although it is easy to see that a small bump in the surface will affect the functionality of a screw much more than a small pit in the surface, it is hard to express the knowledge needed to make this inference.

- Third, there are some technical difficulties that prevent formalization of certain types of knowledge. For example, there is still no comprehensive theory of how agents infer and reason about other agents' ignorance (although [109] is an excellent start in this direction); this makes it difficult to axiomatize realistic theories of multi-agent planning, which depend crucially on inferring what other agents do and do not know, and how they make up for their ignorance.

- Fourth, the construction of an ontology for a domain is a necessary but difficult prerequisite to axiomatization. Deciding what basic constructs are necessary and how to organize them is a tricky enterprise, which often must be reworked when one starts to write down axioms and finds that it is awkward to formalize the necessary knowledge.

- Fifth, it is hard to integrate existing axiomatizations. Davis gives as an example his axiomatizations of string, and of cutting. There are various technical difficulties—mainly, assumptions that have been built into each domain axiomatization—that prevent a straightforward integration of the two axiomatizations into a single theory that could support simple inferences about cutting string. The problem of integration, in simpler form, will also be familiar to anyone who has ever tried to integrate ontologies. Concepts do not always line up neatly; how one alters these concepts in order to allow subsumption is a challenging task.

There have nonetheless been many successes in writing down knowledge correctly. The best known are the theories of causation and temporal reasoning that were developed in part to deal with the frame and Yale shooting problems. Other successful axiomatizations, including theories of knowledge and belief, multiple agency, spatial reasoning, and physical reasoning, are well illustrated in the domain theories in this Handbook.

The argument: Other approaches do it better and/or cheaper

> Anyone familiar with AI must realize that the study of knowledge representation—at least as it applies to the "commonsense" knowledge required for reading typical text such as newspapers—is not going anywhere fast. This subfield of AI has become notorious for the production of countless non-monotonic logics and almost as many logics of knowledge and belief, and none of the work shows any obvious application to actual knowledge-representation problems.
>
> *Eugene Charniak [54]*

During the last fifteen years, statistical learning techniques have become increasingly popular within AI, particularly for applications such as natural language processing for which classic knowledge representation techniques had once been considered essential. For decades, for example, it had been assumed that much background domain knowledge would be needed in order to correctly parse sentences. For instance, a sentence like *John saw the girl with the toothbrush* has two parses, one in which the prepositional phrase *with the toothbrush* modifies the phrase *John saw*, and one in which it modifies the noun phrase *the girl*. Background knowledge, however, eliminates the first parse, since people do not see with toothbrushes. (In contrast, both parses are plausible for the sentence *John saw the girl with the telescope*.) The difficulty with KR-based approaches is that it requires a great deal of knowledge to properly process even small corpora of sentences.

Statistical learning techniques offers a different paradigm for many issues that arise in processing language. One useful concept is that of *collocation* [166], in which a program learns about commonly occurring collocated words and phrases, and subsequently uses this knowledge in order to parse. This is particularly useful for parsing and disambiguating phonemes for voice recognition applications. A statistical learning program might learn, for example, that *weapons of mass destruction* are words that are collocated with a high frequency. If this knowledge is then fed into a voice recognition program, it could be used to disambiguate between the words *math* and *mass*. The words in the phrase *Weapons of math destruction* are collocated with a low frequency, so that interpretation becomes less likely.

Programs using statistical learning techniques have become popular in text-retrieval applications; in particular, they are used in systems that have performed well in recent TREC competitions [262–266]. Statistical-learning systems stand out because they are often cheaper to build. There is no need to painstakingly build tailor-made knowledge bases for the purposes of understanding a small corpora of texts.

Nevertheless, it is unlikely that statistical-learning systems will ever obviate the need for logicist AI in these applications. Statistical techniques can go only so far. They are especially useful in domains in which language is highly restricted (e.g., newspaper texts, the example cited by Charniak), and for applications in which deep understanding is not required. But for many true AI applications, such as story understanding and deep question-answering applications, deep understanding is essential.

It is no coincidence that the rising popularity of statistical techniques has coincided with the rise of the text-*retrieval* competitions (TREC) as opposed to the message-*understanding* competitions (MUC). It is also worth noting that the successful participants in HPKB relied heavily on classical logicist KR techniques [58].

In general, this pattern appears in other applications. Statistical learning techniques do well with low cost on relatively easy problems. However, hard problems remain resistant to these techniques. For these problems, logicist-KR-based techniques appear to work best.

This may likely mean that the most successful applications in the future will make use of both approaches. As with the other critiques discussed above, the logicist research agenda is once again being set and influenced by non-logicist approaches; ultimately, this can only serve to strengthen the applicability of the logicist approach and the success of logicist-based applications.

Acknowledgements

The comments of Eyal Amir, Peter Andrews, Peter Baumgartner, Ernie Davis, Esra Erdem, Joohyung Lee, Christopher Lynch, Bill McCune, Sheila McIlraith, J. Moore, Maria Paola Bonacina, J. Hsiang, H. Kirchner, M. Rusinowitch, and Geoff Sutcliffe contributed to the material in this chapter. The first author was partially supported by the National Science Foundation under Grant IIS-0412907.

Bibliography

[1] E. Amir and P. Maynard-Reid. Logic-based subsumption architecture. *Artificial Intelligence*, 153(1–2):167–237, 2004.

[2] E. Amir and S. McIlraith. Partition-based logical reasoning for first-order and propositional theories. *Artificial Intelligence*, 162(1–2):49–88, 2005.

[3] P.B. Andrews. Theorem proving via general matings. *Journal of the ACM*, 28:193–214, 1981.

[4] P.B. Andrews, M. Bishop, S. Issar, D. Nesmith, F. Pfenning, and H. Xi. TPS: A theorem proving system for classical type theory. *Journal of Automated Reasoning*, 16:321–353, 1996.

[5] P.B. Andrews and C.E. Brown. TPS: A hybrid automatic-interactive system for developing proofs. *Journal of Applied Logic*, 4:367–395, 2006.

[6] C. Aravindan, J. Dix, and I. Niemelä. Dislop: A research project on disjunctive logic programming. *AI Commun.*, 10(3–4):151–165, 1997.

[7] T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1–2):133–178, 2000.

[8] F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi, and P.F. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation, Applications*. Cambridge University Press, Cambridge, UK, 2003.

[9] F. Baader and W. Snyder. Unification theory. In A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning*, vol. I, pages 445–532. Elsevier Science, 2001 (Chapter 8).

[10] F. Baader, editor. *CADE-19, 19th International Conference on Automated Deduction*, Miami Beach, FL, USA, July 28–August 2, 2003. *Lecture Notes in Computer Science*, vol. 2741. Springer, 2003.

[11] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, Cambridge, England, 1998.

[12] L. Bachmair and D. Plaisted. Termination orderings for associative–commutative rewriting systems. *J. Symbolic Computation*, 1:329–349, 1985.

[13] L. Bachmair and N. Dershowitz. Commutation, transformation, and termination. In J.H. Siekmann, editor. *Proceedings of the Eighth International Conference on Automated Deduction*, pages 5–20, 1986.

[14] L. Bachmair, N. Dershowitz, and J. Hsiang. Orderings for equational proofs. In *Proceedings of the Symposium on Logic in Computer Science*, pages 346–357, 1986.

[15] L. Bachmair, N. Dershowitz, and D. Plaisted. Completion without failure. In H. Aït-Kaci and M. Nivat, editors. *Resolution of Equations in Algebraic Structures 2: Rewriting Techniques*, pages 1–30. Academic Press, New York, 1989.

[16] L. Bachmair and H. Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *J. Logic Comput.*, 4(3):217–247, 1994.

[17] L. Bachmair and H. Ganzinger. Resolution theorem proving. In Robinson and Voronkov [231], pages 19–99.

[18] L. Bachmair, H. Ganzinger, C. Lynch, and W. Snyder. Basic paramodulation. *Information and Computation*, 121(2):172–192, September 1995.

[19] L. Bachmair, H. Ganzinger, and A. Voronkov. Elimination of equality via transformation with ordering constraints. *Lecture Notes in Computer Science*, 1421:175–190, 1998.

[20] K. Baclawski, M.M. Kokar, R.J. Waldinger, and P.A. Kogut. Consistency checking of semantic web ontologies. In I. Horrocks and J.A. Hendler, editors. *International Semantic Web Conference*, *Lecture Notes in Computer Science*, vol. 2342, pages 454–459. Springer, 2002.

[21] Y. Bar-Hillel, J. McCarthy, and O. Selfridge. Discussion of the paper: Programs with common sense. In V. Lifschitz, editor. *Formalizing Common Sense*, pages 17–20. Intellect, 1998.

[22] H.G. Barrow. Verify: A program for proving correctness of digital hardware designs. *Artificial Intelligence*, 24(1–3):437–491, 1984.

[23] P. Baumgartner. FDPLL—A first-order Davis–Putnam–Logemann–Loveland procedure. In D. McAllester, editor. *CADE-17—The 17th International Conference on Automated Deduction*, vol. 1831, pages 200–219. Springer, 2000.

[24] P. Baumgartner and U. Furbach. PROTEIN: A PROver with a theory extension INterface. In *Proceedings of the Conference on Automated Deduction*, 1994.

[25] P. Baumgartner and C. Tinelli. The model evolution calculus. In F. Baader, editor. *CADE-19: The 19th International Conference on Automated Deduction*, *Lecture Notes in Artificial Intelligence*, vol. 2741, pages 350–364. Springer, 2003.

[26] J.G.F. Belinfante. Computer proofs in Gödel's class theory with equational definitions for composite and cross. *Journal of Automated Reasoning*, 22:311–339, 1999.

[27] C.G. Bell and A. Newell. *Computer Structures: Readings and Examples*. McGraw-Hill, 1971.

[28] W. Bibel. *Automated Theorem Proving*. 2nd edition. Vieweg, Braunschweig/Wiesbaden, 1987.

[29] J.-P. Billon. The disconnection method. In P. Miglioli, U. Moscato, D. Mundici, and M. Ornaghi, editors. *Proceedings of TABLEAUX-96*, *Lecture Notes in Artificial Intelligence*, vol. 1071, pages 110–126. Springer, 1996.

[30] G. Birkhoff. On the structure of abstract algebras. *Proc. Cambridge Philos. Soc.*, 31:433–454, 1935.

[31] M. Bishop. A breadth-first strategy for mating search. In H. Ganzinger, editor. *CADE-16: Proceedings of the 16th International Conference on Automated Deduction Trento, Italy, 1999*, *Lecture Notes in Artificial Intelligence*, vol. 1632, pages 359–373. Springer-Verlag, 1999.

[32] M. Bishop and P.B. Andrews. Selectively instantiating definitions. In *Proceedings of the 15th International Conference on Automated Deduction*, pages 365–380, 1998.

[33] A. Bockmayr and V. Weispfenning. Solving numerical constraints. In A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning*, vol. 1, pages 751–842. Elsevier, Amsterdam, The Netherlands, January 2001 (Chapter 12).

[34] M.P. Bonacina. On the reconstruction of proofs in distributed theorem proving: a modified clause-diffusion method. *J. Symbolic Comput.*, 21(4):507–522, 1996.

[35] J. Bos and K. Markert. Recognising textual entailment with logical inference. In *HLT/EMNLP*. The Association for Computational Linguistics, 2005.

[36] R. Boyer, M. Kaufmann, and J. Moore. The Boyer–Moore theorem prover and its interactive enhancement. *Computers and Mathematics with Applications*, 29(2):27–62, 1995.

[37] R. Boyer, E. Lusk, W. McCune, R. Overbeek, M. Stickel, and L. Wos. Set theory in first-order logic: Clauses for Gödel's axioms. *Journal of Automated Reasoning*, 2:287–327, 1986.

[38] R. Boyer and J. Moore. *A Computational Logic*. Academic Press, New York, 1979.

[39] R.J. Brachman and H.J. Levesque. *Knowledge Representation and Reasoning*. Morgan Kaufmann, 2004.

[40] D. Brand. Proving theorems with the modification method. *SIAM J. Comput.*, 4:412–430, 1975.

[41] I. Bratko. *Prolog Programming for Artificial Intelligence*. 3rd edition. Addison-Wesley, 2000.

[42] R.A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47(1–3):139–159, 1991.

[43] R. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[44] B. Buchberger. Gröbner bases: An algorithmic method in polynomial ideal theory. In N.K. Bose, editor. *Multidimensional Systems Theory*, pages 184–232. Reidel, 1985.

[45] A. Bundy. *The Computer Modelling of Mathematical Reasoning*. Academic Press, New York, 1983.

[46] A. Bundy. The automation of proof by mathematical induction. In A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning*, vol. I, pages 845–911. Elsevier Science, 2001 (Chapter 13).

[47] A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-Level Guidance for Mathematical Reasoning*. *Cambridge Tracts in Theoretical Computer Science*, vol. 56. Cambridge University Press, 2005.

[48] J. Burch, E. Clarke, K. McMillan, D. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[49] M. Burrows, M. Abadi, and R.M. Needham. Authentication: A practical study in belief and action. In M.Y. Vardi, editor. *TARK*, pages 325–342. Morgan Kaufmann, 1988.

[50] S. Buvac. Resolving lexical ambiguity using a formal theory of context. In K. van Deemter and S. Peters, editors. *Semantic Ambiguity and Underspecification*. Center for the Study of Language and Information, Stanford, 1996.

[51] R. Caferra, A. Leitsch, and N. Peltier. *Automated Model Building*. Kluwer Academic Publishers, 2004.

[52] B.F. Caviness and J.R. Johnson, editors. *Quantifier Elimination and Cylindrical Algebraic Decomposition*. Springer-Verlag, New York, 1998.

[53] C. Chang and R. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, New York, 1973.

[54] E. Charniak. *Statistical Language Learning*. MIT Press, 1993.

[55] S.C. Chou and X.S. Gao. Automated reasoning in geometry. In A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning*, vol. I, pages 707–749. Elsevier Science, 2001 (Chapter 11).

[56] A. Church. A note on the Entscheidungsproblem. *Journal of Symbolic Logic*, 1:40–41, 1936. Correction, ibid., 101–102.

[57] K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors. *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

[58] P.R. Cohen, R. Schrag, E.K. Jones, A. Pease, A. Lin, B. Starr, D. Gunning, and M. Burke. The DARPA high-performance knowledge bases project. *AI Magazine*, 19(4):25–49, 1998.

[59] H. Comon. Inductionless induction. In A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning*, vol. I, pages 913–962. Elsevier Science, 2001 (Chapter 14).

[60] H. Comon and F. Jacquemard. Ground reducibility is EXPTIME-complete. In *Proc. 12th IEEE Symp. Logic in Computer Science (LICS'97)*, Warsaw, Poland, June–July 1997, pages 26–34. IEEE Comp. Soc. Press, 1997.

[61] H. Comon, P. Narendran, R. Nieuwenhuis, and M. Rusinowitch. Deciding the confluence of ordered term rewrite systems. *ACM Trans. Comput. Logic*, 4(1):33–55, 2003.

[62] R.L. Constable, et al. *Implementing Mathematics with the NuPrl Proof Development System*. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[63] M. Dauchet. Simulation of Turing machines by a left-linear rewrite rule. In *Proceedings of the 3rd International Conference on Rewriting Techniques and Applications*, *Lecture Notes in Computer Science*, vol. 355, pages 109–120. Springer, 1989.

[64] E. Davis. *Representations of Commonsense Knowledge*. Morgan Kaufmann, San Francisco, CA, 1990.

[65] E. Davis. The naive physics perplex. *AI Magazine*, 19(3):51–79, 1998.

[66] M. Davis. Eliminating the irrelevant from mechanical proofs. In: *Proceedings Symp. of Applied Math.* vol. 15, pages 15–30, 1963.

[67] M. Davis. The prehistory and early history of automated deduction. In J. Siekmann and G. Wrightson, editors. *Automation of Reasoning*, vol. 1. Springer-Verlag, Berlin, 1983.
[68] M. Davis. First order logic. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors. *Handbook of Logic in AI and Logic Programming*, vol. 1, pages 31–65. Oxford University Press, 1993.
[69] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.
[70] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.
[71] A. Degtyarev and A. Voronkov. The inverse method. In A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning*, vol. I, pages 179–272. Elsevier Science, 2001 (Chapter 4).
[72] E. Deplagne, C. Kirchner, H. Kirchner, and Q.H. Nguyen. Proof search and proof check for equational and inductive theorems. In Baader [10], pages 297–316.
[73] N. Dershowitz. On representing ordinals up to γ_0. Unpublished note, 1980.
[74] N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17:279–301, 1982.
[75] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Comput.*, 3:69–116, 1987.
[76] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor. *Handbook of Theoretical Computer Science*. North-Holland, Amsterdam, 1990.
[77] N. Dershowitz and D.A. Plaisted. Rewriting. In A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning*, vol. I, pages 535–610. Elsevier Science, 2001 (Chapter 9).
[78] N. Dershowitz, J. Hsiang, N. Josephson, and D.A. Plaisted. Associative–commutative rewriting. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, pages 940–944, August 1983.
[79] E. Domenjoud. AC-unification through order-sorted AC1-unification. In *Proceedings of the 4th International Conference on Rewriting Techniques and Applications*, *Lecture Notes in Computer Science*, vol. 488. Springer-Verlag, 1991.
[80] E. Domenjoud. Number of minimal unifiers of the equation $\alpha x_1 + \cdots + \alpha x_p =_{AC} \beta y_1 + \cdots + \beta y_q$. *Journal of Automated Reasoning*, 8:39–44, 1992.
[81] P.J. Downey, R. Sethi, and R. Tarjan. Variations on the common subexpression problem. *Journal of the ACM*, 27(4):758–771, 1980.
[82] J. Endrullis, J. Waldmann, and H. Zantema. Matrix interpretations for proving termination of term rewriting. In Furbach and Shankar [91], pages 574–588.
[83] R.S. Engelmore. Knowledge-based systems in Japan, 1993. http://www.wtec.org/loyola/kb/.
[84] D. Fensel and A. Schönegge. Specifying and verifying knowledge-based systems with KIV. In J. Vanthienen and F. van Harmelen, editors. *EUROVAV*, pages 107–116. Katholieke Universiteit Leuven, Belgium, 1997.
[85] R. Fikes and N.J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3–4):189–208, 1971.
[86] B. Fischer, J. Schumann, and G. Snelting. Deduction-based software component retrieval. In W. Bibeland and P.H. Schmitt, editors. *Automated Deduction: A Basis for Applications*, vol. 3. Kluwer Academic, 1998.

[87] M. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer-Verlag, New York, 1990.

[88] E. Franconi, A.L. Palma, N. Leone, S. Perri, and F. Scarcello. Census data repair: a challenging application of disjunctive logic programming. In R. Nieuwenhuis and A. Voronkov, editors. *LPAR*, *Lecture Notes in Computer Science*, vol. 2250, pages 561–578. Springer, 2001.

[89] G. Frege. *Begriffsschrift, eine der arithmetischen nachgebildete Formelschprache des reinen Denkens*. Halle, 1879. English translation: [261, pp. 1–82].

[90] T.W. Frühwirth and S. Abdennadher. The Munich rent advisor: A success for logic programming on the Internet. *TPLP*, 1(3):303–319, 2001.

[91] U. Furbach and N. Shankar, editors. *Automated Reasoning, Third International Joint Conference, IJCAR 2006*, Seattle, WA, USA, August 17–20, 2006, Proceedings. *Lecture Notes in Computer Science*, vol. 4130. Springer, 2006.

[92] J.-M. Gaillourdet, T. Hillenbrand, B. Löchner, and H. Spies. The new Waldmeister loop at work. In Baader [10], pages 317–321.

[93] H. Ganzinger and K. Korovin. New directions in instantiation-based theorem proving. In *Proc. 18th IEEE Symposium on Logic in Computer Science, (LICS'03)*, pages 55–64. IEEE Computer Society Press, 2003.

[94] H. Gelernter, J.R. Hansen, and D.W. Loveland. Empirical explorations of the geometry theorem proving machine. In E. Feigenbaum and J. Feldman, editors. *Computers and Thought*, pages 153–167. McGraw-Hill, New York, 1963.

[95] M. Genesereth and N.J. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann, San Mateo, CA, 1987.

[96] G. Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 1935.

[97] J. Giesl and D. Kapur. Deciding inductive validity of equations. In Baader [10], pages 17–31.

[98] J. Giesl, P. Schneider-Kamp, and R. Thiemann. Automatic termination proofs in the dependency pair framework. In Furbach and Shankar [91], pages 281–286.

[99] P.C. Gilmore. A proof method for quantification theory. *IBM Journal of Research and Development*, 4:28–35, 1960.

[100] K. Gödel. Die Vollständigkeit fer Axiome des logischen Funktionenkalküls. *Monatshefte für Mathematik und Physik*, 37:349–360, 1930. English translation: [261, pp. 582–591].

[101] M.J. Gordon and T.F. Melham, editors. *Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.

[102] B.C. Grau, B. Parsia, E. Sirin, and A. Kalyanpur. Automatic partitioning of owl ontologies using-connections. In I. Horrocks, U. Sattler, and F. Wolter, editors. *Description Logics*, volume 147 of *CEUR Workshop Proceedings*, 2005.

[103] C.C. Green. *The Applications of Theorem Proving to Question-Answering Systems*. Garland, New York, 1969.

[104] C.C. Green. Application of theorem proving to problem solving. In *IJCAI*, pages 219–240, 1969.

[105] J.V. Guttag, D. Kapur, and D. Musser. On proving uniform termination and restricted termination of rewriting systems. *SIAM J. Comput.*, 12:189–214, 1983.

[106] R. Hähnle. Tableaux and related methods. In A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning*, vol. I, pages 100–178. Elsevier Science, 2001 (Chapter 3).

[107] A. Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–308, 1985.

[108] T.R. Halfhill. An error in a lookup table created the infamous bug in Intel's latest processor. *BYTE*, March 1995.

[109] J.Y. Halpern and G. Lakemeyer. Multi-agent only knowing. *J. Logic Comput.*, 11(1):41–70, 2001.

[110] S. Hanks and D.V. McDermott. Nonmonotonic logic and temporal projection. *Artificial Intelligence*, 33(3):379–412, 1987.

[111] P.J. Hayes. Naive physics I: Ontology for liquids. In J. Hobbs and R. Moore, editors. *Formal Theories of the Commonsense World*, pages 71–107. Ablex, Norwood, NJ, 1975.

[112] P.J. Hayes. In defence of logic. In *IJCAI*, pages 559–565, 1977.

[113] P.J. Hayes. The naive physics manifesto. In D. Michie, editor. *Expert Systems in the Microelectronic Age*. Edinburgh University Press, 1979.

[114] P.J. Hayes. The second naive physics manifesto. In J. Hobbs and R. Moore, editors. *Formal Theories of the Commonsense World*, pages 1–36. Ablex, Norwood, NJ, 1985.

[115] P.J. Hayes, T.C. Eskridge, R. Saavedra, T. Reichherzer, M. Mehrotra, and D. Bobrovnikoff. Collaborative knowledge capture in ontologies. In P. Clark and G. Schreiber, editors. *K-CAP*, pages 99–106. ACM, 2005.

[116] S. Heilbrunner and S. Hölldobler. The undecidability of the unification and matching problem for canonical theories. *Acta Informatica*, 24:157–171, 1987.

[117] C.G. Hempel. *Aspects of Scientific Explanation and Other Essays in the Philosophy of Science*. Free Press, 1965.

[118] C.G. Hempel and P. Oppeneheim. Studies in the logic of explanation. In C.G. Hempel, editor. *Aspects of Scientific Explanation and Other Essays in the Philosophy of Science*, pages 245–295. Free Press, 1965. Also includes 1964 postscript. Originally published in Philosophy of Science, 1948.

[119] J. Hendrix, J. Meseguer, and H. Ohsaki. A sufficient completeness checker for linear order-sorted specifications modulo axioms. In Furbach and Shankar [91], pages 151–155.

[120] N. Hirokawa and A. Middeldorp. Automating the dependency pair method. In Baader [10], pages 32–46.

[121] J.R. Hobbs. An overview of the TACITUS project. *Computational Linguistics*, 12(3), 1986.

[122] J.R. Hobbs, D.E. Appelt, J. Bear, D.J. Israel, M. Kameyama, M.E. Stickel, and M. Tyson. Fastus: A cascaded finite-state transducer for extracting information from natural-language text. *CoRR*, cmp-lg/9705013, 1997. Earlier version available as SRI Technical Report 519.

[123] J.R. Hobbs and R.C. Moore. *Formal Theories of the Commonsense World*. Ablex, 1985.

[124] J.R. Hobbs, M.E. Stickel, D.E. Appelt, and P.A. Martin. Interpretation as abduction. *Artificial Intelligence*, 63(1–2):69–142, 1993.

[125] J. Hsiang and M. Rusinowitch. Proving refutational completeness of theorem-proving strategies: the transfinite semantic tree method. *Journal of the ACM*, 38(3):559–587, July 1991.

[126] G. Huet. A complete proof of correctness of the Knuth–Bendix completion algorithm. *J. Comput. Systems Sci.*, 23(1):11–21, 1981.

[127] G. Huet and J.M. Hullot. Proofs by induction in equational theories with constructors. *Journal of Computer and System Sciences*, 25:239–266, 1982.

[128] G. Huet and D. Lankford. On the uniform halting problem for term rewriting systems. Technical Report Rapport Laboria 283, IRIA, Le Chesnay, France, 1978.

[129] F. Jacquemard, M. Rusinowitch, and L. Vigneron. Compiling and verifying security protocols. In *Logic Programming and Automated Reasoning*, pages 131–160, 2000.

[130] J.-P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. *SIAM J. Comput.*, 15:1155–1194, November 1986.

[131] J.-P. Jouannaud and P. Lescanne. On multiset orderings. *Information Processing Letters*, 15:57–63, 1982.

[132] J.-P. Jouannaud, P. Lescanne, and F. Reinig. Recursive decomposition ordering. In *Proceedings of the Second IFIP Workshop on Formal Description of Programming Concepts*, pages 331–348. North-Holland, 1982.

[133] J.-P. Jouannaud and C. Kirchner. Solving equations in abstract algebras: A rule-based survey of unification. In J.-L. Lassez and G. Plotkin, editors. *Computational Logic: Essays in Honor of Alan Robinson*. MIT Press, Cambridge, MA, 1991.

[134] J.-P. Jouannaud and E. Kounalis. Automatic proofs by induction in theories without constructors. *Inform. and Comput.*, 82(1):1–33, 1989.

[135] L. Kalmár. Zurückführung des Entscheidungsproblems auf den Fall von Formeln mit einer einzigen, bindren, Funktionsvariablen. *Compositio Mathematica*, 4:137–144, 1936.

[136] S. Kamin and J.-J. Levy. Two generalizations of the recursive path ordering. Unpublished, February 1980.

[137] D. Kapur and P. Narendran. Double-exponential complexity of computing a complete set of AC-unifiers. In *Proceedings 7th IEEE Symposium on Logic in Computer Science*, pages 11–21. Santa Cruz, CA, 1992.

[138] D. Kapur, P. Narendran, and H. Zhang. On sufficient completeness and related properties of term rewriting systems. *Acta Informatica*, 24:395–416, 1987.

[139] D. Kapur, G. Sivakumar, and H. Zhang. A new method for proving termination of AC-rewrite systems. In *Proc. of Tenth Conference on Foundations of Software Technology and Theoretical Computer Science*, *Lecture Notes in Comput. Sci.*, vol. 472, pages 133–148. Springer-Verlag, December 1990.

[140] D. Kapur and M. Subramaniam. Extending decision procedures with induction schemes. In D.A. McAllester, editor. *CADE-17: Proceedings of the 17th International Conference on Automated Deduction,* vol. 1831, pages 324–345. Springer-Verlag, London, UK, 2000.

[141] M. Kaufmann, P. Manolios, and J.S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Press, Boston, MA, 2000.

[142] M. Kaufmann, P. Manolios, and J.S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, Boston, MA, 2000.

[143] C. Kirchner, H. Kirchner, and M. Rusinowitch. Deduction with symbolic constraints. *Revue Francaise d'Intelligence Artificielle*, 4(3):9–52, 1990.
[144] D.E. Knuth and P.B. Bendix. Simple word problems in universal algebras. In J. Leech, editor. *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Oxford, 1970.
[145] A. Koprowski and H. Zantema. Automation of recursive path ordering for infinite labelled rewrite systems. In Furbach and Shankar [91], pages 332–346.
[146] K. Korovin and A. Voronkov. An AC-compatible Knuth–Bendix order. In Baader [10], pages 47–59.
[147] R.A. Kowalski. *Logic for Problem Solving*. North-Holland, Amsterdam, 1980.
[148] J.B. Kruskal. Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 95:210–225, 1960.
[149] D. Lankford. Canonical algebraic simplification in computational logic. Technical Report Memo ATP-25, Automatic Theorem Proving Project. University of Texas, Austin, TX, 1975.
[150] D. Lankford. On proving term rewriting systems are Noetherian. Technical Report Memo MTP-3, Mathematics Department, Louisiana Tech., University, Ruston, LA, 1979.
[151] D. Lankford and A.M. Ballantyne. Decision problems for simple equational theories with commutative-associative axioms: Complete sets of commutative-associative reductions. Technical Report Memo ATP-39, Department of Mathematics and Computer Science, University of Texas, Austin, TX, 1977.
[152] D. Lankford, G. Butler, and A. Ballantyne. A progress report on new decision algorithms for finitely presented abelian groups. In *Proceedings of the 7th International Conference on Automated Deduction*, *Lecture Notes in Computer Science*, vol. 170, pages 128–141. Springer, May 1984.
[153] S.-J. Lee and D. Plaisted. Eliminating duplication with the hyper-linking strategy. *Journal of Automated Reasoning*, 9(1):25–42, 1992.
[154] S.-J. Lee and D. Plaisted. Use of replace rules in theorem proving. *Methods of Logic in Computer Science*, 1:217–240, 1994.
[155] A. Leitsch. *The Resolution Calculus*. *Texts in Theoretical Computer Science*. Springer-Verlag, Berlin, 1997.
[156] D.B. Lenat. Cyc: A large-scale investment in knowledge infrastructure. *Communications of the ACM*, 38(11):32–38, 1995.
[157] D.B. Lenat and R.V. Guha. *Building Large Knowledge Based Systems: Representation and Inference in the Cyc Project*. Addison-Wesley, Reading, MA, 1990.
[158] R. Letz and G. Stenz. Model elimination and connection tableau procedures. In A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning*, vol. II, pages 2015–2114. Elsevier Science, 2001 (Chapter 28).
[159] V. Lifschitz. What is the inverse method? *J. Autom. Reason.*, 5(1):1–23, 1989.
[160] V. Lifschitz. Pointwise circumscription: Preliminary report. In *AAAI*, pages 406–410, 1986.
[161] V. Lifschitz. Formal theories of action (preliminary report). In *IJCAI*, pages 966–972, 1987.
[162] D. Loveland. A simplified format for the model elimination procedure. *Journal of the ACM*, 16:349–363, 1969.

[163] D. Loveland. *Automated Theorem Proving: A Logical Basis*. North-Holland, New York, 1978.
[164] D.W. Loveland. Automated deduction: looking ahead. *AI Magazine*, 20(1):77–98, Spring 1999.
[165] B. MacCartney, S.A. McIlraith, E. Amir, and T.E. Uribe. Practical partition-based theorem proving for large knowledge bases. In G. Gottlob and T. Walsh, editors. *IJCAI*, pages 89–98. Morgan Kaufmann, 2003.
[166] C. Manning and H. Schutze. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.
[167] A. Martelli and U. Montanari. An efficient unification algorithm. *Transactions on Programming Languages and Systems*, 4(2):258–282, April 1982.
[168] S.Ju. Maslov. An inverse method of establishing deducibilities in the classical predicate calculus. *Dokl. Akad. Nauk SSSR*, 159:1420–1424, 1964. Reprinted in SiekmannWrightson83a.
[169] C. Matuszek, J. Cabral, M.J. Witbrock, and J. DeOliviera. An introduction to the syntax and content of cyc. In *Proceedings of the AAAI 2006 Spring Symposium on Formalizing and Compiling Background Knowledge and its Applications to Knowledge Representation and Question Answering*, 2006.
[170] C. Matuszek, M.J. Witbrock, R.C. Kahlert, J. Cabral, D. Schneider, P. Shah, and D.B. Lenat. Searching for common sense: Populating cyc from the web. In M.M. Veloso and S. Kambhampati, editors. *AAAI*, pages 1430–1435. AAAI Press/The MIT Press, 2005.
[171] J. McCarthy. Programs with common sense. In *Proceedings of the Teddington Conference on the Mechanization of Thought Processes*, pages 75–91. London, 1959.
[172] J. McCarthy. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*. North-Holland, 1963.
[173] J. McCarthy. Circumscription: A form of non-monotonic reasoning. *Artificial Intelligence*, 13(1–2):23–79, 1980.
[174] J. McCarthy. Applications of circumscription to formalizing common sense knowledge. *Artificial Intelligence*, 26(3):89–116, 1986.
[175] J. McCarthy and P.J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors. *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, Edinburgh, 1969.
[176] W.W. McCune. Solution of the Robbins problem. *Journal of Automated Reasoning*, 19(3):263–276, December 1997.
[177] W. McCune and L. Wos. Otter—the CADE-13 competition incarnations. *J. Autom. Reason.*, 18(2):211–220, 1997.
[178] D.V. McDermott. Tarskian semantics, or no notation without denotation!. *Cognitive Science*, 2(3):277–282, 1978.
[179] D.V. McDermott. A temporal logic for reasoning about processes and plans. *Cognitive Science*, 6:101–155, 1982.
[180] D.V. McDermott. A critique of pure reason. *Computational Intelligence*, 3:151–160, 1987.
[181] D.V. McDermott and J. Doyle. Non-monotonic logic I. *Artificial Intelligence*, 13(1–2):41–72, 1980.
[182] A. Middeldorp. Modular properties of term rewriting systems. PhD thesis, Vrije Universiteit, Amsterdam, 1990.

[183] R. Miller and L. Morgenstern. The commonsense problem page, 1997. http://www-formal.stanford.edu/leora/commonsense.
[184] S. Miller and D.A. Plaisted. Performance of OSHL on problems requiring definition expansion. In R. Letz, editor. *7th International Workshop on First-Order Theorem Proving*, Koblenz, Germany, September 15–17, 2005.
[185] M. Minsky. A framework for representing knowledge. In P.H. Winston, editor. *The Psychology of Computer Vision*. McGraw-Hill, 1975. Also available as MIT-AI Lab Memo 306.
[186] R.C. Moore. The role of logic in knowledge representation and commonsense reasoning. In *AAAI*, pages 428–433, 1982.
[187] L. Morgenstern. The problems with solutions to the frame problem. In K.M. Ford and Z.W. Pylyshyn, editors. *The Robot's Dilemma Revisited*. Ablex, 1996.
[188] L. Morgenstern. A first-order axiomatization of the surprise birthday present problem: Preliminary report. In *Proceedings of the Seventh International Symposium on Logical Formalizations of Commonsense Reasoning*, 2005. Also published as Dresden Technical Report, ISSN 1430-211X.
[189] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *J. Logic Program.*, 19/20:629–679, 1994.
[190] G. Nelson and D.C. Oppen. Simplification by cooperating decision procedures. *ACM TOPLAS*, 1(2):245–257, 1979.
[191] G. Nelson and D.C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, 1980.
[192] M.H.A. Newman. On theories with a combinatorial definition of 'equivalence'. *Annals of Mathematics*, 43(2):223–243, 1942.
[193] R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning*, vol. I, pages 371–443. Elsevier Science, 2001 (Chapter 7).
[194] R. Nieuwenhuis and A. Rubio. Theorem proving with ordering and equality constrained clauses. *J. Symbolic Comput.*, 19(4):321–351, 1995.
[195] N.J. Nilsson. Shakey the robot. Technical Report 323. SRI International, 1984.
[196] T. Nipkow, G. Bauer, and P. Schultz. Flyspeck I: Tame graphs. In Furbach and Shankar [91], pages 21–35.
[197] T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer-Verlag, 2003.
[198] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry. An A-Prolog decision support system for the Space Shuttle. In *Proceedings of International Symposium on Practical Aspects of Declarative Languages (PADL)*, pages 169–183, 2001.
[199] E. Ohlebusch. *Advanced Topics in Term Rewriting*. Springer, New York, 2002.
[200] D.C. Oppen. Elementary bounds for Presburger Arithmetic. In *STOC'73: Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, pages 34–37. ACM Press, New York, NY, USA, 1973.
[201] S. Owrie, J.M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor. *Proceedings of the Eleventh Conference on Automated Deduction*, *Lecture Notes in Artificial Intelligence*, vol. 607, pages 748–752. Springer, June 1992.
[202] M. Paterson and M.N. Wegman. Linear unification. *J. Comput. System Sci.*, 16(2):158–167, 1978.

[203] L.C. Paulson. *Isabelle: A Generic Theorem Prover*. *Lecture Notes in Comput. Sci.*, vol. 828. Springer-Verlag, New York, 1994.
[204] G. Peano. Arithmetices principia, nova methodo exposita. Turin, 1889. English translation: [261, pp. 83–97].
[205] G.E. Peterson and M.E. Stickel. Complete sets of reductions for some equational theories. *Journal of the ACM*, 28(2):233–264, 1981.
[206] L. Pike. Formal verification of time-triggered systems. PhD thesis, Indiana University, 2005.
[207] D. Plaisted. A recursively defined ordering for proving termination of term rewriting systems. Technical report R-78-943, University of Illinois at Urbana-Champaign, Urbana, IL, 1978.
[208] D. Plaisted. Well-founded orderings for proving termination of systems of rewrite rules. Technical report R-78-932, University of Illinois at Urbana-Champaign, Urbana, IL, 1978.
[209] D. Plaisted. An associative path ordering. In *Proceedings of an NSF Workshop on the Rewrite Rule Laboratory*, pages 123–136, April 1984.
[210] D. Plaisted. Semantic confluence tests and completion methods. *Information and Control*, 65(2–3):182–215, 1985.
[211] D. Plaisted and S.-J. Lee. Inference by clause matching. In Z. Ras and M. Zemankova, editors. *Intelligent Systems: State of the Art and Future Directions*, pages 200–235. Ellis Horwood, West Sussex, 1990.
[212] D. Plaisted and Y. Zhu. *The Efficiency of Theorem Proving Strategies: A Comparative and Asymptotic Analysis*. Vieweg, Wiesbaden, 1997.
[213] D.A. Plaisted and Y. Zhu. Ordered semantic hyperlinking. *Journal of Automated Reasoning*, 25(3):167–217, October 2000.
[214] G. Plotkin. Building-in equational theories. In *Machine Intelligence*, vol. 7, pages 73–90. Edinburgh University Press, 1972.
[215] K. Popper. *The Logic of Scientific Discovery*. Hutchinson, London, 1959.
[216] E. Post. Introduction to a general theory of elementary propositions. *American Journal of Mathematics*, 43:163–185, 1921. Reproduced in [261, pp. 264–283].
[217] D. Prawitz. An improved proof procedure. *Theoria*, 26:102–139, 1960.
[218] QED Group. The QED manifesto. In A. Bundy, editor. *Proceedings of the Twelfth International Conference on Automated Deduction*, *Lecture Notes in Artificial Intelligence*, vol. 814, pages 238–251. Springer-Verlag, New York, 1994.
[219] A. Quaife. Automated deduction in von Neumann–Bernays–Gödel set theory. *Journal of Automated Reasoning*, 8:91–147, 1992.
[220] D. Ramachandran, P. Reagan, and K. Goolsbey. First-orderized researchcyc: Expressivity and efficiency in a common-sense ontology. Working Papers of the *AAAI Workshop on Contexts and Ontologies: Theory, Practice, and Applications*, 2005.
[221] D. Ramachandran, P. Reagan, K. Goolsbey, K. Keefe, and E. Amir. Inference-friendly translation of researchcyc to first order logic, 2005. Unpublished.
[222] I.V. Ramakrishnan, R. Sekar, and A. Voronkov. Term indexing. In A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning*, vol. II, pages 1853–1964. Elsevier Science, 2001 (Chapter 26).
[223] A.L. Rector. Modularisation of domain ontologies implemented in description logics and related formalisms including owl. In J.H. Gennari, B.W. Porter, and Y. Gil, editors. *K-CAP*, pages 121–128. ACM, 2003.

[224] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1–2):81–132, 1980.
[225] R. Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, 2001.
[226] A. Riazanov. Implementing an efficient theorem prover. PhD thesis, The University of Manchester, Manchester, July 2003.
[227] A. Riazanov and A. Voronkov. The design and implementation of VAMPIRE. *AI Communications*, 15(2–3):91–110, 2002.
[228] G. Robinson and L. Wos. Paramodulation and theorem-proving in first order theories with equality. In *Machine Intelligence*, vol. 4, pages 135–150. Edinburgh University Press, Edinburgh, Scotland, 1969.
[229] J. Robinson. Theorem proving on the computer. *Journal of the ACM*, 10:163–174, 1963.
[230] J. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.
[231] J.A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier/MIT Press, 2001.
[232] J.F. Rulifson, J.A. Derksen, and R.J. Waldinger. Qa4: A procedural calculus for intuitive reasoning. Technical Report 73, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, Nov. 1972.
[233] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. 2nd edition. Prentice-Hall, 2003.
[234] W.C. Salmon. *Four Decades of Scientific Explanation*. University of Minnesota Press, 1989.
[235] S. Schulz. E—a brainiac theorem prover. *AI Communications*, 15(2):111–126, 2002.
[236] R. Schwitter. English as a formal specification language. In *DEXA Workshops*, pages 228–232. IEEE Computer Society, 2002.
[237] M. Shanahan. *Solving the Frame Problem*. MIT Press, Cambridge, MA, 1997.
[238] Y. Shoham. Chronological ignorance: Time, nonmonotonicity, necessity and causal theories. In *AAAI*, pages 389–393, 1986.
[239] J. Siekmann. Unification theory. *Journal of Symbolic Computation*, 7:207–274, 1989.
[240] J. Siekmann, C. Benzmüller, and S. Autexier. Computer supported mathematics with Omega. *Journal of Applied Logic*, 4(4):533–559, 2006.
[241] D.R. Smith. KIDS: A knowledge-based software development system. In M. Lowry and R. McCartney, editors. *Automating Software Design*, pages 483–514. MIT Press, 1991.
[242] C. Sprenger, M. Backes, D.A. Basin, B. Pfitzmann, and M. Waidner. Cryptographically sound theorem proving. In *CSFW*, pages 153–166. IEEE Computer Society, 2006.
[243] L.A. Stein and L. Morgenstern. Motivated action theory: A formal theory of causal reasoning. *Artificial Intelligence*, 71(1):1–42, 1994.
[244] J. Steinbach. Extensions and comparison of simplification orderings. In *Proceedings of the 3rd International Conference on rewriting techniques and applications*, *Lecture Notes in Computer Science*, vol. 355, pages 434–448. Springer, 1989.

[245] G. Stenz and R. Letz. DCTP—a disconnection calculus theorem prover. In R. Gore, A. Leitsch, and T. Nipkow, editors. *Proc. of the International Joint Conference on Automated Reasoning*, *Lecture Notes in Artificial Intelligence*, vol. 2083, pages 381–385. Springer, 2001.

[246] M.E. Stickel. A Prolog technology theorem prover: Implementation by an extended Prolog compiler. *Journal of Automated Reasoning*, 4(4):353–380, 1988.

[247] M.E. Stickel. A Prolog-like inference system for computing minimum-cost abductive explanation in natural-language interpretation. *Annals of Mathematics and Artificial Intelligence*, 4:89–106, 1991.

[248] M.E. Stickel. A Prolog technology theorem prover: A new exposition and implementation in Prolog. *Theoretical Computer Science*, 104:109–128, 1992.

[249] M.E. Stickel, R.J. Waldinger, and V.K. Chaudhri. A guide to SNARK. Technical report, SRI International, 2000.

[250] M.E. Stickel, R.J. Waldinger, M.R. Lowry, T. Pressburger, and I. Underwood. Deductive composition of astronomical software from subroutine libraries. In A. Bundy, editor. *CADE*, *Lecture Notes in Computer Science*, vol. 814, pages 341–355. Springer, 1994.

[251] M.E. Stickel. A unification algorithm for associative–commutative functions. *J. of the ACM*, 28:423–434, 1981.

[252] M.E. Stickel. A Prolog technology theorem prover: Implementation by an extended Prolog compiler. In *Proceedings of the 8th International Conference on Automated Deduction*, pages 573–587, 1986.

[253] G. Sutcliffe. CASC-J3: The 3rd IJCAR ATP system competition. In U. Furbach and N. Shankar, editors. *Proc. of the International Joint Conference on Automated Reasoning*, *Lecture Notes in Artificial Intelligence*, vol. 4130, pages 572–573. Springer, 2006.

[254] G. Sutcliffe. The CADE-20 automated theorem proving competition. *AI Communications*, 19(2):173–181, 2006.

[255] C.B. Suttner and G. Sutcliffe. The TPTP problem library (TPTP v2.0.0). Technical Report AR-97-01, Institut für Informatik, Technische Universität München, Germany, 1997.

[256] Terese. *Term Rewriting Systems*. *Cambridge Tracts in Theoretical Computer Science*, vol. 55. Cambridge University Press, 2003.

[257] R. Thomason. Logic and artificial intelligence. In *Stanford Encyclopedia of Philosophy*. Stanford University, 2003.

[258] Y. Toyama. On the Church–Rosser property for the direct sum of term rewriting systems. *Journal of the ACM*, 34(1):128–143, January 1987.

[259] Y. Toyama, J.W. Klop, and H.-P. Barendregt. Termination for the direct sum of left-linear term rewriting systems. In *Proceedings of the 3rd International Conference on Rewriting Techniques and Applications*, *Lecture Notes in Computer Science*, vol. 355, pages 477–491. Springer, 1989.

[260] A. Trybulec and H. Blair. Computer aided reasoning with Mizar. In R. Parikh, editor. *Logic of Programs*, *Lecture Notes in Comput. Sci.*, vol. 193. Springer-Verlag, New York, 1985.

[261] J. van Heijenoort, editor. *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*. Harvard University Press, 1967.

[262] E.M. Voorhees and L.P. Buckland, editors. *The Eleventh Text Retrieval Conference*, 2002.

[263] E.M. Voorhees and L.P. Buckland, editors. *The Twelfth Text Retrieval Conference*, 2003.
[264] E.M. Voorhees and L.P. Buckland, editors. *The Thirteenth Text Retrieval Conference*, 2004.
[265] E.M. Voorhees and L.P. Buckland, editors. *The Fourteenth Text Retrieval Conference*, 2005.
[266] E.M. Voorhees and L.P. Buckland, editors. *The Fifteenth Text Retrieval Conference*, 2006.
[267] Y. Wang, P. Haase, and J. Bao. A survey of formalisms for modular ontologies. In *Workshop on Semantic Web for Collaborative Knowledge Acquisition*, 2007.
[268] C. Weidenbach. Combining superposition, sorts and splitting. In Robinson and Voronkov [231], pages 1965–2013.
[269] C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobald, and D. Topic. S pass version 2.0. In A. Voronkov, editor. *CADE*, *Lecture Notes in Computer Science*, vol. 2392, pages 275–279. Springer, 2002.
[270] A.N. Whitehead and B. Russell. *Principia Mathematica*. University Press, 1957. Originally published 1910–1913.
[271] L. Wos, R. Overbeek, E. Lusk, and J. Boyle. *Automated Reasoning: Introduction and Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1984.
[272] W.-T. Wu. On the decision problem and the mechanization of theorem proving in elementary geometry. *Scientia Sinica*, 21:159–172, 1978.
[273] P. Youn, B. Adida, M. Bon, J. Clulow, J. Herzog, A. Lin, R.L. Rivest, and R. Anderson. Robbing the bank with a theorem prover. Technical report 644, University of Cambridge Computer Laboratory, August 2005.