Computational logic and set theory

Jacob T. Schwartz

Domenico Cantone

Eugenio G. Omodeo

Foreword

A word on the audience for whom this book is intended.

Any technical book must, by emphasizing certain details and leaving others unspoken, make certain assumptions about the prior knowledge which its reader brings to its study. This book assumes that its reader has a good knowledge of standard programming techniques, particularly of string manipulation and parsing, and a general familiarity with those parts of mathematics which are analyzed in detail in the main series of definitions and proof scenarios to which much of its bulk is devoted. Less of knowledge of formal logic is assumed. For this reason we try to present needed ideas from logic in a reasonably self-contained way, emphasizing guiding ideas likely to be important in pragmatic extensions of the work begun here, rather than technicalities. Foundational issues, for example consideration of the strength or necessity of axioms, or the precise relationship of our formalism to other weaker or stronger formalisms studied in the literature, are neglected. Because we expect our readers to be programmers of some sophistication, syntactic details of the kind that often appear early in books on logic are underplayed, and we repeatedly assume that anything programmable with relative ease can be taken as routine, and that the properties of such programmable operators can be proved when necessary to some theoretical discussion. This reflects our feeling that understanding develops top-down, focusing on details only as these become necessary.

This belief implies that too much detail is more likely to impede than to promote understanding. Who reads, or would want to read, the Whitehead-Russell *Principia*, or to testify that its hundreds of formula-filled pages are without error? But since we ask this question, why do we include hundreds of formula-filled pages in this book, and hope to regard it as a successor to this very same *Principia*? The reason lies in the fact that our formal proof text is fully computer-checked. The CD_ROM accompanying this book, gives all these proofs in computer readable form, along with software capable of checking them. Though relatively useless to the human reader unless their correctness can be verified mechanically, long lists of formulae become useful once such verification has been achieved.

Chapter 1. Introduction

> *... This then is the advantage of our method: that immediately ... guided only by characters in a safe and really analytic way, we bring to light truths that others have barely achieved by an immense effort of mind and by chance. And therefore we are able to present results within our century which otherwise would hardly be attained in the course of millennia.*
>
> Gottfried Wilhelm Leibniz, 1679

1.1 Loomings

Logic begins with Aristotle's systematic enumeration of the forms of syllogism, as an attempt to improve the rigor of philosophical (and possibly also political) reasoning. Euclid then demonstrated that reasoning at Aristotle's syllogistic level of rigor could cover a substantial body of knowledge, namely the whole of geometry as known in his day. Subsequent mediaeval work, first in the Islamic world and later in Europe, began to uncover new algebraic forms of symbolic reasoning. Fifteen centuries after Euclid, Leibniz proposed that algebra be extended to a larger symbolism covering all rigorous thought. So two basic demands, for rigor and for extensive applicability, are fundamental to logic.

Leibniz did little to advance his proposal, which only began to move forward with the much later work of Boole (on the algebra of propositions), the 1879 Concept-Notation (*Begriffsschrift*) of Frege, and Peano's axiomatization of the foundations of arithmetic. This stream of work reached a pinnacle in Whitehead and Russell's 1910 demonstration that the whole corpus of mathematics could be covered by an improved Frege-like logical system.

Developments in mathematics had meanwhile prepared the ground for the Whitehead-Russell work. Mathematics can be seen as the combination of two forms of thought. Of these, the most basic is intuitive, and, as shown by geometry (or more primitively by arithmetic), often inspired by experience with the physical world which it captures and abstracts. But mathematics works on this material by systematically manipulating collections of statements about it. Thus the second face of mathematics is linguistic and formal. Mathematics attains rigor by demanding that the statement sequences which it admits as proofs conform to rigid formal constraints. For this to be possible, the pre-existing, intuition-inspired content of mathematics must be progressively resolved into carefully formalized concepts, and thus ultimately into sentences which a Leibniz-like formal logical language can cover. A major step in this analysis was Descartes reduction, via his coordinate method, of 2- and 3-dimensional geometry to algebra. To complete this, it became necessary to solve a nagging technical problem, the 'problem of the continuum', concerning the system of numbers used. An intuition basic to certain types of geometric reasoning is that no continuous curve can cross from one side of a line to another without intersecting the line in at least one point. To capture this principle in an algebraic model of the whole of geometry one must give a formal definition of the system of 'real' numbers which models the intuitively conceived real axis, must top this by giving a formal definition of the notion of continuity, and must use this definition to prove the fundamental theorem that a continuous function cannot pass from a positive to a negative value without becoming zero somewhere between.

This work was accomplished gradually during the 19th century. The necessary definition of continuity appeared in Cauchy's *Cours d'Analyse* of 1821. A formal definition of the system of 'real' numbers rigorously completing Cauchy's work was given in Dedekind's 1872 study *Continuity and Irrational Numbers*. Together these two efforts showed that the whole of classical calculus could be based on the system of fractions, and so, by a short step, on the whole numbers. What remained was to analyze the notion of number itself into something more fundamental. Such an analysis, of the notion of number into that of sets of arbitrary objects standing in 1-1 correspondence, appeared in Frege's 1884 *Foundations of Arithmetic*, was generalized and polished in Cantor's transfinite set theory of 1895, and was approached in alternative, more conventionally axiomatic terms by Peano in his 1894 *Mathematical Formulary*. Like Whitehead and Russell's *Principia Mathematica*, the series of definitions and theorems found later in this work walks the path blazed by Cauchy, Dedekind, Frege, Cantor, and Peano.

As set theory evolved, its striving for ultimate generality came to be limited by certain formal paradoxes, which become unavoidable if the doors of formal set-theoretic definition are opened too widely. These arise very simply. Suppose, for example, that we allow ourselves to consider 'the set of all sets that are not

members of themselves'. In a formal notation very close to that continually used below, this is simply s = {x: x notin x}. But now consider the proposition 's in s'. On formal grounds this is equivalent to 's in {x: x notin x}', and so by the very definition of set membership to the proposition 's notin s'. So in these few formal steps we have derived the proposition

s in s *eq s notin s,

a situation around which no coherent logical system can be built. The means adopted to avoid this immediate collapse of the formal structure that one wants to build is to restrict the syntax of the set-formers which can legally be written, in a way which forbids constructions like {x: x notin x} without ruling out the similar but somewhat more limited expressions needed to express the whole of standard mathematics. These fine adjustments to the formal structure of logic were worked out, first by Russell and Whitehead, later and a bit differently by their successors.

A higher technical polish was put on all this work by 20th century efforts. Cantor's work was extended, and began to be formalized, by Zermelo in 1908, and more completely formalized by Fraenkel in 1923. The axiomatization of set theory at which they arrived is called Zermelo-Fraenkel set theory. Starting in 1905 the great German mathematician David Hilbert began the influential series of studies of the algebra of logic, later summarized in his 1939 work *Foundations of Mathematics* (with Paul Bernays). First in his 1925 paper 'An Axiomatization of Set Theory', and then in a fuller 1928 version, John von Neumann elegantly recast the Zermelo-Fraenkel set formalism, along with Frege's analysis of the concept of number, by encoding the integers set-theoretically: the number 0 as the empty set, 1 as the singleton-set {0}, 2 as the set {0,1}, and more generally each integer n as the n-element set {0,1,...,n - 1}. A corresponding, equally elegant definition of the notions of ordinal and cardinal numbers (both finite and infinite) was given in von Neumann's carefully honed formalism, from which the more computer-oriented exposition found later in the present work derives very closely.

Especially at first, Hilbert's logical studies stood in a positive relation to the program proposed by Leibniz, since it was hoped that close analysis of the algebra of logic might in principle lead to a set of algorithms allowing any mathematical statement to be decided by a suitable calculation. But the radical attack on the intuitive soundness of non-constructive Cantorian reasoning and of the conventional foundations of mathematics published by the Dutch mathematician L.E. J. Brouwer in 1918 led Hilbert's work in a different direction. Hilbert hoped that the 'meta-mathematical' tools he was developing could be used to reply to Brouwer's critique. For this reply, a combinatorial analysis of the algebra of logic, to which Brouwer could have no objections since only constructive arguments would be involved, would be used metamathematically to demonstrate formal limits on what could be proved within standard mathematics, and in particular to show that no contradiction could follow from any standard proof. Once done, this would demonstrate the formal consistency of standard mathematics within a Brouwerian framework. But things turned out differently. In a startling and fundamentally new development, the metamathematical techniques pioneered by the Hilbert school were used in 1931 by Kurt Gödel to show that Hilbert's program was certainly unrealizable, since no logical system of the type considered by Hilbert could be used to prove its own consistency. The brilliance of this result changed the common professional view of logic, which came to be seen, not as a Leibnizian engine for the formal statement and verification of ordinary mathematics, but as a negatively-oriented tool for proving various qualitative and quantitative limits on the power of formalized mathematical systems.

In the late 1940's the coming of the computer brought in new influences. Expression in a rigorously defined system of formulae makes mathematics amenable to computer processing, and daily work with computer programs makes the absolute rigor of formalized mathematical systems obvious. The possibility of using computer assistance to lighten the tedium (so evident in Russell and Whitehead) of fully formalized proof began to make the Leibniz program seem more practical. (Initially it was even hoped

that suitably pruned computer searches could be used rather directly to find many of the ordinary proofs used in mathematics). The fact that the methods of formalized proof could be used to check and verify the correctness of computer programs gave economic importance to what would otherwise remain an esoteric endeavor. Computerized proof verifier systems, emphasizing various styles of proof and potential application areas, began to appear in the 1960's. The system described in the present text belongs to this stream of work.

A fully satisfactory formal logical system should be able to digest 'the whole of mathematics', as this develops by progressive extension of mathematics-like reasoning to new domains of thought. To avoid continual reworking of foundations, one wants the formal system taken as basic to remain unchanged, or at any rate to change only by extension as such efforts progress. In any fundamentally new area work and language will initially be controlled more by guiding intuitions than by precise formal rules, as when Euclid and his predecessors first realized that the intuitive properties of geometric figures in 2 and 3 dimensions, and also some familiar properties of whole numbers, could be covered by modes of reasoning more precise than those used in everyday life. Similarly, the initially semiformal languages that developed around studies of the 'complex' and 'imaginary' roots of algebraic equations, the 'infinitesimal' quantities spoken of in early versions of the calculus, the 'random' quantities of the probabilist, and the physicist's 'Dirac delta functions', all need to be absorbed into a single formal system. This is done by modeling the intuitively grasped objects appearing in important semi-formalized languages by precisely defined objects of the formal system, in a way that maps all the useful statements of the imprecise initial language into corresponding formulae. If less than vital, statements of the initial language that do not fit into its formalized version can then be dismissed as 'misunderstandings'.

The mathematical developments surveyed in the preceding discussion succeeded in re-expressing the intuitive content of geometry, arithmetic, and calculus ('analysis') in set-theoretic terms. The geometric notion of 'space' maps into 'set of all pairs (or triples) of real numbers', preparing the way for consideration of the 'set of all n-tuples of real numbers', as 'n-dimensional space', and of more general related constructs as 'infinite dimensional' and 'functional' spaces. The 'figures' originally studied in geometry map, via the 'locus' concept, into sets of such pairs, triples, etc. The next necessary step is to analyze the notion of real number into something more basic, the essential technical requirement for this being to ensure that no function roots (e.g. Pythagoras' square root of 2) are 'missing'. As noted above, this was accomplished by Dedekind, who reduced 'real number x' to 'nonempty set x of rational numbers, bounded above, such that every rational not in x is larger than every rational in x'. To eliminate everything but set theory from the formal foundations of mathematics, it only remains (since 'fractions' can be seen as pairs of numbers) to reduce the notion of 'integer' to set-theoretic terms. This was done by Cantor and Frege: an integer is the class of all finite sets in 1-1 correspondence with any one such set. None of the other important mathematical developments enumerated in the preceding paragraph required fundamental extension of the set-theoretic foundation thereby attained. Gauss realized that the 'complex' numbers used in algebra could be modeled as pairs of real numbers, Kolmogrorov modeled 'random' variables as functions defined on an implicit set-theoretic measure space, and Laurent Schwartz interpreted the initially puzzling 'delta functions' in terms of a broader notion of generalized function defined systematically in set theoretic terms. So all of these concepts were digested without forcing any adjustment of the set-theoretic foundation constructed for arithmetic, analysis, and geometry. This foundation also supports all the more abstract mathematical constructions elaborated in such 20th century fields as topology, abstract algebra, and category theory. Indeed, these were expressed set-theoretically from their inception. So (if we ignore a few ongoing explorations whose significance remains to be determined) set theory currently stands as a comfortable and universal basis for the whole of mathematics.

It can even be said that set theory captures a set of reality-derived intuitions more fundamental than such basic mathematical ideas as that of number. Arithmetic would be very different if the real-world process

of counting did not return the same result each time a set of objects was counted, or if a subset of a finite set S of objects proved to have a larger count than S. So, even though Peano showed how to characterize the integers and derive many of their properties using axioms free of any explicit set-theoretic content, his approach robs the integers of much of their intuitive significance, since in his reduced context they cannot be used to count anything. For this and the other reasons listed above, we will work with a thoroughly set-theoretic formalism, contrived to mimic the language and procedures of standard mathematics closely.

The special nature of mathematical reasoning within human reason in general

The syllogistic patterns characteristic of mathematical reasoning derive from, and thus often reappear in, other reasoned forms of human discourse, for example in arguments offered by lawyers and philosophers. Mathematical reasoning is distinguished within this world of reason by its rigorous adherence to the pattern originally set by Euclid. Some fixed set of statements, the *axioms,* perhaps carrying some insight about an observed or intuited world, must be firmly set down. Certain named predicates (and perhaps also function symbols) will appear in these axioms. The ensuing discourse (which may be lengthy) must work exclusively with properties of these predicates (and symbols) which follow formally from the axioms, precisely as if these predicates had no meaning other than that which the axioms give them. When new vocabulary is introduced (as will generally be necessary to provide intellectual variety and sustain interest) this must be by formal definition in terms of predicates (and function symbols) which either are those found in the axioms or which appear earlier in the discourse. Such extensions of vocabulary are subject to rules which ensure that all new symbols introduced can be regarded as tools of art which add nothing fundamental to the axioms. That is, mathematics' rules of definition ensure that allowed extensions of vocabulary cannot make it possible to prove any statement made in the original vocabulary that could not be proved, in the axioms' original vocabulary, from the axioms. This rule, which insists that definitions must be devoid of all hidden axiomatic content, is fundamental to mathematics. It will appear in our later technical discussions as the *conservative extension principle.*

Legal, philosophical, and scientific reasoning commonly fail to observe the rules which restrict mathematical discourse, since these styles of agument allow new terms with explicitly or implicitly assumed properties to be introduced far more freely. Science cannot avoid this, since it is dedicated to exploration of the world in all its variety, and must therefore speak of what it finds as best it can. But unconstrained introduction, into a line of reasoning, of even a few new terms having implicitly assumed properties can readily become an engine of deception (and of self-deception). Science tries to avoid such self-deception by taking all of its reasoned outcomes as provisional subject to comparison with observed reality. If observation conflicts with the outcome of a line of scientific reasoning, the assumptions and informal definitions entering into it will be adjusted until better agreement is attained. Legal and philosophical reasoning, lacking this mechanism, remain more permanently able to be used as engines of deception (perhaps deliberate) or of self-deception (which has its intellectual delights).

1.2 Proof verifiers

A *Proof verifier* is an interactive program for manipulation of the state of a mathematical discourse. It allows computer checking of such discourse in full detail, and collection of the resulting theorems for subsequent re-use. It must

(a) only allow theorems to be derived;

(b) allow all theorems to be derived.

Besides their theoretical interest, proof verifiers have one potential practical use: Program Verification.

To adapt a proof verifier to this use, we can simply annotate (ordinary procedural) programs with assertions A breaking every loop in their control flow. Then, for every path forward through the annotated program P and its assignments

$$x_1 := expn(x_1,...,x_n)$$

running from an assertion A_1 immediately before such an assignment to an assertion A_2 immediately after the assignment we must show that

$$(FORALL\ x_1,...,x_n \mid A_1(x_1,...,x_n)\ *imp\ A_2(expn(x_1,...,x_n),x_2,...,x_n))$$

holds. Once this has been done systematically throughout the program, we can be sure that the program is correct.

To give proofs acceptable to a programmed verifier, i.e. proofs every one of whose details can be checked by a computer, we must 'walk in shackles'; but then we want these shackles to be as light as possible. That is, we want the ordinary small steps of mathematical discourse to remain small, rather than expanding into tedious masses of detail. We aim for a formalized interactive conversation with the computer whose general 'feel' resembles that of ordinary mathematical exposition. The better we succeed in achieving this, the closer the verifier comes to passing the 'Turing test', at least in the restricted mathematical setting in which it is designed to operate. So the internal structure of a successful proof verifier can be seen as a model both of mathematics and of mathematical intelligence, which is an important, albeit limited, form of intelligence in general.

1.3 Informal introduction to the formalism in which we will work

A proof verifier must provide various tools. First of all, it must allow the elementary steps of proofs to be expressed by formulae in some agreed-on system. These formulae become the elementary steps which the system allows. The system-provided tools, which embody the system's 'deduction rules', must allow manipulation of these formulae in ways which mimic the normal flow of a mathematical discourse.

The collection of proofs presented to a verifier for validation is expressed as a sequence of logical formulae, to which we may attach formalized annotations to guide the action of the verifier. Given such a sequence of formulae, the verifier first checks all the statements presented to it for syntactic legality, and then goes on to verify the successive statements of each proof. As in ordinary proof, the verifier's user aims to guide discourse along paths which bring designated target theorems into the collection of proved statements. This is done by arranging the formulae (proof steps) of the discourse in such a way as to ensure that each step encountered satisfies the conditions required for it to be accepted as a consequence of what has gone before. This will be the case in various situations, each corresponding to one of the basic deduction rules which the system allows. Broadly speaking, these are as follows:

(i) Immediate deduction

The collection of statements already accepted as proved are always included in a 'penumbra' D of additional statements which follow from them as elementary consequences. The verifier as programmed is able to check that each statement in D follows immediately from statements already accepted. Some well-known examples are as follows.

(a) If a formula F in a proof is preceded by an (already accepted) formula G, and by a second (already accepted) formula of the form 'G *imp F', where '*imp' is the operator sign designating implication, then

F will be accepted;

(b) If a formula 'x in E' in a proof is preceded by an (already accepted) formula 'x in H', and by a second (already accepted) formula 'E incs H', 'where 'incs' is the operator sign designating set-theoretic inclusion, then 'x in E' will be accepted;

(c) If (c.1) we are given a formula having the syntactic structure 'P(e)', where 'P(x)' is a formula containing a variable x, and P(e) is the result of replacing each of the occurrences of x in P with an occurrence of the (syntactically well-formed) subexpression 'e'; (c.2) the formula P(e) is preceded by an (already accepted) formula (FORALL x | P(x)), where the symbol 'FORALL' designates the 'universal quantifier' construct of logic, then P(e) will be accepted.

The more we can enlarge the available family of immediate deductions by extending a verifier's immediate-deduction algorithms, the more we will succeed in reducing the number of steps needed to reach our target theorems. Means for doing this are explored later in this chapter, and then more systematically in Chapter 3.

(ii) Proof by 'supposition' and 'discharge' ('Natural Deduction')

At any point in a proof, any syntactically well-formed statement S can be introduced for provisional use by including a verifier directive of the form

Suppose ==> S.

Conclusions can be drawn from such statements in the normal way, but such conclusions are not accepted as having been definitively proved, but only as having been 'provisionally proved', subject to the 'assumption' expressed by S. However, if such an assumption S can be shown to lead to the impossible conclusion 'false', then S can be 'discharged', i.e. its negation 'not S' can be accepted as a definitely proved formula. This manner of proceeding mimics the familiar method of 'proof by contradiction' (also called 'reductio ad absurdum') of ordinary mathematical discourse.

(iii) Use of definitions

Statements which introduce entirely new constant or function names can be true 'by definition'. Suppose, for example, that constants b and c, and a monadic function symbol f, have already been introduced into a discourse, and that d is a name not previously used. Then the statement

d = f(b,f(c,b))

can be accepted immediately, since it merely *defines* d, i.e. makes an initial reference to an object d concerning which we know nothing else. Such definitions are subject to rules which serve to ensure that the new symbols introduced by such definitions imply only those properties of previously introduced symbols which are entailed by our previous knowledge concerning them. For example, a statement like

b = f(b,f(d,b))

is not a valid definition for a new constant d, since at the very least it implies that there exists some x for which b = f(b,f(x,b)) (and this may be false).

Definitions serve various purposes. At their simplest they are merely abbreviations which concentrate attention on interesting constructs by assigning them names which shorten their syntactic form. (But of

course the compounding of such abbreviations can change the appearance of a discourse completely, transforming what would otherwise be an exponentially lengthening welter of bewildering formulae into a sequence of sentences which carry helpful intuitions). Beyond this, definitions serve to 'instantiate', that is, to introduce the objects whose special properties are crucial to an intended argument. Like the selection of crucial lines, points, and circles from the infinity of geometric elements that might be considered in a Euclidean argument, definitions of this kind often carry a proof's most vital ideas.

As explained in more detail below, we use the dictions of set theory, in particular its general set formers, as an essential means of instantiating new objects. As we will show later by writing a hundred or so short statements which define all the essential foundations of standard mathematics, set theory gives us a very flexible and powerful tool for making definitions.

Our system allows four forms of definition. The first of these is definition using set formers (or 'algebraic constructions' more generally), as exemplified by

$$Un(s) := \{y: x \text{ in } s, y \text{ in } x\}$$

(which defines 'the set of all elements of elements of s', i.e. 'the union of all elements of s'), and assigns it the symbol 'Un' (which must never have been used previous to this definition). A second example is

$$Less\_usual(s) := \{y: x \text{ in } s, y \text{ in } x\} - s$$

(which defines 'the set of all elements of elements of s which are not directly elements of s').

The second form of definition allowed generalizes this kind of set-theoretic definition in a less commonly used but very powerful way. In ordinary definitions, the symbol being defined can only appear on the left-hand side of the definition, not on its right. This standard rule prohibits 'circular' definitions. In a *recursive definition* this rule is relaxed. Here the symbol being defined, which must designate a function of one or more variables, can also appear on the right of the definition, but only in a special way. More specifically, we allow function definitions like

$$f(s,t) := d(\{g(f(x,h_1(t)),s,t): x \text{ in } s \mid P(x,f(x,h_2(t)),s,t)\})$$

where it is assumed that d, g, h_1, h_2, and P are previously defined symbols and that f is the symbol being defined by the formula displayed. Here circularity is avoided by the fact that the value of f(s,t) can be calculated from values f(x,t') for which we can be sure that x is a member of s, so x must come before s in the implicit (possibly infinite) sequence of steps which build sets up from their members, starting with the empty set as the only necessary foundation object for the so-called 'pure' theory of sets.

'Transfinite recursive' definitions like that displayed above give us access to the sledgehammer technique called 'transfinite induction', which like other sledgehammers we use occasionally to break through key obstacles, but generally set aside.

The third and fourth forms of definition allowed, 'Skolemization' and use of 'theories', are explained later.

1.4 More about our formalism

Any formalism begins with some initial 'endowment', i.e. system of allowed formulae and built-in rules for the derivation of new formulae from old. If one intends to use such a formalism as a basis for metamathematical reasoning, one may aim to simplify the implied combinatorial analyses of the

formalism by minimizing this endowment. But we intend to use our formalism to track ordinary mathematical reasoning as closely and comfortably as we can; hence we streamline the endowment of formulae and formula transformations with which our system begins, but try to maximize its power. Accordingly, the system we propose incorporates various very powerful means for definition of objects and proof of their properties.

Propositional and predicate calculus

First consider what is most necessary, which we will handle in entirely standard ways. The apparatus of Boolean reasoning is needed if we are to make such statements as 'a and b are both true', 'a or b is true', 'a implies b', etc. The 'propositional calculus' required for this is elementary, and easily automated. We simply adopt this calculus, writing its operators as '&' (conjunction), 'or' (disjunction), 'not' (negation), '*imp' (implication), '*eq' (logical equivalence). Our system is *decidable*, in the sense it includes an algorithm able to detect statements which are universally true by virtue of their propositional form. This will, for example, automatically detect that

(p *imp q) *imp ((not q) *imp (not p))

and

(F(x + y) = F(F(x)) *imp F(F(x)) = 0) *imp (F(F(x)) /= 0) *imp (F(x + y) /= F(F(x))))

are both always true. The first of these formulae belongs directly to the 'propositional calculus'. Automatic treatment of the second formula uses a fundamental internal system operation called 'blobbing', which works by reducing formulae to skeletal forms legal in some tractable sublanguage of the full set-theoretic language in which we work. Applied to the second formula displayed above, 'blobbing' sees it to have a Boolean skeleton identical to that of the first. More is said about this important technique below.

Statements of the form 'for all..' and 'there exists ...', as in 'for all integers n greater than 2 there exists a unique non-decreasing sequence of prime integers whose product is n', are obviously needed for mathematics. To handle these, we adopt the standard apparatus of the 'predicate calculus' (or more properly 'first order predicate calculus'). This extends the propositional calculus by allowing its proposition-symbols p,q,... to be replaced by predicate subformulae constructed recursively out of

(i) *constants* c and *variables* x denoting specified or arbitrary objects drawn from some (implicit) universe U of objects.

(ii) Named *predicates*, e.g. P(x,y), Ord(x), Between(x,c,z), depending on some given number of constants and variables, which for each combination x,y,... yield some true/false (i.e. Boolean) value.

(iii) Named *function symbols*, e.g. f(x), g(x,y), h(x,c,z), depending on some given number of constants and variables, which for each combination x,y,... chosen from the 'universe' U yield an object belonging to this same universe.

(iv) Two '*quantifiers*',

(FORALL x | P(x)) and (EXISTS x | P(x)),

respectively representing the constructs 'for all possible values of the variable x , P(x) (the statement which follows the vertical bar) is true' and 'there exists some value of the variable x for which P(x) (the

statement which follows the vertical bar) is true'. For example, to express the condition that at least one of the predicates P(x) and Q(x) is true for each possible value of the variable x, we write

(FORALL x | P(x) or Q(x)).

To state that exactly one of these conditions is true for every possible value of the variable x, we can write

(FORALL x | (P(x) or Q(x)) & (not (P(x) & Q(x))))

To state that for each possible value of the variable x having the property P(x) there exists a value standing in the relationship R(x,y) to it, we can write

(1a) (FORALL x | P(x) *imp (EXISTS y | R(x,y))),

or equivalently

(1b) (FORALL x | (EXISTS y | P(x) *imp R(x,y))).

It should be plain that this predicate notation allows us to write universally and existentially quantified statements generally, provided only that names are available for all the multivariable predicates in which we are interested.

Intuitively speaking, a universally quantified (resp. existentially quantified) formula represents the conjunction (resp. disjunction) of all possible cases of the formula; e.g., (FORALL x | P(x)) can be regarded as a formalized abbreviation for the 'infinite conjunction' that might be written informally as

$P(x_1)$ & $P(x_2)$ & $P(x_3)$ & , ...

where x_1, x_2, x_3,... is an enumeration of all the values which the variable x can assume. Similarly, an existentially quantified statement like (EXISTS x | P(x)) can be regarded as a formalized abbreviation for the 'infinite disjunction' that might be written as

$P(x_1)$ or $P(x_2)$ or $P(x_3)$ or ... .

This shows us why the two predicate formulae (1a) and (1b) displayed above are equivalent, namely this informal style of interpretation explicates (FORALL x | P(x) *imp (EXISTS y | R(x,y))) as

($P(x_1)$ *imp (EXISTS y | $R(x_1,y)$)) & ($P(x_2)$ *imp (EXISTS y | $R(x_2,y)$)) & ...

and hence as

(1) ($P(x_1)$ *imp ($R(x_1,x_1)$ or $R(x_1,x_2)$ or $R(x_1,x_3)$ or ...)

& ($P(x_2)$ *imp ($R(x_2,x_1)$ or $R(x_2,x_2)$ or $R(x_2,x_3)$ or ...)

& ... .

Expansion of (FORALL x | EXISTS y | P(x) *imp R(x,y)) in exactly the same way results in

(2) (($P(x_1)$ *imp $R(x_1,x_1)$) or ($P(x_1)$ *imp $R(x_1,x_2)$) or ($P(x_1)$ *imp $R(x_1,x_3)$) or ...)

$$\& ((P(x_2) \text{ *imp } R(x_1,x_1)) \text{ or } (P(x_2) \text{ *imp } R(x_1,x_2)) \text{ or } (P(x_2) \text{ *imp } R(x_1,x_3)) \text{ or } ...)$$

$$\& \ ... \ .$$

Applying the standard propositional reduction of the implication operator 'p *imp q' to '(not p) or q' and using the commutativity of the disjunction operator 'or', we can rewrite the first line of (1) as

$$(\text{not } P(x_1)) \text{ or } (R(x_1,x_1) \text{ or } R(x_1,x_2) \text{ or } R(x_1,x_3) \text{ or } ...)$$

and the first line of (2) as

$$(\text{not } P(x_1) \text{ or } R(x_1,x_1)) \text{ or } (\text{not } P(x_1) \text{ or } R(x_1,x_2)) \text{ or } (\text{not } P(x_1) \text{ or } R(x_1,x_3)) \ ... \ ,$$

respectively, and similarly for all later lines. But since disjunction is idempotent, i.e.

$$p \text{ or } p \text{ or } p \text{ or } ...$$

is exactly equivalent to p, the two propositional expansions seen above are equivalent. Hence the claimed equivalence of

(FORALL x | P(x) *imp (EXISTS y | R(x,y))) and (FORALL x | EXISTS y | P(x) *imp R(x,y))

is intuitively apparent. We will explain later how the predicate calculus manages to handle all of this formally.

Set theory: the third main ingredient of our formalism

We view set theory as the established language of mathematics and take a rich version of it as fundamental. In particular, the language with which we will work includes a full sublanguage of set formers, constrained just enough to avoid paradoxical constructions like the {x: x notin x} setformer discussed above. Setformer expressions like

{e(x): x in s | P(x)},

{e(x,y): x in s(y) | P(x,y)},

{e(x,y,z): x in s(z), y in s'(x,z) | P(x,y,z)}

and even

{e(x,y,z,w): x in s(w), y in s'(x,w), z in s''(x,y,w) | P(x,y,z,w)}

are all allowed, as are

{e(x): x *incin s | P(x)},

{e(x,y): x *incin s(y) | P(x,y)},

{e(x,y,z): x *incin s(z), y *incin s'(x,z) | P(x,y,z)},

and

$$\{e(x,y,z,w): x \text{ *incin } s(w), y \text{ in } s'(x,w), z \text{ *incin } s''(x,y,w) \mid P(x,y,z,w)\},$$

which use the sign '*incin' designating set inclusion in place of one or more occurrences of the sign 'in' (designating set membership).

Set formers have several crucial advantages as language elements. First of all, they give us very powerful means for defining most mathematical objects of strategic interest. This allows the very succinct series of mathematical definitions given later, which lead in roughly 100 lines from rudimentary set theoretic concepts to core statements in analysis (e.g. the Cauchy integral theorem). A second advantage of set formers traces back to the fact that the human mind is 'perception dominated', in the sense that we all depend heavily upon many innate perceptual abilities, which operate rapidly and subconsciously, and by which the conscious (and reasoning) abilities of the mind are largely limited. Perceivable things and relationships can be dealt with rapidly. Where direct perception fails, we must fall back on more tortuous processes of reconstruction and detection, slowing progress by orders of magnitude. Hence the importance of notations, diagrams, graphs, animations, and scientific visualization techniques generally (e.g. the Arabic numerals, algebra, calculus, 'commutative diagrams' in topology, etc.). Among innate perceptual abilities we count the ability to decode spoken and written language, to remember phrases and simple relationships among them, and to recognize various language-like but somewhat more abstract syntactic structures. From this point of view, much of the importance of set theory and its set-former notations lies in the fact that their syntax reveals various simplifications and relationships with which the mind operates comfortably. These include:

(i) Various algebraic transformations of set formers, of which

$$\{e(x): x \text{ in } \{e'(y): y \text{ in } s \mid Q(y)\} \mid P(x)\} = \{e(e'(y)): y \text{ in } s \mid P(e'(y)) \,\&\, Q(y)\}$$

and

$$\{e(x): x \text{ in } \{e'(y,z): y \text{ in } s_1, z \text{ in } s_2 \mid Q(y,z)\} \mid P(x)\} =$$

$$\{e(e'(y,z)): y \text{ in } s_1, z \text{ in } s_2 \mid P(e'(y,z)) \,\&\, Q(y,z)\}$$

are typical.

(ii) Setformer expressions make various important monotonicity and domination relationships visible. For example, a glance at

$$\{e(x): x \text{ in } s \mid F(x) \text{ in } s - t\}$$

tells us that this expression is monotone increasing in s and monotone decreasing in t. From this, a statement like

```
(g(a) incs g(b) & h(a) *incin h(b)) *imp       ({e(x): x in s | F(x) in g(a) - h(a)} incs
```

is obvious by elementary reasoning concerning set unions, differences, and inclusions, which an algorithm can handle very adequately.

Deductions like this are frequent in the long sequence of steps which we will use to verify the standard mathematical material at which this text aims. Hence the stress we lay on deduction methods like that just explained, which we make available within our system under such names as ELEM ('elementary set-theoretic deduction', expanded as much as we dare), SIMPLF (deduction methods based on algebraic

simplification), etc. Hence also the special methods provided to deal with set-theoretic, predicate, and algebraic monotonicity.

The setformer constructs described above, and the other elementary operations of set theory, play two roles. On the one hand, they define operations on finite sets which can be implemented explicitly, for example by programming them systematically so as to create a full programming language which allows free use of finite sets as data objects. On the other hand, they define a language in which one can talk about a much larger universe of infinite sets, even though such sets can have no explicit representation other than the formulae used to speak of them. Since the formulae used to speak of infinite sets are the same as those used for finite sets, and since much the same axioms are assumed for sets of both kinds, many of the properties deduced for infinite sets stand in analogy to the more directly visible properties of finite sets.

A few simple but basic set constructs. The operation {x,y} which forms the (unordered) pair of two sets is an important but entirely elementary set operation. For this we have

```
z in {x,y} *eq (z = x or z = y).
```

Then plainly {x,x} satisfies z in {x,x} *eq z = x, so {x,x} is the singleton {x} whose only member is x.

The setformer expression

```
Un(x) := {z: y in x, z in y}
```

defines the set of all z which are elements of some element of x. This is the so-called 'general union set' of x, which can be thought of as 'the union of all elements of x'. Since we have

```
z in Un({x,y}) *eq (z in x or z in y),
```

Un({x,y}) is the set of all z which are either members of x or of y. This very commonly used operation is generally written as x + y. Given any two sets x and y, it gives us a way of constructing a set at least as large as either of them, of which both are subsets.

We can use the union operator to define the sets having three, four, etc. given elements by writing

```
{x,y,z} = {x,y} + {z}, {x,y,z,w} = {x,y,z} + {w},...
```

It is easily proved from these definitions that

```
u in {x,y,z} *eq (u = x or u = y or u = z),    u in {x,y,z,w} *eq (u = x or u = y or u = z or u
```

etc. The intersection operator, which gives the common part of two sets s and t, can be defined directly by a setformer:

```
s * t := {x: x in s | x in t}.
```

The powerset operator, which gives the set of all subsets of a set s, can also be defined by a setformer expression:

```
pow(s) := {x: x *incin s}.
```

The choice operator 'arb'. The less elementary 'choice' operation arb(s) reflects the intuition, verifiably true in the hereditarily finite case discussed in Chapter 2, that all sets can be constructed in an order in which all the elements of set s are constructed before s itself is constructed. Since, as we shall see, a finite string representation is available for each hereditarily finite set, we can arrange such sets in order of the length of their string representations. Then arb(s) can be defined for each finite set as the first member of s, in this standard order. We complete this definition for the one special case in which s has no members, i.e. is the null set, by agreeing that arb({}) = {}. Then, for each nonempty set s, arb(s) must be disjoint from s, since if x were a common member of s and arb(s), x would have to be an element of s coming earlier than arb(s) in standard order, contradicting our definition of arb(s) as the first element of s in this order. Hence, whenever this notion of 'construction in some standard order' applies, we can expect the 'arb' operator, defined in the manner just explained, to satisfy

```
(FORALL s | (s = {} & arb(s) = {}) or (arb(s) in s & arb(s) * s = {})).
```

This statement, intuitively justified in the manner just explained, is taken as an axiom in the version of set theory used in this book. It is assumed to apply to all sets, whether finite or infinite. In conventional terms, this axiom states a very strong form of the so-called 'axiom of choice': arb chooses a first element from each nonempty set, 'first' in the sense that there exists no other element of s which is also an element of arb(s).

It follows that there can exist no set x for which

```
x in x.
```

For if there were, we would have arb({x}) = x, and so x would be a common element of {x} and arb({x}), contradicting our assumption concerning 'arb'. It follows similarly that there can exist no 'membership cycle', i.e. no sequence $x_1,x_2,...,x_n$ of sets of which each is a member of the next and for which the last is a member of the first. For if there were, we would have arb($\{x_1,x_2,...,x_n\}$) = x_j for some j, and then either x_{j-1} or x_n would be a common element of arb($\{x_1,x_2,...,x_n\}$) and $\{x_1,x_2,...,x_n\}$. Much the same argument shows that there can exist no infinite sequence $x_1,x_2,...,x_n,...$ for which each x_{j+1} is a member of x_j. Note however that x, {x},{{x}},... is always a sequence each of whose components is an element of the next following component.

The 'arb' operator as the basis for proofs by transfinite induction. The standard (Peano) principle of mathematical induction is equivalent to the statement that every non-empty set s of integers contains a smallest element n_0. For suppose that P(n) is a predicate, defined for integers, for which the implication

```
(FORALL n | (FORALL m | (m < n) *imp P(m)) *imp P(n))
```

has been established, that is, for which P(n) must be true for a given n if it is true for all smaller m. Then P(n) must be true for all integers n. For if not, the set of all integers n such that P(n) is false will be nonempty, and so will contain a smallest integer n_0. But then P(m) is clearly true for all $m < n_0$, implying that $P(n_0)$ is true, contrary to assumption.

Use of the 'arb' operator allows us to extend this very convenient style of inductive reasoning to entirely general sets, irrespective of whether they are finite or infinite. Suppose, more specifically, that P(s) is a predicate, defined for sets, for which the implication

```
(FORALL s | (FORALL t | (t in s) *imp P(t)) *imp P(s))
```

has been established. That is, we suppose that P(s) must be true for a given s if it is true for all members

of s. Then P(s) must be true for all sets s. For if not, then P(s) must be false for some member s_1 of s. Repeating this argument, we see that there must exist a member s_2 of s_1 for which $P(s_2)$ is false, then a member s_3 of s_2 for which $P(s_3)$ is false, and so forth. This gives us an infinite sequence $s = s_0,s_1,s_2,...,s_n,...$, each component of which is a member of the preceding component, which we have seen to be impossible.

This very broad generalization of the ordinary principle of mathematical induction is called the *principle of transfinite induction*. It plays much the same role for the infinite ordinals discussed in the next section that the ordinary principle of mathematical induction plays for integers.

Ordered pairs We need, in many situations, not the unordered pair construct {x,y} described above, but rather an ordered pair construct [x,y]. The only properties of [x,y] that we require are: (i) [x,y] is defined for any two sets x, y and is itself a set; (ii) the pair [x,y] defines its two components x and y uniquely, i.e. there exist operations car(z) and cdr(z) such that car([x,y]) = x and cdr([x,y]) = y for all x and y. It is not necessary to add these statements as additional set-theoretic axioms, since the necessary pairing operations can be defined using the unordered pair construct {x,y} and the arb operator, in any number of (artificial) ways (none of them having any particular significance). For example, we can use the definition

```
[x,y] := {{x},{{x},{{y},y}}}.
```

Then arb([x,y]) = {x}, since the only other element of {{x},{{x},{{y},y}}} has the element {x} in common with [x,y]. Thus the expression arb(arb([x,y])) always reconstructs x from [x,y]. Moreover {{x},{{x},{{y},y}}} - {{x}} = {{{x},{{y},y}}}, so

```
arb(arb([x,y] - {arb([x,y])}) - {arb(x)} = {{y},y},
```

and therefore the expression arb(arb(arb([x,y] - {arb([x,y])}) - {arb(x)}) reconstructs y from [x,y]. The reader is invited to amuse him/herself by inventing other like constructions having similar properties.

Once ordered pairs and the operators which extract their components have been defined in this way, it is easy to define the general set-theoretic notion of 'relationship' and the associated notions of 'single-valued mapping', 'inverse relationship', and '1-1 relationship'. A *relationship* or *mapping*, or just *map*, is simply a set of ordered pairs. To formalize this, we have only to write

```
is_map(f) := (f = {[car(x),cdr(x)]: x in f}).
```

The domain and range of a relationship are then defined in the usual way as

```
domain(f) := {car(x): x in f}
```

and

```
range(f) := {cdr(x): x in f}
```

respectively. A relationship is single-valued if the first component u of each pair [u,v] in it defines the associated second component v uniquely. Formally this is

```
Svm(f) := is_map(f) &        (FORALL x in f | (FORALL y in f | (car(x) = car(y)) *imp (x = y)))
```

The inverse of a relationship is defined by

```
inv(f) := {[cdr(x),car(x)]: x in f}.
```

A relationship is 1-1 if it and its inverse are both single-valued. Other standard constructs involving mappings, for example the composition of two mappings, are equally easy to define.

Integers and ordinal numbers in set theory. As noted above, John von Neumann suggested that the fundamental mathematical notion of 'integer' be expressed set-theoretically by encoding 0,1,..,.n,... set theoretically as {},{0},...,{0,1,..,n - 1},... The set Z of all integers is then

{0,1,..,n,...}.

All of these sets s, including the infinite set Z, have the following properties:

(i) any member of a member of s is also a member of s;

(ii) given any two distinct members x,y of s, one of x and y must (come earlier in the sequence in which we have enumerated the members of s, and so must) be a member of the other;

von Neumann then realized that sets having these two properties had exactly the properties of 'ordinal numbers' as originally defined by Cantor, so that (i) and (ii) can be taken as the definition of the notion of ordinal number. Besides its striking directness and simplicity, this definition has the advantage (over Cantor's original definition) of representing each ordinal number by a unique set. Moreover, all the basic operations on infinite ordinals which Cantor introduced take on simple set-theoretic forms if ordinals are defined in this way. For example, for the integers in their von Neumann representation, each integer m less than an integer n is a member of n; hence the arithmetic relationship 'm < n' can be defined 'm in n', i.e. by the simplest of all set theoretical relationships. We use this definition, i.e. 's less than t' means simply 's in t', for arbitrary ordinals s.

Instantiation and proof by use of 'Theories'

The 'theory' mechanism which our system provides relates to logical proof in something like the way in which the use of 'procedures' relates to programming practice. It facilitates introduction of symbol groups or single symbols (like the standard mathematical summation operator Σ and the rather similar product operator Π) which derive from previously defined functions and constants ('+' and '0' in the case of Σ, multiplication and '1' in the case of Π), that have the properties required for definition of the new symbols. As these examples indicate, our 'theory' mechanism eases an important class of instantiations which need to be justified by supporting theorems. It adds a touch of second-order logic capability to the first-order system in which we work.

The syntax used to work with 'theories' is described by the following procedure-like template.

THEORY theory_name(list_of_assumed_symbols) assumptions ==>(list_of_defined_symbols) conclusions END theory_name;

The formal description of the important 'theory of sigma', which we will use as a running example, illustrates the way in which we set up and use theories. This theory captures a construction, ubiquitous in mathematical practice, which is normally written using 'three dots' notation, e.g. as $f_1 + f_2 + ... + f_k$.

```
THEORY SIGMA_theory(s,PLUZ,e)    e in s    (FORALL x in s | (FORALL y in s | x PLUZ y in s))    (F
```

(The final conclusion displayed encapsulates a general 'rearrangement of sums' principle). The assumed_symbols of this theory are s, PLUZ, and e, and its only defined symbol is SIGMA. 'Finite' and 'Svm' are standard set-theoretic predicates, which we assume to have been defined prior to the

introduction of the theory displayed: 'Finite(f)' states that f is finite, and 'Svm(f)' states that f is a single-valued map. Similarly, 'domain(f)' and 'range(f)' denote the domain and range of f respectively, 'fld' denotes the restriction of f to d (namely the largest possible map which is included in f and whose domain is included in d), and 'INV_IM{g,y}' denotes the set of all elements of the domain of g which g maps into the element y. f{x} designates the range of f on the set {x}, and arb(f{x}) the unique element of this range, i.e. the image of x under the single-valued mapping f.

Were the mechanisms of second-order predicate calculus available to us, the meaning of the theory could be rendered precisely by

```
(FORALL s | (FORALL PLUZ | (FORALL e | (EXISTS SIGMA |       (e in s &        (FORALL x in s | (
```

Informally speaking, this second-order formula states that given any set s and commutative-associative operator defined on it, there must exist a monadic function SIGMA which relates to them in the manner stated in the conclusion of the quantified formula displayed. If our formalism allowed the second-order mechanisms (of quantification over function and relation symbols, which it does not) seen here, and were this second-order formula proved, we could substitute any three actual symbols for which the hypotheses of the formula had been proved for the three universally quantified function symbols s, PLUZ, and e which appear, thereby obtaining the existentially quantified conclusion

```
(EXISTS SIGMA |       (e in s &        (FORALL x in s | (FORALL y in s | x PLUZ y in s)) &
```

This last statement (still second-order, since it is quantified over the function symbol SIGMA) would allow us to introduce a new symbol SIGMA_ for which

```
(e in s &        (FORALL x in s | (FORALL y in s | x PLUZ y in s)) &         (FORALL x in s | x
```

is known. This final statement is now first-order.

The second-order mechanisms needed to proceed in just the manner explained are not available in our first-order setting. The theory mechanism that is provided serves as a partial but adequate substitute for it.

After these introductory remarks we return to a detailed consideration of the general theory template displayed at the start of this section. In it, 'theory_name' names the theory in which we are interested. A theory's 'list_of_assumed_symbols' is analogous to the parameter list of a procedure. It is a comma-separated list of symbol names, which stand for other symbols which must replace the assumed_symbols whenever the theory is *applied*. The members of the list of 'assumptions' which follow must be formulae which, aside from basic predicate and set-theoretic constructions (quantifiers and set formers), involve only elements of the list_of_assumed_symbols, possibly along with other symbols that have been defined previously to introduction of the theory, in the context in which the theory is introduced. The formal description of the 'theory of SIGMA' given above illustrates these rules.

The 'conclusions' which follow the syntactic delimiter '==>' in the general template must be formulae which, aside from basic predicate and set-theoretic constructions, involve only elements of the list_of_assumed_symbols and the list_of_defined_symbols, along with other symbols that have previously been defined in the context in which the theory is introduced. The elements of the (comma-delimited) list_of_defined_symbols are symbol names, which must be defined within the theory, more precisely as part of a proof (given within the theory), of the theory's stated conclusions. Each defined_symbol is replaced with a previously unused symbol whenever the theory is *applied*.

Once a theory has been introduced in the manner just explained, and before it can be used, a sequence of theorems and definitions culminating in those which appear as the conclusions of the theory must be proved in the theory. The syntax used to begin this process, which temporarily 'enters' the theory, is simply

```
ENTER_THEORY theory_name
```

This statement creates a subordinate proof context in which the assumed_symbols of the theory, together with all its stated assumptions, are available. Then, using these assumptions, one must give definitions of all the theory's defined_symbols, and proofs of all its conclusions. Once this has been done, one can return from the subordinate logical context to the parent context from which it was entered by executing another ENTER_THEORY command, which now must name the parent theory to which we are returning. (Proof always begins in a top-level context named 'set_theory'). After return, the theory's conclusions become available for application. Note also that theories previously developed in the parent context of a new theory T are available for application during the construction of T.

The syntax (analogous to that for 'calling' procedures) used to apply theories is

```
APPLY(new_symbol:defined_symbol_of_theory,...)          theory_name(list_of_replacements_for_as
```

As indicated, the keyword 'APPLY' is followed by a comma-delimited sequence of colon-separated pairs which associates each defined_symbol of the theory with a previously unused symbol, which then replaces the defined_symbol in the set of conclusions that results from successful application of the theory. Next there must follow a comma-delimited list of symbols defined previously, equal in length to the theory's list of assumed_symbols, which specifies the symbols which are to replace the assumed_symbols at the point of application. Our verifier replaces all the assumed_symbols appearing in the theory's assumptions with these replacement symbols, and searches the logical context available at the point of theory application for theorems identical with the resulting formulae. If any of these is missing, the requested theory application is refused. If all are found, then the conclusions of the theory are turned into theorems by replacing every occurrence of the theory's defined symbols by the corresponding new_symbol and every occurrence of the theory's assumed symbols by its specified replacement symbol.

Assume, for example, that the 'SIGMA_theory' displayed above has been made available (in the way explained above), and that theorems

e in Z , (FORALL x in Z | x + 0 = x) , (FORALL x in Z | (FORALL y in Z | x + y = y + x)) , (FORALL x in Z | (FORALL y in Z | (FORALL z in Z | (x + y) + z = x + (y + z))))

have been proved (separately from the theory) for the integers Z, and integer addition. Then the verifier instruction

APPLY(SIG:SIGMA) SIGMA_theory(Z,+,0)

makes the symbol SIG (which must not have been defined previously) available, and gives us the theorem

```
(FORALL f | (Finite(f) & Svm(f) & range(f) *incin s) *imp          (SIG(f) in Z & SIG({}) = 0 &
```

without further proof.

The theory of equivalence classes is a second important 'theory' example.

THEORY equivalence_classes(P,s) (FORALL x in s | (FORALL y in s | (P(x,y) *eq P(y,x)) & P(x,x))) (FORALL x in s | (FORALL y in s | (FORALL z in s | (P(x,y) & P(y,z)) *imp P(x,z)))) ==>(Eqc,F) (FORALL x in s | F(x) in Eqc) & (FORALL y in Eqc | (arb(y) in s & F(arb(y)) = y)) (FORALL x in s | (FORALL y in s | P(x,y) *eq (F(x) = F(y)))) (FORALL x in s | P(x,arb(F(x)))) (FORALL x in s | x in F(x)) END equivalence_classes;

This states that any dyadic 'equivalence relation' P(x,y) can be represented in the form P(x,y) *eq (F(x) = F(y)) by some monadic function F. (Conventionally, one speaks of F(x) as the equivalence class of x; notice, however, that we are deliberately 'hiding' such secondary facts as '{ } notin Eqc', 's=Un(Eqc)', and '(FORALL y in Eqc, x in y | F(x) = y)'). The theory of equivalence classes is one of a family of easy but widely applicable results which represent various kinds of monadic relationships in terms of elementary relationships which are especially easy to work with (often because decision algorithms apply to them). For example, one can easily show that any partial ordering on set elements x,y can be represented in the form F(x) *incin F(y). Results of this kind lend particular importance to the relationships to which they apply.

1.5 An informal overview of the sequence of formal set-theoretic proofs to be given later

This text culminates in the sequence of definitions and proofs found in Chapters XX-YY. The theorems (with proofs set up to be verifiable by our system) fall into the following categories:

Basic elementary results

(i) Definition and basic properties of ordered pairs. These are fundamental to many of the following definitions, e.g. of maps and of the Cartesian product.

(ii) Definition of the notions of map, single-valued map, 1-1-map, map restriction, domain, range, map product, etc. and derivation of the ubiquitous elementary properties of maps, as a long series of elementary theorems. Some of these properties of maps are captured for convenience in a theory called 'fcn_symbol' which can be used to prove basic properties of set formers defining single-valued maps.

Ordinals

(iii) Definition of the notion of 'ordinal', and proof of the basic properties of ordinals. Completely formal proofs of all the basic properties of ordinal numbers will be given in Chapter 5. But to make these proofs more comprehensible it is well to translate some of them, and some of the key definitions used in them, into the more comfortable language of ordinary mathematics. We follow von Neumann in defining an ordinal as a set (I) properly ordered by membership, and for which (II) members of members are also members. The key results proved are: (a) the collection of all ordinals is itself properly ordered by membership, and members of ordinals are ordinals, but (b) the collection of all ordinals is not a set. Also, (c) proceeding recursively in the manner explained in Section XX, we define a standard enumeration for every set and show that this puts the members of the set in 1-1 correspondence with an ordinal. This is the 'enumerability principle' fundamental to our subsequent work with cardinal numbers.

The von Neumann representation ties the ordinal concept very directly to the most basic concepts of set theory, allowing the properties of ordinals to be established by reasoning that uses only elementary properties of sets and set formers, with occasional use of transfinite induction. (For ease of use, statement and proof of this general principle are captured as a theory called 'transfinite_induction': the principle follows very directly from our strong form of the axiom of choice).

For example, in the von Neumann representation, the next ordinal after an ordinal s is simply s + {s}. To see that s' = s + {s} must be an ordinal, note first that each member of a member of s' is either a member of a member of s, or directly a member of s; and hence in any case a member of s'; thus s' has property (II). The proof that s' also has property (I) is equally elementary and is left to the reader. Together these show that s' is an ordinal. Other equally elementary results concerning ordinals, whose proof is also left to the reader are:

a. The intersection s * t of any two ordinals is an ordinal.

b. Any member t of an ordinal s is an ordinal.

Let s be an ordinal. Since any member of a member t of s is a member of s by (i), any member t of s is a subset of s. Thus for ordinals the membership relation 't in s' implies the inclusion relation 't *incin s'. On the other hand, if t is also an ordinal and t *incin s, then either t = s or t in s. To prove this, suppose that t /= s, and consider the element x = arb(s - t). Any element y of t is also an element of s, so by (ii) we have either y in x, y = x, or x in y. Both y = x and x in y would imply x in t which is impossible. Thus we must have y in x whenever y in t, i.e. t *incin x. But x - t must be null. Indeed, let z in x - t. Then z in x, but also z in s - t, contradicting the fact that x = arb(s - t) is disjoint from s - t. Hence x = t, i.e. t is an element of s, proving our assertion that any subset t of s which is also an ordinal must either be identical to s or must be a member of s. That is, for ordinals the relationship '*incin s' is equivalent to the condition 'is a member of s or is equal to s.'

Next we show that, given any two distinct ordinals s and t, one is a member of the other. Suppose that this is not the case. Then if s = s * t then s is a subset of t, and hence, by the result just proved, is a member of t. Similarly, if t = s * t then t is a member of s. So it follows that s /= s * t and t /= s * t. Since s * t is an ordinal and a subset of s, it follows by the result just proved that s * t in s; similarly s* t in t, so s * t in s * t, which is impossible since the membership operator can admit no cycles. This proves our claim.

It follows that if s and t are both ordinals, the intersection s * t is the smaller of s and t, while the union s + t is the larger of s and t. If O is any non-empty set of ordinals, then x = arb(O) is a member of O and hence an ordinal. By definition of arb, x must be disjoint from O. Hence if y is any other member of O, y in x is impossible so x in y must be true. That is, arb(O) must be the smallest of all the elements of O. Moreover the union Un(O) of all the elements of O must be an ordinal, since if x in Un(O) and y in x then there is an s in O such that x in s, from which it follows that y in s and so y in Un(O), proving that Un(O) has property (i). Moreover if x in Un(O) and y in Un(O), then there must exist s in O and t in O such that x in s and y in t. Then one of s and t, say s, must include the other, and so x and y must both be members of s. Since s is an ordinal and therefore has property (ii), it follows that either x in y, x = y, of y in x. Hence Un(O) also has property (ii). This shows that the union Un(O) of any set of ordinals must itself be an ordinal, which is easily seen to be the smallest ordinal including all the members of O.

Using the statements just proved it is easy to show that if s is an ordinal, then s' = s + {s} is the least ordinal greater than s. Indeed, we have shown above that s' is an ordinal. Moreover s in s', so s' is larger than s in the ordering of ordinals. If t is any ordinal larger than s, i.e. s in t, then either s' in t, s' = t, or t in s' by what has been proved above. But t in s' is impossible, since it would imply that either t in s or t = s, and so in either case would lead to an impossible membership cycle. Therefore either s' in t or s' = t, i.e. t is no smaller than s', proving that s' is the least ordinal greater than s, as asserted. It is therefore reasonable to write s + {s} as next(s).

Any ordinal s which is greater than every integer n must have all such n as members, proving that the set Z of all integers must be a subset of the set s. Hence Z must be the smallest ordinal which is greater than every integer n. Therefore the smallest members of the collection of all ordinals can be written as

0,1,..,n,...,Z,next(Z),next(next(Z)),...

in their natural order (of membership). In his initial series of papers on ordinals Georg Cantor introduced a variety of constructions for ordinals which generalize various arithmetic constructions for ordinary integers and which allow the sequence of ordinal notations shown above to be extended systematically.

Well ordering: the principle of transfinite enumerability

The ordinal numbers, as we (or von Neumann, or Cantor) have defined them capture an abstract notion of sequential enumeration, even for sets which are not restricted to be finite. A crucial property of the ordinals is that they allow any set s to be enumerated, irrespective of whether s is finite or infinite. This is the so-called Well-Ordering Theorem. This famous result is not hard to prove given the very generous variant of set theory which we allow, which as explained earlier lets us write very general recursive definitions in set theoretic notation, and also admits free use of the choice operator 'arb'.

To prove the well-ordering theorem, we first show that the collection Ord of all ordinals is not a set, i.e. that there is no set O such that s is an ordinal if and only if s in O. For otherwise s = Un(O) would be an ordinal by what we have just proved, and so as shown above s + {s} is also an ordinal, implying that s is a member of a member of s, and so s in s, which is impossible.

Next define a function enum(X,S) of two parameters by writing

enum(X,S) := if S *incin {enum(y,S): y in X} then S else arb(S - {enum(y,S): y in X}) end if.

That is, we define enum(X,S) to be the element of S - {enum(y,S): y in X} chosen by 'arb' if {enum(y,S): y in X} differs from S; otherwise enum(X,S) is simply S. This definition implies that the elements enum(0,S),enum(1,S),enum(2,S),...,enum(Z,S),... have the following values:

```
enum(0,S) = arb(S)    enum(1,S) = arb(S - {arb(S)})    enum(2,S) = arb(S - {arb(S),enum(1,S)})    .
```

The crucial fact, proved in the next paragraph, is that the elements enum(x,S) remain distinct, for distinct ordinals x, as long as {enum(y,S): y in x} is a proper subset of S. Note also that as the ordinal x increases, so does the set {enum(y,S): y in x}.

It is easy to prove that enum(x,S) and enum(y,S) must be distinct if x and y are distinct ordinals and both enum(x,S) and enum(y,S) are different from S. Indeed, one of x and y, say y, must be a member of the other, and then by definition we must have enum(x,S) = arb(S - {enum(z,S): z in x}), so enum(x,S) in S - {enum(z,S): z in x}, while enum(y,S) in {enum(z,S): z in x}. It follows from this that there must exist an ordinal x for which S = {enum(z,S): z in x}. For if this is false, then by what we have just proved the mapping z :-> enum(z,S) maps the collection of all ordinals in 1-1 fashion into a subset of the set S. But an axiom of set theory (the so-called 'Axiom of Replacement', detailed below) tells us that every collection which can be put in 1-1 correspondence with a set must itself be a set. Hence it would follow that the collection of all ordinals is a set, contradicting what has been proved above.

Since we have just shown that there exists an ordinal x such that S = {enum(z,S): z in x}, there must exist a least such ordinal y (which we can define as

```
arb({y in next(x) | S = {enum(z,S): z in y}}).
```

It is easily seen (we leave details to the reader) that z :-> enum(z,S) maps this y in 1-1 fashion onto S, completing our proof of the Well-Ordering Theorem.

Cardinal numbers

(iv) Definition of 'cardinality' and of the operator #s which gives the (possibly infinite) number of members of a set s. The cardinality of a set is defined as the smallest ordinal which can be put into 1-1 correspondence with the set, and it is proved that (a) there is only one such ordinal, and (b) this is also the smallest ordinal which can be mapped onto s by a single-valued map.

The proof of the Well-Ordering Theorem puts us in position to introduce the notion of *cardinal number* and to prove the basic elementary properties of these numbers. We define the cardinals as a subset of the ordinals; an ordinal x is called a cardinal if x cannot be put into 1-1 correspondence with any smaller ordinal. By the Well-Ordering Theorem, any set s can be put in 1-1 correspondence with some ordinal, and arguing as above it follows that s can be put in 1-1 correspondence with some smallest ordinal x. Since the composition of two 1-1 mappings is itself 1-1, it follows that this unique x must itself be a cardinal. We call this cardinal the *cardinality of* s, and write it (using the standard number sign) as #s.

In this section we also define the notions of cardinal sum and product of two sets a and b. These are respectively defined as #(copy_a + copy_b), where copy_a and copy_b are disjoint copies of a and b, and the cardinality of the Cartesian product a *PROD b of a and b. Using these definitions, it is easy to prove the associative and distributive laws of cardinal arithmetic. We also prove a few basic properties of the #s operator, e.g. its monotonicity.

(v) A set s is then defined to be *finite* if it has no 1-1 mapping into a proper subset of itself, or, equivalently, is not the single-valued image of any such proper subset. We prove that the null set and any singleton are finite, and (using transfinite induction) that the collection of finite sets is closed under the union, Cartesian product, and power set operators. It is proved that s is finite if and only if its cardinality #s is finite. We then prepare for the introduction of signed integer arithmetic by proving all the basic arithmetic properties of unsigned integers and then defining the cardinal subtraction operator a MINUS b and showing that for finite cardinals subtraction has its expected properties. We also prove that integer division with remainder is always possible. These results are proved with the help of a modified version of the principle of induction which is demonstrated for finite sets: given any predicate P(x) not true for all finite sets, there exists a finite set s for which P(s) is false, but P(s') is true for all proper subsets of s. Like the rather similar transfinite induction, this principle is captured for convenience in a theory.

(vi) Sets which are not finite are said to be *infinite.* By considering the cardinality #s_inf of the infinite set s_inf whose existence is assumed in an axiom of infinity, we prove that there exists an infinite cardinal, and so can define the set Z of integers as the least infinite ordinal, and show that this is a cardinal, and is in fact the set of all finite cardinals. The set Z of all integers is infinite, since the 1-1 correspondence n :-> next(n) maps Z to a subset of itself (the zero integer, i.e. {}, is not in the range of 'next'). It is not hard to see that if the set s is finite, so is next(s) = s + {s}. Indeed, if s + {s} is infinite, there exists a 1-1 mapping f of s + {s} to a proper subset of itself. The range of the mapping f must therefore omit some element of s + {s}, i.e. must either omit s or some element x of s. Consider the latter of these two cases. We can plainly construct a 1-1 mapping g of s + {s} onto itself which interchanges x and s. Then the composition of f and g is a 1-1 mapping of s + {s} into itself whose range omits the value s. This shows that if next(s) is infinite, there must always exist a 1-1 mapping f of next(s) into s, but then f maps s into s - {f(s)}, so s is also infinite. I.e., s is infinite if next(s) is infinite, implying that next(s) is finite if s is finite.

It follows that all the integers 0 = {}, 1 = next(0), 2 = next(1),... are finite, and so each of these ordinals must also be a cardinal. Moreover, the infinite ordinal Z must also be a cardinal. Indeed, if this is not the case, there would exist a 1-1 mapping f of Z into a smaller ordinal, i.e. to some integer n in Z. But then f would also map the subset next(n) of Z into its proper subset n, implying that next(n) is infinite, which we have seen to be impossible. Thus Z is not only the smallest infinite ordinal but also the smallest infinite

cardinal. This implies that

```
#0 = 0, #1 = 1, #2 = 2,... ,#Z = Z
```

(every cardinal is its own cardinality, and every ordinal less than or equal to Z is a cardinal). On the other hand, the cardinality of next(Z) = Z + {Z} is simply Z. Indeed, we have seen that there exists a 1-1 mapping f of Z into itself whose range omits the integer 0; this can plainly be extended to a 1-1 mapping of Z + {Z} into Z. This same argument shows that if #s = Z then #next(s) = Z also. Therefore the sequence of cardinalities of the ordinals

```
0,1,2,...,Z,next(Z),next(next(Z)),next(next(next(Z))),...
```

is

```
0,1,2,...,Z,Z,Z,Z,...
```

That is, all the infinite ordinals displayed, though distinct, have the same cardinality. Any set s whose cardinality #s is Z is said to be *denumerable*, or *countably infinite*; and a set which is either finite or denumerable is said to be *countable*. Our next question is: how can we be sure that *uncountable* sets, namely sets whose cardinality exceeds Z, actually exist?

(vii) Another idea is plainly needed if we are to show that there exist any cardinals larger than Z. As a digression, we prove that the sum and product of any two infinite cardinals degenerates to their maximum (hence there are no more rational numbers than there are integer numbers), but (Cantor's Theorem) that the power set of any cardinal always has a larger cardinality. Cantor noted that for any set s, the set pow(s) of all subsets of s must have cardinality larger than that of s. For suppose the contrary, i.e. suppose that there exists a 1-1 mapping f of s onto pow(s). Then consider the subset {x: x in s | x notin f(x)} of s. This must have the form f(y) for some y in s; hence f(y) = {x: x in s | x notin f(x)}. But then y in f(y) is equivalent to y in {x: x in s | x notin f(x)}, i.e. to y notin f(y), which is impossible. (Incidentally, since a 1-1 correspondence between reals and pow(Z) can be found, this implies that real numbers form an uncountable set).

Since s always has a 1-1 embedding into pow(s) (we can simply map each x in s into the singleton {x}), the cardinality of s is never greater than that of pow(s). The theorem of Cantor proved in the preceding paragraph shows that in fact we always have #s < #pow(s), i.e. #s in #pow(s). Hence #pow(Z) is an infinite cardinal which is definitely larger than Z; similarly #pow(#pow(Z)) is larger than #pow(Z) and so forth, proving that there must exist infinitely many infinite cardinals. In fact, we can easily prove that there exists a 1-1 correspondence between the collection of all ordinals and the collection of all cardinals. For this, we simply need to make the transfinite inductive definition

```
alph(x) := arb({z: z in     (next(#pow(Un({alph(y): y in x})))) - {alph(y): y in x}) | is_cardinal(z
```

where 'is_cardinal' is the predicate, easily expressible in elementary set-theoretic terms, which states that its argument y is a cardinal number. Since all the occurrences of 'alph' on the right-hand side of this definition lie in the scope of constraints of the form 'y in x', this is a legal transfinite definition according to the rule stated earlier. For each ordinal x, this formula defines alph(x) to be the smallest cardinal (if any) which is not more than #pow(Un({alph(y): y in x}) but is not one of the cardinals alph(y) for any ordinal y less than x. Since we have seen above that u = Un({alph(y): y in x} is an ordinal at least as large as any of the alph(y) for y in x, and also that #pow(u) is larger than u, the set next(#pow(u)) - {alph(y): y in x}) must be nonempty, and so alph(x) must indeed be the smallest cardinal greater than all of the cardinals alph(y) for any ordinal y in x. It is easily seen (details are left to the reader) that alph(y) <

alph(z) if y < z. Hence the function 'alph' is a 1-1, monotone increasing map of the collection of all ordinals to the collection of all cardinals. It is not hard to prove that every cardinal must appear as on of the alph(y). Thus 'alph' actually puts the collection of all ordinals in 1-1 correspondence with the collection of all cardinals. For small ordinals we have

```
alph(0) = 0, alph(1) = 1, alph(2) = 2,..., alph(Z) = Z.
```

A mystery, first encountered by Cantor, occurs at the very next position in this sequence. alph(next(Z)) is the smallest cardinal greater than Z. We have seen that the cardinal number #pow(Z) is larger than Z; hence alph(next(Z)) <= #pow(Z). But is this inequality actually an equality, or does there exist a cardinal number between Z and #pow(Z)? Indeed, do there exist infinitely many cardinal numbers in this range? This is the so-called 'Continuum problem', originally stated by Cantor. Its very surprising resolution, ultimately achieved by Kurt Gödel and Paul Cohen, required over 60 years of penetrating work: the statement alph(next(Z)) = #pow(Z) is independent of the axioms of set theory, which admit both of models in which this statement is true and of many structurally distinct models in which it is false.

All the semi-formal proofs given above will recur, in completely formalized versions, in Chapter 5. The semi-formal proofs given in this section can serve as intuitive guides to the larger mass of detail appearing in these formal proofs.

Survey of the major sequence of definitions and proofs considered in this text

(viii) The set of signed integers is then introduced as the set of pairs [x,0] (representing the positive integers) and [0,x] (representing the integers of negative sign). [0,0] is the 'signed integer' 0, and the 1-1 mapping x :-> [x,0], whose inverse is simply y :-> car(y), embeds Z into the set of signed integers, in a manner allowing easy extension of the addition, subtraction, multiplication, and division operators to signed integers. In preparation for introduction of the set of rational numbers, it is proved that the set of signed integers is an 'integral domain'. At this point, we are well on the royal road of standard mathematics.

(ix) Next we introduce two important 'theories' mentioned above: the theory of equivalence classes and the theory of SIGMA. As previously noted, the theory of SIGMA is a formal substitute for the common but informal mathematical use of 'three dot' summation (and product) notations like

$$a_1 + a_2 + ... + a_n \text{ and } a_1 * a_2 * ... * a_n.$$

The theory of equivalence classes characterizes the dyadic predicates R(x,y) which can be represented in terms of the equality predicate using a monadic function, i.e. as R(x,y) *eq (F(x) = F(y)). These R are the so-called 'equivalence relationships', and for each such R defined for all x belonging to a set s, the theory of equivalence classes constructs F (for which arb turns out to be an inverse), and the set into which F maps s. This range is the 'family of equivalence classes' defined by the dyadic predicate R. The construction seen here, which traces back to Gauss, is ubiquitous in 20th century mathematics.

To illustrate the use of the theory of SIGMA we digress slightly from our main line of development to prove the prime factorization theorem: every integer greater than 1 can be factored as a product of prime integers, essentially in only one way.

(x) Next the family Q of rational numbers is defined as the set of equivalence classes arising from the set of all pairs [n,m] of signed integers for which m /= 0. To do this we consider the equivalence relationship Same_frac([n,m],[n',m']) := n * m' = n' * m. The mapping n :-> [n,1], whose inverse is simply car(x), embeds the signed integers into the rationals in a manner preserving all elementary algebraic operations,

and also preserving order. From the fact that the set of signed integers is an ordered integral domain we easily prove that the rationals are an ordered field.

(xi) Our next step, following Cantor, is to define real numbers as equivalence classes of 'Cauchy sequences' s_i of rationals. Here, a sequence is a 'Cauchy sequence' if it satisfies

```
(FORALL x in Q | (EXISTS n in Z | (FORALL i, j in Z |       (x > 0 & i > n &  j > n) *imp
```

The equivalence relation used is

```
Same_real(s,t) = (FORALL x in Q | (EXISTS n in Z |       (x > 0 & i > n &  j > n) *imp
```

Arithmetic operations for these equivalence classes are easily derived from the corresponding functions for rationals, and the 'completeness' of the set of real numbers, a key goal of early 19-th century foundational work on analysis, can be proved without difficulty.

Since it is required for the elementary discussion of complex numbers, we prove the existence and basic properties of the square root, which is shown to exist for any non-negative real number.

(xii) Next the complex numbers are introduced as pairs of real numbers, and their elementary properties are established. In particular, they are shown to constitute a field, within which the field of real numbers has a natural embedding. The modulus of a complex number is defined and its basic properties demonstrated.

(xiii) This completes our preliminary work. What remains is to give the formal details of those parts of standard mathematical analysis needed to state and prove our assigned target result, the Cauchy integral theorem. For this, various familiar results concerning differentiation and integration are needed, first for functions of a real variable, then for functions of a complex variable. Our approach is as follows. The space of all real functions of a real variable is defined, along with the (pointwise) operations of addition, subtraction, and multiplication for functions, function comparison, the positive part of a function, and the least upper bound of a set of functions. Various elementary facts concerning this space of functions are established. In particular, it is shown that they form a ring under addition and multiplication. This allows application of the previously developed 'theory of sigma' to define the sum of an arbitrary finite sequence of real functions. In preparation for the definition of the (ordinary Lebesgue) integral, the sum of an absolutely convergent series of positive real numbers is defined, and the basic properties of such sums are established. This prepares for definition of the sum of an absolutely convergent series of positive real functions, and for a proof of a few basic properties of such series.

In more direct preparation for definition of the integral, we define 'block' functions as real-valued functions of a real variable which are constant inside some finite interval of the real axis, and zero outside this interval. The integral of such a function is simply the area under its graph, which is an elementary rectangular block.

The greatest lower bound of a set of real numbers bounded below is then defined. This is immediately used to define the (Lebesgue) 'upper' integral of an arbitrary non-negative real function of a real variable. This is the greatest lower bound of the sum the integrals of all infinite sequences of non-negative block functions, extended over all such sequences whose (pointwise) sum exceeds the value f(x) at each real point x. Using this, we can define the integral of an arbitrary real function f (which now can have values of both signs) as the difference of the upper integrals of its positive and negative parts.

A function f of a real value is defined to be continuous if it satisfies the standard 'epsilon-delta' condition. To define the derivative of such functions by the technique we adopt, the extension of this definition to the space of real-valued functions of two real variables is needed. To set this up, we first define n-dimensional Euclidean space as the set of all real-valued maps whose domain is the set of integers less than n. The standard Euclidean distance function is defined in this space and its basic properties are proved. Once this has been done, the space of continuous real-valued functions on a Euclidean space of any number of dimensions can be defined by extending the 'epsilon-delta' formulation to this slightly more general setting. We can then define a real-valued function f of one real variable to be (continuously) differentiable if there exists a real-valued function g of two real variables such that (x - y)g(x,y) = f(x) - f(y) for all real x and y. We prove that if such a g exists it is unique, in which case we define the derivative of f as the function h of one variable satisfying h(x) = g(x,x).

Next this whole discussion is carried over to complex functions of a complex variable. We successively define the space of all such functions, the complex Euclidean space of n dimensions with its norm, and the sum, difference, and product for complex-valued functions, either of a single complex variable, or of a point in complex Euclidean space. The 'epsilon-delta' definition of continuity is extended to the complex case for both these classes of functions. This allows direct extension of the notion of derivative, and of its elementary properties, to complex-valued functions of a complex variable.

A set of points in the complex plane is defined to be open if it is the union of the interiors of a set of circles, and a complex function defined in such a set is defined to be analytic if it is differentiable within the set.

Next we define the complex exponential function cexp as the unique complex function analytic everywhere in the complex plane and satisfying the equations Dcexp= cexp and cexp([0,0]) = [1,0], where Dcexp denotes the derivative of cexp. The constant pi is then defined as the smallest positive real root of cexp([0,x]) = [-1,0].

Directly after this, we define the notion of a continuous complex function of a real variable by extending the 'epsilon-delta' formulation to this case in the obvious way. A similar extension of the construction used in the real case gives us the notion of a differentiable complex-valued function of a real variable (i.e. of a smooth curve in the complex plane), and of its derivative. The complex line integral of a complex function g defined on such a curve is then taken to be the ordinary integral of the complex product of g by Df (where as before Df is the derivative of f); the integral of the complex-valued function h = g * Df (which is a function of a single real variable) is by definition obtained by adding the real integrals of the real and imaginary parts of h. We show that the line integrals of an analytic function g over any two curves lying in its domain of analyticity are the same, provided that the two curves lie sufficiently close to one another. Using this, we show that the line integral over the periphery of the unit circle of the quotient function f/(z - w) is 2*pi*i*f(w) for every function f analytic in an open set including the unit circle and its interior, and for every point interior to the unit circle.

Satisfied with this somewhat special form of the Cauchy integral theorem, we rest from our labors.

Chapter 2. Propositional and Predicate-calculus preliminaries.

This chapter prepares for the extensive account of our verifier system given in Chapter 3 by describing and analyzing two of the system's basic ingredients, the *propositional calculus* from which we take all necessary properties of the logical operations &, or, not, *imp, and *eq, and the (first order) *predicate calculus*, which to these propositional mechanisms adds compound functional and predicate constructions

and the two quantifiers FORALL and EXISTS.

Why predicate calculus? Our aim is to develop a mechanism capable of ensuring that the logical formulae in which we are interested are universally valid. Since, as we shall see in Chapter 4, there can exist no algorithm capable of making this determination in all cases, we must use the mechanism of proof. This embeds the formulae in which we are interested in some system of sequences of formulae, within which we can define a property Is_a_proof(p) capable of being verified by an algorithm, such that we can be certain that the final component t of any sequence p satisfying Is_a_proof(p) is universally valid. Then we can use intuition freely to find aesthetically pleasing sequences p, the proofs, leading to interesting end goals t, the theorems. In principle, any system of formulae and sequences of formulae having this property is acceptable. The propositional/predicate calculus and set theory in which we work is merely one such formalism, of interest because of its convenience and wide use, and because much effort has gone into ensuring its reliability.

2.1. The propositional calculus

The propositional calculus constitutes the 'bottom-most' part of the full logical formalism with which we will work in this book. It provides only the operations &, or, not, *imp, and *eq, and the two constants 'true' and 'false', all other symbolic constructions being reduced ('blobbed') down to single letters when propositional deductions must be made. An example given earlier, i.e. the formula

```
(F(x + y) = F(F(x)) *imp F(F(x)) = 0) *imp        (F(F(x)) /= 0) *imp (F(x + y) /= F(F(x)))
```

whose 'blobbed' propositional skeleton is

(p *imp q) *imp ((not q) *imp (not p)),

illustrates what is meant.

Formulae of the propositional calculus are built starting with string names designating propositional variables and combining them using the dyadic infix operators '&', 'or', '*imp', and '*eq' and the monadic operator 'not'. Parentheses are used to group the subparts of formulae. The only precedence relation supported is the rule that '&' binds more tightly than 'or', so parentheses must normally be used rather liberally. Syntactically, the propositional calculus is a simple operator language, whose (syntactically valid) formulae parse unambiguously into syntax trees, each of whose internal nodes is marked either with one of the allowed infix operators, in which case it has two descendants, or with the monadic operator 'not', in which case it has one descendant. Each leaf of such a tree is marked either with the name of a propositional variable or with one of the two allowed constant symbols 'true' and 'false'.

An example is

(pan *imp quack) *imp ((not quack) *imp (not true)).

Here the propositional variables which appear are 'pan' and 'quack', and the constant 'true' also appears.

Since the derivation of the syntax tree of a propositional formula from its string form ('parsing') and of the string form from the syntax tree ('unparsing') are both standard programming operations, we generally regard these two structures as being roughly synonymous and use whichever is convenient without further ado.

As in other logical systems we can think of our formulae either in terms of the values of functions which they represent, or as statements deducible from one another under certain circumstances, and so as the ingredients of some system of formalized proof. We begin with the first approach. In this way of looking at things, each propositional variable represents one of the truth values 1 or 0, which the propositional operators combine in standard ways. The following more formal definition captures this idea:

Definition: An *assignment* for a collection of propositional formulae is a single-valued function A mapping each of its constants and variables into one of the two values 1 and 0. Each assignment is required to map 'true' into 1 and 'false' into 0. The assignment is said to *cover* each of the formulae in the collection.

Given any such assignment A, and a formula F which it covers, the value *Val(A,F) of the assignment A for the expression F* is the Boolean value defined in the following recursive way.

(i) If the formula F is just a variable x or is one of the constants 'true' and 'false', then Val(A,F) = A(x).

(ii) If the formula F has the form 'G & H', then Val(A,F) is the minimum of Val(A,G) and Val(A,H).

(iii) If the formula F has the form 'G or H', then Val(A,F) is the maximum of Val(A,G) and Val(A,H).

(iv) If the formula F has the form 'not G', then `Val(A,F) = 1 - Val(A,G).`

(v) If the formula F has the form 'G *imp H', then `Val(A,F) = Val(A,'(not G) or H').`

(vi) If the formula F has the form 'G *eq H', then `Val(A,F) = Val(A,'(G & H) or ((not G) & (not H))').`

Definition: A propositional formula F is a *tautology* if Val(A,F) = 1 for all the assignments A covering it.

So tautologies are propositional formulae which evaluate to true no matter what truth values are assigned to their variables. Examples are

```
p or (not p) ,    q *imp (p *imp q) ,    p *imp (q *imp (p & q)) ,
```

and many others, some listed below. These are the propositional formulae which possess 'universal logical validity'.

Since the number of possible assignments A for a propositional formula F is at most 2^n, where n is the number of variables in the formula, we can determine whether F is a tautology by evaluating Val(A,F) for all such A. An alternative approach is to establish a system of proof by singling out some initial collection of tautologies (which we will call 'axioms') from which all remaining tautologies can be derived using rules of inference, which must also be defined. (This is the 'logical system' approach). The axioms and rules of inference can be chosen in many ways. Though not at all the smallest possible set, the following collection has a familiar and convenient algebraic flavor.

```
(i) (p & q) *eq (q & p)  (ii) ((p & q) & r) *eq (p & (q & r))  (iii) (p & p) *eq p  (iv) (p or q)
```

The preceding are to be understood as axiom 'templates' or 'schemas', in the sense that all formulae resulting from one of them by substitution of syntactically legal propositional formulae P,Q,... for the letters p,q,... occurring in them are also axioms. For example,

```
(((p or q) or (r *imp r)) & ((p or q) or          (r *imp r))) *eq ((p or q) or (r *imp r))
```

is a substituted instance of (iii) and therefore is also regarded as an axiom.

The reader can verify that all of the axioms listed are in fact tautologies.

In the presence of this lush collection of axioms we need only one rule of inference (namely the 'modus ponens' of mediaeval logicians). From any two formulae of the form

```
p
```

and

```
p *imp q
```

this allows us to deduce q. As with the axioms, this rule is to be understood as a template, covering all of its substituted instances.

To ensure that the tautologies are exactly the derivable propositional formulae we must prove that (I) only tautologies can be derived, and (II) all tautologies can be derived. (I) is easy. We reason as follows. All the axioms are tautologies. Moreover, since

```
Val(A,p *imp q) = Max(1 - Val(A,p),Val(A,q)),
```

it follows that if Val(A,p *imp q) and Val(A,p) are both 1, so is Val(A,q). So if 'p *imp q' and p are both tautologies, then so is q. This proves our claim (I).

Proving claim (II) takes a bit more work, whose general pattern is much like that used to reduce multivariate polynomials to their canonical form. Starting with any syntactically well formed propositional formula F, we can proceed in the following way to derive a chain of formulae equivalent to F (via an explicit chain of equivalences F_i *eq F_{i+1}). Note that axioms (xviii-xx) ensure that the equivalence relator '*eq' has the same transitivity, symmetry, and reflexivity properties as equality, while (xi-xiii) allow us to replace any subexpression of an expression formed using only the three operators &, or, not by any equivalent subexpression.

Using these facts and (xv-xvi) we first descend recursively through the syntax tree of F, replacing any occurrence of one of the operations *imp, *eq by an equivalent expression involving only &, or, not. This reduces F to an equivalent formula involving only the operators &, or, not. Then, using (vii-viii) and (x), we systematically push 'not' and 'or' operators down in the syntax tree, moving '&' operators up. Subformulae of the form (not (not p)) are simplified to p using axiom (xxiii). Axioms (xxiv-xxix) can be used to simplify expressions containing the constants 'true' and 'false'. When this work is complete F will been have reduced to an equivalent formula F' which is either one of the constants 'true' or 'false' or has the form a_1 & ... & a_k, where each a_j is a disjunction of the form

```
b1 or ... or bh,
```

each b_m being either a propositional variable or the negation of a propositional variable. (ii) and (v) allow us to think of these conjunctions and disjunctions without worrying about how they are parenthesized.

Then (iv) and (vi) can be used to bring all the b_m involving a particular propositional variable together within each a_j.

Now assume that F is a tautology, so that every one of the formulae to which we have reduced it must also be a tautology (since the substitutions performed all convert tautologies to tautologies), and so our final formula F' is a tautology. We will now further reduce F', so that it becomes the formula 'true'. Unless F' is already 'true', in each a_j, there must occur at least one pair b_m, b_n of disjuncts such that b_m is a propositional variable of which b_n is the negation, 'not b_m'. Indeed, if this is not the case, then any propositional variable which occurs in a_j will occur either negated or non-negated, but not both. Given this, we can assign the value 0 to each non-negated variable and the value 1 to each negated variable. Then every b_m in a_j will evaluate to 0, so the whole expression b_1 or ... or b_h will evaluate to 0, that is, a_j will evaluate to 0. But as soon as this happens the whole formula a_1 & ... & a_k will evaluate to 0. This shows that there exists an assignment A such that Val(A,F') = 0, contradicting the fact that F' is a tautology. This contradiction proves our claim that each a_j must contain at least one pair b_m, b_n of disjuncts which agree except for the presence of a negation operator in one but not in the other.

Given this fact, (xxii) tells us that 'b_m or b_n' simplifies to 'true', so that (xxvi) can be used repeatedly to simplify a_j to 'true'. Since this is the case for each a_j, repeated use of (xxiv) allows us to reduce any tautology to 'true' using a chain of equivalences. Since this chain of equivalences can as well be traversed in the reverse direction, we can equally well expand the axiom 'true' (axiom (xxx)) into our original formula F using a chain of equivalences. Then (xiv) can be used to convert this chain of equivalences into a chain of implications, giving us a proof of F by repeated uses of modus ponens.

Any set of axioms from which all the statements (i-xxx) can be derived as theorems can clearly be used as an axiomatic basis for the propositional calculus. This allows much leaner sets of axioms to be used. We refrain from exploring this point, which lacks importance for the rest of our discussion.

However, it is worth embedding the notion of 'tautology' in a wider, relativized, set of ideas. Suppose that we write

```
|= F
```

to indicate that the formula F is a tautology, and

```
|- F
```

to indicate that F is a provable formula of the propositional calculus. The preceding discussion shows that |= F and |- F are equivalent conditions. This result can be generalized as follows. Let S designate any finite set of syntactically well-formed formulae of the propositional calculus. We can then write

```
S |= F
```

to indicate that, for each assignment A covering both F and all the formulae in S, we have Val(A,F) = 1 whenever Val(A,G) = 1 for all G in S. Also, we write

```
S |- F
```

to indicate that F follows by propositional proof if the statements in S are added to the axioms of propositional calculus (each of them acting as an individual axiom, not as a template). Then it is easy to show that

```
S |= F if and only if S |- F.
```

To show this, first suppose that S |= F. Let C designate the conjunction

```
G1 & ... & Gk
```

of all the formulae in S. Then since Val(A,H_1 & H_2) = Min(Val(A,H_1),Val(A,H_2)) for any two formulae H_1,H_2, it follows that Val(A,C) = 1 if and only if Val(A,G) = 1 for all G in S. We have

```
Val(A,C *imp F) = Val(A,(not C) or F)      = Max(1 - Val(A,C),Val(F))
```

for all assignments A covering C *imp F (i.e. covering both F and all the formulae in S). It follows that for each assignment A covering both F and all the formulae in S, we have Val(A,C *imp F) = 1, since if 1 - Val(A,C) /= 1 then Val(A,C) must be 1 and so Val(F) must be 1. Thus

```
|= C *imp F,
```

and so it follows that

```
|- C *imp F,
```

i.e. C *imp F can be proved from the axioms of propositional calculus alone. But then if the statements in S are added as additional axioms we can prove F by first proving C *imp F and then using the statements in S to prove the conjunction C. This shows that S |= F implies S |- F.

Next suppose that S |- F, and let A be an assignment A covering both F and all the formulae in S so that Val(A,G) = 1 for every statement G in S. Then Val(A,G) = 1 for every statement G that can be used as an axiom in the proof of F from the standard axioms of propositional calculus and the statements in S as additional axioms. But we have seen above that if Val(A,p *imp q) and Val(A,p) are both 1, so is Val(A,q). Since derivation of q from p and p *imp q is the only inference step allowed in propositional calculus proofs, it follows that S |= F, completing our proof that the conditions S |= F and S |- F are equivalent.

We shall see that similar statements apply to the much more general predicate calculus studied in the following section. In that section, we will need the following extension of the preceding results to countably infinite collections of propositional formulae.

Definition. A (finite or infinite) collection S of formulae of the propositional calculus is said to be *consistent* if the proposition 'false' cannot be deduced from S, i.e.

```
S |- false
```

is false. We say that *S has a model A* if there exists some assignment A covering all the formulae of S such that Val(A,F) = 1 for every F in S.

Theorem (Compactness): Let S be a denumerable collection of formulae of the propositional calculus. Then the following three conditions are equivalent:

(i) S is consistent.

(ii) Every finite subset of S is consistent.

(iii) S has a model.

Proof: Since subsets of a consistent S are plainly consistent, (i) implies (ii). On the other hand, any proof of 'false' from the statements of S is of finite length by definition, and so uses only a finite number of the statements of S. Thus (ii) implies (i), so (ii) and (i) are equivalent.

Next suppose that S is not consistent, so that 'false' can be proved from some finite subset S' of the statements in S. Let C be the conjunction of all the statements in S'. It follows from the discussion immediately preceding the statement of the present theorem that |- C *imp false, and so Val(A,'C *imp false') = 1 for any assignment A covering all the propositional symbols in S. This gives Val(A,C) = 0 for all such A, so that S has no model. This proves that (iii) implies (i).

Next we show that (i) implies (iii). For this, let S_j be an increasing sequence of finite subsets of S whose union is all of S. Each S_j is plainly consistent, so

```
Sj |- false
```

is false for each j, and therefore

```
Sj |= false
```

is false, since we have shown above that these two conditions are equivalent for finite S_j. That is, for each j there must exist an assignment A_j covering all the variables appearing in any formula of S_j, such that $Val(A_j, S_j) = 1$. Let v_1, v_2, v_3,... be an enumeration of all the variables appearing in any of the formulae of S. Then each v_k must be in the domain of all A_j for all j beyond a certain point $J = J_k$.

Let I_0 designate the sequence of all integers. Since $A_j(v_1)$ must have one of the two values 0 and 1, there must exist an infinite subsequence I_1 of I_0 for all j of which $A_j(v_1)$ has the same value. Call this value $B(v_1)$. Arguing in the same way we see that here must exist an infinite subsequence I_2 of I_1 and a Boolean value $B(v_2)$ such that

```
B(v2) = Aj(v2) for all j in I2.
```

Arguing repeatedly in this way we eventually construct values $B(v_k)$ for each k such that for each finite m, there exist infinitely many j such that

```
B(vn) = Aj(vn) for all n from 1 to m.
```

Now consider any of the formulae G of S. Since G can involve only finitely many propositional variables v_j, all its variables will be included in the set $\{v_1,...,v_k\}$ for each sufficiently large k. Take any A_j for which $B(v_n) = A_j(v_n)$ for all n from 1 to k. Then it is clear that for some i greater than j, we have

```
Val(B,G) = Val(Ai,G) = 1.
```

Hence Val(B,G) = 1 for all G in S, so that B is a model of S, proving that (i) implies (iii), and thereby completing the proof of our theorem. **QED**

Using the Compactness Theorem, we can show that the conditions S |- F and S |= F are equivalent even in the case in which S is an infinite set of propositional formulae.

To show this, first assume that S |= F. Then the set S + {not F} of propositions is plainly not consistent, and so by the Compactness Theorem S must contain some finite subset S_0 such that S_0 + {not F} is not consistent. Then plainly S_0 |= F, so we have S_0 |- F. This clearly implies S |- F; so S |- F follows from S |= F.

But, as noted at the end of the proof of the Compactness Theorem, S |= F follows from S |- F even if S is infinite, completing the proof of our claim.

2.2. The predicate calculus

The predicate calculus constitutes the next main part of the logical formalism used in this book. This calculus enlarges the propositional calculus, preserving all its operations but also allowing compound functional and predicate terms and the two quantifiers FORALL and EXISTS. An example is the formula

```
((FORALL x,y | F(x + y) = F(F(x))) *imp F(F(x)) = 0) *imp          ((EXISTS x | F(F(x)) /= 0) *imp
```

Formulae of the predicate calculus are built starting with string names of three kinds, respectively designating 'individual' variables, function symbols, and predicate symbols. These are combined into '*terms*', '*atomic formulae*', and '*formulae*' using the following recursive syntactic rules.

(i) Any variable name is a term. (We assume variable names to be alphanumeric and to start with lower case letters).

(ii) Each function symbol has some fixed finite number k of arguments. If f is a function symbol of k arguments, and $t_1,...,t_k$ are any k terms, then $f(t_1,...,t_k)$ is a term. (We assume function names to be alphanumeric and to start with lower case letters).

(iii) Each predicate symbol has some fixed finite number k of arguments. If P is a predicate symbol of k arguments, and $t_1,...,t_k$ are any k terms, then $P(t_1,...,t_k)$ is an atomic formula. (We assume predicate names to be alphanumeric and to start with upper case letters).

(iv) Formulae are formed starting from atomic formulae and using the operators and syntactic rules of the propositional calculus and the two quantifiers FORALL and EXISTS. More precisely, if e and f are any two predicate formulae and $v_1,...,v_n$ are any n variable names, with n>0, then the following expressions are predicate formulae:

```
e & f,   e or f,    e *imp f,    e *eq f,    not e,    (FORALL v1,...,vn | e),    (EXISTS v1,
```

Like propositional formulae, the formulae of predicate calculus parse unambiguously into syntax trees each of whose internal nodes is marked either (i) with one of the propositional operators, and then has as many descendants as the corresponding propositional node, or (ii) with a function or predicate symbol, in which case its descendants correspond to the arguments of the function or predicate symbol; (iii) a quantifier FORALL or EXISTS involving n variable names, in which case the node has n + 1 descendants, the first n marked with the n variable names appearing in the quantifier and the n + 1-st which is the syntax tree of the expression e that is being quantified. Each leaf of such a tree is marked either with the name of an individual variable or a function symbol of zero arguments. (Such function symbols are called 'constants').

Each occurrence of a variable v at a leaf of the syntax tree of a valid predicate formula is either *free* or *bound*. A variable v is considered to be bound if it appears as the descendant of some syntax tree node which is marked with a quantifier in whose associated list of variables v occurs; otherwise the occurrence is a free occurrence. These notions clearly translate back into corresponding notions for variable occurrences in the unparsed string forms of the same formulae. For example, in the predicate formula

```
(FORALL x,z,x | F(x + y + z)) or (EXISTS y,y | F(x + y))
```

the first three occurrences of x are bound, but the fourth occurrence of x is free. Likewise the last three occurrences of y are bound, but its first occurrence is free. Note that, as this example shows, repeated occurrences of a variable in the list following one of the quantifier symbols FORALL or EXISTS are legal. However, we will see, when we come to define the semantics of predicate formulae, that such repetitions are always superfluous since any variable occurrence repeated later in the list following a quantifier symbol can simply be dropped. For example, the formula shown above has the same meaning as

```
(FORALL z,x | F(x + y + z)) or (EXISTS y | F(x + y)).
```

Bound variables are considered to belong to the *scope* of the nearest ancestor quantifier in whose list of variables they appear; this quantifier is said to *bind* them. For example, in

```
(FORALL x | F(x) or (EXISTS x | G(x)) or H(x))
```

the first, second, and final occurrences of x are in the scope of the first quantifier 'FORALL', but the third and fourth occurrences are in the scope of the second quantifier 'EXISTS'.

As was the case for the propositional calculus, we can think of predicate formulae either as representing certain functions, or as the ingredients of a system of formalized proof. Again we begin with the first approach. Here the required definitions are a bit trickier.

Definition: An *interpretation framework* for a collection PF of predicate formulae is a triple (U,I,A) such that

(i) U is a nonempty set, called the *universe* or *domain* of the interpretation framework. We write U^k for the k-fold Cartesian product of U with itself.

(ii) I is a single-valued function, called an *interpretation*, which maps each of the function and predicate symbols occuring in the collection in accordance with the following rules:

(ii.a) Each function symbol f of k arguments occurring in the collection of formulae is mapped into a function I(f) which sends U^k into U.

(ii.b) Each predicate symbol P of k arguments occurring in the collection of formulae is mapped into a function I(P) which sends U^k into the set {0,1} of values.

(iii) A is a single-valued function, called an *assignment*, which maps each of the individual variables occurring freely in the collection PF of formulae into an element of U.

As previously we speak of such an interpretation framework as *covering* the collection PF of predicate formulae.

Suppose that we are given any such interpretation I and assignment A with universe U, and an expression F which they cover. (Note that F can be either a term or a predicate formula). Then the value *Val(I,A,F) of the assignment for the expression* is the value defined in the following recursive way.

(i) If F is just an individual variable x, then Val(I,A,F) = A(x).

(ii) If F is a term having the form $g(t_1,...,t_k)$, and G is the corresponding mapping I(g) from U^k to U, then $Val(I,A,F) = G(Val(I,A,t_1),...,Val(I,A,t_k))$.

(iii) If F is an atomic formula having the form $P(t_1,...,t_k)$, and p is the corresponding mapping I(P) from U^k to {0,1}, then Val(I,A,F) is the 0/1 value $p(Val(I,A,t_1),...,Val(I,A,t_k))$.

(iv) If F is a formula having the form 'G & H', then Val(I,A,F) is the minimum of Val(I,A,G) and Val(I,A,H).

(v) If F is a formula having the form 'G or H', then Val(I,A,F) is the maximum of Val(I,A,G) and Val(I,A,H).

(vi) If F is a formula having the form 'not G', then Val(I,A,F) = 1 - Val(I,A,G).

(vii) If F is a formula having the form 'G *imp H', then Val(I,A,F) = Val(I,A,'(not G) or H').

(viii) If F is a formula having the form 'G *eq H', then Val(I,A,F) = Val(I,A,'(G & H) or ((not G) & (not H))').

(ix) If F is a formula having the form (FORALL $v_1,...,v_n$ | e), then Val(I,A,F) is the minimum of Val(I,A',e), extended over all assignments A' such that A' covers the formula e and A'(x) = A(x) for every variable x not in the list $v_1,...,v_n$.

(x) If F is a formula having the form (EXISTS $v_1,...,v_n$ | e), then Val(I,A,F) is the maximum of Val(I,A',e), extended over all assignments A' such that A' covers the formula e and A'(x) = A(x) for every variable x not in the list $v_1,...,v_n$.

Since, as seen in (ix) and (x) above, the variables appearing in the lists following quantifier symbols 'FORALL' and 'EXISTS' merely serve to mark occurrences of the same variables in the quantifier's scope as being 'bound' and hence subject to minimization/maximization when values Val(I,A,F) are calculated, it follows that these variables can be replaced with any others provided that this replacement is made uniformly over the entire scope of each quantifier, and that no variable occurring freely in the original formula thereby becomes bound. For example, the formula

```
(FORALL x | F(x) or (EXISTS x | G(x)) or H(x))
```

appearing above can as well be written as

```
(FORALL x | F(x) or (EXISTS y | G(y)) or H(x))
```

or as

```
(FORALL y | F(y) or (EXISTS x | G(x)) or H(y)).
```

A convenient way of performing this kind of 'bound variable standardization' is as follows. We make use of some standard list L of bound variable names, reserved for this purpose and used for no other. We work from the leaves of a formula's syntax tree up toward its root, processing all quantifiers more distant from the root before any quantifier closer to the root is processed. Suppose that a quantifier like

```
(FORALL v1,...,vn | e)
```

or

```
(EXISTS v1,...,vn | e)
```

is encountered at a tree node Q during this process. We then take the first n variables $b_1,...,b_n$ from the list L that do not already appear in any descendant of the node Q, replace $v_1,...,v_n$ by $b_1,...,b_n$ respectively, and make the same replacements for every free occurrence of any of the $v_1,...,v_n$ in e.

This standardization will for example transform

```
(FORALL y | (FORALL y | F(y) or (EXISTS x | G(x))) or H(y))
```

into

```
(FORALL b3 | (FORALL b1 | F(b1) or (EXISTS b2 | G(b2))) or H(b3)).
```

Such standardization of bound variables makes it easier to see what quantifier each bound variable occurrence relates to. It also uncovers identities between quantified subexpressions that might otherwise be missed, and so is a valuable preliminary to examination of the propositional structure of predicate formulae.

It also follows from (ix) and (x) that the value assigned to any quantified formula

```
(+) (FORALL v1,v2,...,vn | e)
```

is exactly the same as that assigned to

```
(++) (FORALL v1 | (FORALL v2 | (FORALL ... | (FORALL vn | e)...)))
```

and, likewise, the value assigned to any quantified formula

```
(*) (EXISTS v1,v2,...,vn | e)
```

is exactly the same as that assigned to

```
(**) (EXISTS v1 | (EXISTS v2 | (EXISTS ... | (EXISTS vn | e)...)))
```

Accordingly, we shall regard (+) and (*) as abbreviations for (++) and (**). This allows us to assume (wherever convenient) that each quantifer examined in the following discussion involves only a single variable.

Definition: A predicate formula F is *universally valid* if Val(I,A,F) = 1 for every interpretation framework (U,I,A) covering it.

In predicate calculus, universally valid formulae are those which evaluate to true no matter what 'meanings' are assigned to the variables, function symbols, and predicate symbols that occur within them. Examples are

```
P(x,y) or (not P(x,y)),(FORALL y | Q(x) *imp (P(x,y) *imp Q(x))),(FORALL x | P(x,y) *imp (EXISTS y
```

However, the problem of determining whether a given predicate formula is universally valid is of a much higher order of difficulty than the problem of recognizing propositional tautologies, since the collection of

interpretation frameworks that must be considered is infinite rather than finite. There is no longer any reason for believing that this determination can be made algorithmically, and indeed it cannot, as we shall see in Chapter 4. Thus we have little alternative to setting up the predicate calculus as a logical system in which universally valid formulae are found by proof. We now begin to do this, starting with a special subclass of universally valid formulae, the *predicate tautologies*, which are defined as follows.

Definition: A predicate formula F is a *tautology* if it reduces to a propositional tautology by descending through its syntax tree and reducing each node not marked with a propositional operator to a single propositional variable, identical subnodes always being reduced to the same propositional variable. (In what follows we will call this latter formula the *propositional blobbing* of P).

As an example, note that the indicated reduction sends

```
P(x,y) or (not P(x,y)) into A or (not A),(FORALL y | Q(x) *imp (P(x,y) *imp Q(x))) into B,P(x,y) *i
```

Thus the first of these three formulae is a predicate tautology, but the two others are not.

The recursive computation of Val(I,A,F) assigns some 0/1 value to each subtree of the syntax tree of F, and plainly assigns the same value to identical subtrees of the syntax tree of F. This makes it clear that every predicate tautology is universally valid. But there are other basic forms of universally well-formed predicate formulae, of which the most crucial are listed in the following definition.

Definition: A formula is an *axiom of the predicate calculus* if it is either

(i) any predicate tautology;

(ii) any formula of the form

((FORALL y | P *imp Q) & (FORALL y | P)) *imp (FORALL y | Q);

(iii) any formula of the form

(not (FORALL y | not P)) *eq (EXISTS y | P);

(iv) any formula of the form P *eq (FORALL y | P), where the variable y does not occur in P as a free variable;

(v) any formula of the form (FORALL y | P) *imp P(y-->e), where P(y-->e) is the formula obtained from P by substituting the syntactically well-formed term e for each free occurrence of the variable y in P, provided that no variable free in e is bound at the point of occurrence of any such y in P.

We can easily see that all of these predicate axioms are universally valid. Given a formula P of the predicate calculus, let P' designate its propositional blobbing. Predicate tautologies are universally valid since the final stages of computation of Val(I,A,P) always use the values assigned to certain basic subformulae of P in the same way that values assigned to corresponding propositional variables are used in the propositional computation of Val(I,A,P'). To see that (iii) is universally valid, we have only to note that for 0/1 valued functions f of any number of arguments we always have

```
Max(f) = 1 - Min(1 - f).
```

(iv) is universally valid because if y does not occur in P as a free variable, we have

```
Val(I,A,'(FORALL y | P)') = Val(I,A,P)
```

for every interpretation I and assignment A covering P.

(v) is universally valid because any interpretation I and assignment A covering P(y-->e) will assign some value a_0 to e, and then Val(I,A,P(y-->e)) = Val(I,A',P), where A' is the assignment identical to A except that it assigns the value a_0 to y. Since Val(I,A',(FORALL y | P)) is by definition the minimum of Val(I,B,P) extended over all assignments B which are identical to A except on the variable y, it follows that Val(I,A,'(FORALL y | P)') = 1 implies Val(I,A,P(y-->e)) = 1, so that

```
Max(1 - Val(I,A,'(FORALL y | P)'),Val(I,A,P(y-->e)))
```

is identically 1, i.e. (FORALL y | P) *imp P(y-->e) is universally valid.

To show that (ii) is universally valid, note that for any interpretation I and assignment A covering (ii)

```
Val(I,A,'(FORALL y | P *imp Q)')
```

and

```
Val(I,A,'(FORALL y | P)')
```

are respectively the minimum of Max(1 - Val(I,A',P),Val(I,A',Q)) and of Val(I,A',P), extended over all assignments A' which are identical to A except on the variable y. If both of these minima are 1, then 1 - Val(I,A',P) must be 0 for all such A', so Val(I,A',Q) must be 1 for all such A', proving that Val(I,A,'(FORALL y | Q)') = 1. This implies the universal validity of (ii), completing our proof that all predicate axioms are universally valid.

Proof rules of the predicate calculus

The predicate calculus has just two proof rules. The first is identical with the modus ponens rule of propositional calculus. The second is the *Rule of Generalization,* which states that if P is any previously proved result, then

```
    (FORALL x | P)
```

can be deduced.

A stronger variant of the Rule of Generalization, which turns out to be very useful in practice, allows us to deduce the formula

```
   P *imp (FORALL x | Q)
```

from P *imp Q, provided that the variable x does not occur free in P. This variant can be justified as follows. Let us assume that the formula P *imp Q has been derived and that x is a variable which does not have free occurrences in P. By generalization and as instance of the predicate axiom (ii) we can derive the formulae

```
   (FORALL x | P *imp Q),          ((FORALL x | P *imp Q) & (FORALL x | P)) *imp (FORALL x | Q).
```

By propositional reasoning these imply the formula

```
(FORALL x | P) *imp (FORALL x | Q).
```

Since we are assuming that the variable x does not occur free in P, we can derive the formula

```
P *eq (FORALL x | P)
```

using predicate axiom (iv), and it follows by propositional reasoning that

```
P *imp (FORALL x | Q),
```

which establishes the strong form of the rule of generalization that we have stated.

In what follows we will not always distinguish between the two variants of the rule of generalization and we will use whichever version is more convenient for the purposes at hand. The argument given above shows that any proof which uses the strong variant of the Rule of Generalization can be transformed mechanically into a proof which uses only the standard form of this Rule.

We can easily see that any formula deduced from universally valid formulae using the two proof rules just explained must also be universally valid. For the modus ponens rule this follows as in the propositional case. For the rule of generalization we reason as follows. If Val(I,A,P) = 1 for every interpretation I and assignment A covering P, then since for every assignment B covering (FORALL x | P) the value v = Val(I,B,'(FORALL x | P)') is the minimum of Val(I,A,P) extended over all assignments A which give the same value as B to all variables other than x, it follows that v = 1 also.

In analogy with the case of the propositional calculus we write

```
|= F
```

to indicate that the formula F is a universally valid formula of the predicate calculus, and write

```
|- F
```

to indicate that F is a provable formula of the predicate calculus.

The following very important theorem is the predicate analog of the statement that a propositional formula is a tautology if and only if it is provable.

The Gödel completeness theorem

For any predicate formula, the conditions

```
|= F     and     |- F
```

are equivalent.

Half of this theorem is just as easy to prove as in the propositional case. Specifically, suppose that |- F. Then since all the axioms of predicate calculus are universally valid and the predicate calculus rules of inference preserve universal validity, F must be universally valid, i.e. |= F.

The other, more difficult half of this theorem will be proved later, after some preparation. Much as in the case of the propositional calculus, this result can be generalized as follows. Let S designate any set of syntactically well-formed formulae of the predicate calculus. Write

```
S |= F
```

to indicate that, for each interpretation I and assignment A covering both F and all the formulae in S, we have Val(I,A,F) = 1 whenever Val(I,A,G) = 1 for all G in S. Also, write

```
S |- F
```

to indicate that F follows by predicate proof if the statements in S are added to the axioms of predicate calculus. Suppose that none of the formulae in S contain any free variables (formulae with this property are usually called *sentences*). Then for any predicate formula, the conditions

```
S |= F  and S |- F
```

are equivalent. (An easy example, given below, shows that we cannot omit the condition 'none of the formulae in S contain any free variables.') The derivation of this from the more restricted result given by the Gödel completeness theorem is almost the same as the corresponding propositional proof. For the moment we will consider only the case in which S is finite. Suppose first that S |= F and let C designate the conjunction

```
G1 & ... & Gk
```

of all the formulae in S. Let I and A be respectively an interpretation and an assignment which cover C *imp F (i.e. cover both F and all the formulae in S). Then as in the propositional case it follows that Val(I,A,C) = 1 if and only if Val(I,A,G) = 1 for all G in S. Hence

```
Val(I,A,C *imp F) = Val(I,A,(not C) or F)          = Max(1 - Val(I,A,C),Val(I,A,F)) = 1,
```

for all such I and A. Hence

```
|= C *imp F,
```

follows using the Gödel Completeness Theorem, as stated above, and so it follows that

```
|- C *imp F,
```

i.e. C *imp F can be proved from the axioms of predicate calculus alone. But then if the statements in S are added as additional axioms we can prove F by first proving C *imp F, then using the statements in S to prove the conjunction C, and finally proving F by modus ponens from C *imp F and C. This shows that S |= F implies S |- F.

Next suppose that there exists a formula F such that S |- F, but that S |= F is false. Let F be such a formula with the shortest possible proof from S, and let I and A be respectively any interpretation and assignment A covering both F and all the formulae in S such that Val(I,A,G) = 1 for every statement G in S, but Val(I,A,F) = 0. The final step of a shortest proof of F from S cannot be either the citation of an axiom or the citation of a statement of S, since in both these cases we would have Val(I,A,F) = 1. Hence this final step is either a modus ponens inference from two formulae p, p *imp F appearing earlier in the proof, or a generalization inference from one such formula p. In the modus ponens case we must have S |= p, S |= p *imp F by inductive assumption. Hence Val(I,A,p *imp F) and Val(I,A,p) are both 1, and therefore so is

Val(I,A,F), a contradiction.

In the remaining case, i.e. that of a generalization inference, we must have S |= p, where F has the form (FORALL x | p), for some predicate variable x. Since the statements in S have no free variables we have Val(I,A',G) = 1 for every statement G in S and every assignment A' which is identical to A except on the variable x, so that Val(I,A',p) = 1. But then

```
Val(I,A,'(FORALL x | p)')
```

is the minimum of Val(I,A',p), taken over all such A', and therefore it follows that Val(I,A,'(FORALL x | p)') = 1, i.e. Val(I,A,F) = 1, which is again a contradiction. This shows that S |- F implies S |= F, completing our proof that the conditions S |= F and S |- F are equivalent, at least in the case in which S is finite. We will see later that the condition that the set S is finite can be dropped. In fact, we can notice right away that the derivation given above of S |= F from S |- F holds also in the case in which S is infinite. Thus, in order to fully establish the generalization of the Gödel completeness theorem, we are only left with proving that S |= F implies S |- F, for every infinite set S of predicate formulae none of which has occurrences of free variables.

We conclude this subsection by noting that the result just stated fails if the formulae in S are allowed to contain free variables. To see this, consider the simple case in which S consists of the single formula P(x). If this formula were added to the set of axioms of the predicate calculus, we could give the proof

```
P(x)                          [axiom]    (FORALL x | P(x))          [generalization]    (FORALL x
```

Hence we could have {P(x)} |- P(y). But {P(x)} |= P(y) is false, since we can set up a 2-point universe U = {a,b}, the assignment A(x) = a, A(y) = b, and the interpretation I such that I(P)(a) = 1 and I(P)(b) = 0.

Working with universally valid predicate formulae. A few simple examples of predicate proof.

A few basic theorems of predicate calculus are needed for later use. One such is

```
((FORALL x | P *imp Q) & (EXISTS x | P)) *imp (EXISTS x | Q).
```

The following proof of this statement, and two other sample proofs given later in this section, illustrate some of the techniques of direct, fully detailed predicate proof. By predicate axiom (v) we have

```
(FORALL x | P *imp Q) *imp (P *imp Q),
```

and from this by purely propositional reasoning we have

```
(FORALL x | P *imp Q) *imp ((not Q) *imp (not P)).
```

By the (strong) rule of generalization this gives

```
(FORALL x | P *imp Q) *imp (FORALL x | ((not Q) *imp (not P))).
```

Axiom (ii) now tells us that

```
((FORALL x | ((not Q) *imp (not P)))       & (FORALL x | (not Q))) *imp (FORALL x | (not P)),
```

so by propositional reasoning we have

```
(FORALL x | P *imp Q) *imp       ((FORALL x | (not Q)) *imp (FORALL x | (not P))),
```

and also

```
(FORALL x | P *imp Q) *imp ((not (FORALL x | (not P))) *imp       (not (FORALL x | (not Q)))).
```

Since by predicate axiom (iii) we have

```
(not (FORALL x | (not P))) *eq (EXISTS x | P)
```

and

```
(not (FORALL x | (not Q))) *eq (EXISTS x | Q),
```

our target statement

```
((FORALL x | P *imp Q) & (EXISTS x | P)) *imp (EXISTS x | Q)
```

now follows propositionally.

The following is a useful general principle of the predicate calculus whose universal validity is readily understood intuitively, and which can also be proved formally within the predicate calculus.

Suppose that a predicate formula of the form

```
A *eq B
```

has been proved and that F is a syntactically legal predicate formula such that A appears as a subformula of F. Let G be the result of replacing some such occurrence of A in F by an occurrence of B. Then F *eq G is also a theorem.

To show this, note that F can be built up starting from A by steps, each of which either joins subformulae together using a propositional operator, or quantifies a formula. Hence it is enough to show that if

```
(+)   H2 *eq H3
```

has already been proved, then

```
(a)   (H1 and H2) *eq (H1 and H3)(b)   (H1 or H2) *eq (H1 or H3)(c)   (H1 *eq H2) *eq (H1 *eq H3)(d)
```

can be proved as well. Notice that (a)-(f) follow readily from (+) by propositional reasoning. So to prove our claim we have only to establish that (g) and (h) follow from (+) too. This can be shown as follows. By propositional reasoning and the predicate rule of generalization, statement (+) yields

```
(FORALL x | H2 *imp H3).
```

By axiom (ii) we have

```
((FORALL x | H2 *imp H3) & (FORALL x | H2)) *imp (FORALL x | H3),
```

so by propositional reasoning we get

```
(FORALL x | H2) *imp (FORALL x | H3).
```

The formula

```
(FORALL x | H3) *imp (FORALL x | H2)
```

can be derived in the same way, and so we have

```
(FORALL x | H2) *eq (FORALL x | H3).
```

Since (+) yields

```
(not H2) *eq (not H3)
```

by propositional reasoning, it follows in the same way that

```
(FORALL x | (not H2)) *eq (FORALL x | (not H3))
```

and so

```
(not (FORALL x | (not H2))) *eq (not (FORALL x | (not H3))).
```

It follows by predicate axiom (iii) and propositional reasoning that

```
(EXISTS x | H2) *eq (EXISTS x | H3),
```

completing the proof of our claim.

The following 'change of bound variables' law is still another rule of obvious universal validity, which as usual can be proved formally within the predicate calculus.

Let F be a syntactically well-formed predicate formula containing x as a free variable, let y be a variable not occurring in F, and let F(x-->y) be the result of replacing every free occurrence of x by an occurrence of y. Then

```
(FORALL x | F) *eq (FORALL y | F(x-->y))
```

and

```
(EXISTS x | F) *eq (EXISTS y | F(x-->y))
```

are universally valid predicate formulae. To show this, we first use predicate axiom (v) to get

```
(FORALL x | F) *imp F(x-->y),
```

and so

```
(FORALL x | F) *imp (FORALL y | F(x-->y))
```

follows by the (strong) rule of generalization, since y does not occur freely in (FORALL x | F).

Since replacing each free occurrence of x in F by y and then each y by x brings us back to the original x, we have

```
F(x-->y)(y-->x) = F.
```

Thus the argument just given can be used again to show that

```
(FORALL y | F(x-->y)) *imp (FORALL x | F),
```

and so it results propositionally that

```
(FORALL y | F(x-->y)) *eq (FORALL x | F).
```

Applying the same argument to 'not F' we can get

```
(not (FORALL y | not F(x-->y))) *eq (not (FORALL x | not F)),
```

and so

```
(EXISTS y | F(x-->y)) *eq (EXISTS x | F),
```

using predicate axiom (iii).

The observations just made allow any predicate formula F to be transformed, via a sequence of formulae all provably equivalent to each other, into an equivalent formula G all of whose quantifiers appear to the extreme left of the formula. To achieve this, we must also use the following auxiliary group of predicate rules, which apply if the variable x does not occur freely in Q:

(a) (FORALL x | P or Q) *eq ((FORALL x | P) or Q)

(b) (FORALL x | P & Q) *eq ((FORALL x | P) & Q)

(c) (FORALL x | P *imp Q) *eq ((EXISTS x | P) *imp Q)

(d) (FORALL x | Q *imp P) *eq (Q *imp (FORALL x | P))

(e) (EXISTS x | P or Q) *eq ((EXISTS x | P) or Q)

(f) (EXISTS x | P & Q) *eq ((EXISTS x | P) & Q)

(g) (EXISTS x | P *imp Q) *eq ((FORALL x | P) *imp Q)

(h) (EXISTS x | Q *imp P) *eq (Q *imp (EXISTS x | P))

These rules can be proved as follows. Predicate axiom (v) gives

```
(FORALL x | P) *imp P,
```

and so

```
(FORALL x | P) *imp (P or Q)
```

by propositional reasoning. Also we have Q *imp (P or Q), and so by propositional reasoning we have

```
((FORALL x | P) or Q) *imp (P or Q).
```

Since x does not occur freely in ((FORALL x | P) or Q), generalization now gives

```
((FORALL x | P) or Q) *imp (FORALL x | P or Q).
```

Conversely we get

```
(FORALL x | P or Q) *imp (P or Q)
```

from predicate axiom (v), and so

```
((FORALL x | P or Q) & (not Q)) *imp P.
```

Since x does not occur freely in ((FORALL x | P or Q) & (not Q)), by generalization we get

```
((FORALL x | P or Q) & (not Q)) *imp (FORALL x | P),
```

and then

```
(FORALL x | P or Q) *imp ((FORALL x | P) or Q),
```

so altogether

```
(FORALL x | P or Q) *eq ((FORALL x | P) or Q),
```

proving (a).

To prove (b) we reason as follows.

```
(FORALL x | P & Q) *imp (P & Q)
```

by axiom (v), so

```
(FORALL x | P & Q) *imp P
```

by propositional reasoning. Since x does not occur freely in (FORALL x | P & Q), by generalization we derive

```
(FORALL x | P & Q) *imp (FORALL x | P)
```

from this. Thus, by propositional reasoning, we obtain

```
(FORALL x | P & Q) *imp ((FORALL x | P) & Q).
```

Conversely, since

```
((FORALL x | P) & Q) *imp (FORALL x | P)
```

we have

```
((FORALL x | P) & Q) *imp P
```

by axiom (v) and propositional reasoning. Since

```
((FORALL x | P) & Q) *imp Q
```

is propositional, we get

```
((FORALL x | P) & Q) *imp (P & Q),
```

and now

```
((FORALL x | P) & Q) *imp (FORALL x | P & Q)
```

follows by generalization, since x does not occur freely in (FORALL x | P) & Q. Altogether this gives

```
((FORALL x | P) & Q) *eq (FORALL x | P & Q),
```

i.e. (b).

Statement (c) now follows via the chain of equivalences

```
(FORALL x | P *imp Q) *eq (FORALL x | (not P) or Q)       *eq ((FORALL x | (not P)) or Q)      *eq
```

Similarly statement (d) follows via the chain of equivalences

```
(FORALL x | Q *imp P) *eq (FORALL x | (not Q) or P)       *eq ((not Q) or (FORALL x | P))      *eq
```

The proofs of (e-h) are left to the reader.

The prenex normal form of predicate formulae

The prenex normal form of a predicate formula F is a logically equivalent formula in which quantifiers FORALL and EXISTS appear only at the very start of the formula. Rules (a-h) can now be used iteratively in the following way to put an arbitrary formula F into prenex normal form. We first change bound variables, using the equivalences derived above for this purpose, to ensure that all bound variables are distinct and that no bound variable is the same as any variable occurring freely. Then we use equivalences

```
(P *eq Q) *eq ((P *imp Q) & (Q *imp P)
```

to replace all '*eq' operators in our formula with combinations of implication and conjunction operators. After this, we search the syntax tree of the formula, looking for all quantifier nodes whose parent nodes are not already quantifier nodes, and moving them upward in a manner to be described. If there are no such nodes, then all the quantifiers occur in an unbroken sequence starting at the tree root, and so in the unparsed form of the formula they all occur at the left of the formula. The quantifier node moved at any moment should always be one that is as close as possible to the root of the syntax tree. Given that the parent of this quantifier is not itself a quantifier node, the parent must be marked with one of the Boolean operators &, or, *imp, not. If the operator at the parent node is 'not', we use one of the equivalences

```
(FORALL x1,...,xk | not P) *eq (not (EXISTS x1,...,xk | P))
```

and

```
(EXISTS x1,...,xk | not P) *eq (not (FORALL x1,...,xk | P))
```

to interchange the positions of the 'not' operator and the quantifier. In the remaining cases we use one of the equivalences (a-h) to achieve a like interchange. When this process, each of whose steps transforms

our original formula into an equivalent formula, can no longer continue, the formula that remains will clearly be in prenex normal form.

The deduction theorem

The Deduction Theorem of predicate calculus, which will be useful below, states that (provided that neither F or any of the statements in S contain any free variables) the implication F *imp G can be proved from a set S of predicate axioms if and only if G can be proved if F is added to the set S of axioms. Note that this is an easy consequence of the Gödel Completeness Theorem in the generalized form discussed at the start of this section. But in what follows we need to know that this result can be proved directly. This will now be shown.

Theorem. Let S be a collection of predicate formulae with no free variables and let S' be obtained from S by adding to it a predicate formula F with no free variables. Then

```
S |- F *imp G    if and only if    S' |- G,
```

for any predicate formula G.

Proof: Let S, S', F, and G be as above. First assume that S |- F *imp G holds and let

```
H1, H2, ..., Hn,
```

with H_n = F *imp G, be a proof of F *imp G from S. Then it follows immediately that

```
H1, H2, ..., Hn, F, G
```

is a proof of G from S'.

Conversely, assume that S' |- G and let

```
(*) H1, H2, ..., Hn,
```

with H_n = G, be a proof of G from S'. We can suppose without loss of generality that this proof does not use the strong variant of the rule of generalization stated earlier, but only the weaker form of this rule. Consider the sequence of predicate formulae

```
(**) F *imp H1, F *imp H2, ..., F *imp Hn.
```

We will show that by inserting suitable auxiliary formulae into this sequence we can turn it into a proof from S of F *imp G. Indeed, for each i = 1,2,...,n one of the following cases will apply:

(i) H_i may be a predicate axiom or H_i may be an element of S. In this case we insert the formulae

```
Hi  Hi *imp (F *imp Hi)
```

(of which the latter is a tautology) into (**) just before the formula F *imp H_i.

(ii) H_i may follow from H_j and H_k = H_j *imp H_i by modus ponens step. In this case we insert the formulae

```
(F *imp Hj) *imp ((F *imp (Hj *imp Hi)) *imp (F *imp Hi))  (F *imp (Hj *imp Hi)) *imp (F *imp Hi)
```

(of which the former is a tautology) into (**) just before the formula F *imp H_i.

(iii) In the remaining possible cases, namely if H_i is derived from some earlier statement of (*) by the rule of generalization, or if H_i = F, we need not add any formula to (**).

Let

```
K1, K2, ..., Km
```

be the sequence of predicate formulae generated in the manner just described. It is easy to check that this sequence constitutes a proof of K_m = F *imp G from S, provided that we now allow use of the strong variant of the rule of generalization. Since, as shown above, any such proof can be transformed into one in which all uses of the strong variant of the rule of generalization have been eliminated and only the weak form of this rule is used, it follows that S |- F *imp G, concluding our proof of the deduction theorem. **QED**

The deduction theorem admits the following semantic version, whose proof is left to the reader.

Theorem: Let S, S', F, and G be as in the statement of the deduction theorem. Then

```
S |= F *imp G    if and only if    S' |= G.
```

Definitions in predicate calculus; the notion of 'conservative extension'

Since the use of definitions to introduce new predicate and function symbols is fundamental to ordinary mathematical practice, it is important to understand the sense in which the predicate calculus accomodates this notion. The simplest definitions are algebraic, i.e. they simply introduce names for compound expressions written in terms of previously defined predicate and function symbols. Such definitions are unproblematical, since any use of them can be eliminated by expanding the new name back into the underlying expression which it abbreviates. But another, less trivial kind of definition is also essential. This is known as *definition by introduction of Skolem functions.* More specifically, once we have proved a formula of the form

```
(*)  (FORALL y1,...,yn | (EXISTS z | P(y1,...,yn,z)))
```

using the axioms of predicate calculus and some set S of additional axioms (none of which should have any free variables), we can introduce any desired new, never previously used function name f and add the statement

```
(**)  (FORALL y1,...,yn | P(y1,...,yn,f(y1,...,yn)))
```

to S. The point is that, although this added statement clearly allows us to prove new statements concerning the newly introduced symbol f, it does not make it possible to prove any statement *not involving* f that could not have been proved without its introduction.

This very important result can be called the *fundamental principle of definition.* To prove it we argue as follows. (But note that the following proof uses the Gödel Completeness Theorem, and so is entirely nonconstructive, i.e. it does not tell us how to produce the definition-free proof whose existence it asserts). Let P, S and f be as above, and let S' be obtained from S by adjoining the formula (**) to S. Let F be a formula not involving the symbol f, and suppose that S' |- F. Then we have S' |= F by the Gödel completeness theorem (as extended above). Our goal is to show that S |- F. By the Gödel completeness

theorem it is enough to show that S |= F. To this purpose, let (U,I,A) be an interpretation framework covering F and the statements in S and such that Val(I,A,G) = 1 for each G in S. Then we must show that Val(I,A,F) = 1.

Introduce an auxiliary Boolean function $p(u_1,...,u_n,u_{n+1})$, mapping the Cartesian product U^{n+1} of (n + 1) copies of U into {0,1}, by setting

```
p(u1,...,un,un+1) = Val(I,A(u1,...,un,un+1),'P(y1,...,yn,z)'),
```

where $A(u_1,...,u_n,u_{n+1})$ is the assignment which agrees with A everywhere except on the variables $y_1,...,y_n$ and z, for which variables we take

```
A(u1,...,un,un+1)(y1) = u1,      ...   A(u1,...,un,un+1)(yn) = un,   A(u1,...,un,un+1)(z) = un+1.
```

Since

```
S |- (FORALL y1,...,yn | (EXISTS z | P(y1,...,yn,z))),
```

we have

```
S |= (FORALL y1,...,yn | (EXISTS z | P(y1,...,yn,z)))
```

and therefore

```
1 = Val(I,A,(FORALL y1,...,yn | (EXISTS z | P(y1,...,yn,z))))    = Minu1,...,un(Maxun+1(Val(I,A(u1,...,u
```

where the minima and maxima over the subscripts seen extend over all values in U. Hence there exists a function h from U^n into U such that

```
p(u1,...,un,h(u1,...,un)) = 1
```

for all $u_1,...,u_n$ in U. Let I' be an interpretation which agrees with I everywhere except on the function symbol f and such that I'(f) is the function h just defined (which is, as required, a mapping from U^n to U). Hence

```
1 = Minu1,...,un(p(u1,...,un,h(u1,...,un)))    = Minu1,...,un(Val(I',A(u1,...,un),P(y1,...,yn,f(y1,...,yn))
```

where $A(u_1,...,u_n)$ is the assignment which agrees with A everywhere except on the variables $y_1,...,y_n$, for which variables we take

```
A(u1,...,un)(y1) = u1,      ...   A(u1,...,un)(yn) = un.
```

Since no formula G in S involves the function symbol f, we have

```
Val(I',A,G) = Val(I,A,G) = 1,
```

for all G in S. Therefore

```
Val(I',A,F) = 1,
```

since, as observed above, S' |= F. But since the formula F does not involve the function symbol f, we have

```
Val(I,A,F) = 1,
```

proving that S |= F, and so S |- F. This concludes our proof of the fundamental principle of definition.

The central notion implicit in the preceding argument is worth capturing formally.

Definition: Let S be a set of predicate formulae not involving any free variables, and let S' be a larger such set (possibly involving function and predicate symbols that do not occur in S). Then S' is called a *conservative extension* of S if

```
S' |- F implies  S |- F
```

for every formula F involving no predicate or function symbols not present in one of the formulae of S. The argument just given shows that the addition of formula (**) to any set S of formulae not containing free variables for which (*) can be proved yields a conservative extension.

Proof of the Gödel completeness theorem

Now we come to the proof of the Gödel completeness theorem. To prove it we first show, without using it, that the theorem holds for a certain very limited form of Skolem definition, namely if we introduce a single new constant symbol C (i.e. function symbol of 0 arguments) satisfying P(C), provided that we have previously proved a predicate formula of the form

```
(EXISTS z | P(z)).
```

These constants are traditionally called Henkin constants, after Leon Henkin, who introduced the technique that we will use. Our first key lemma is as follows.

Lemma 1: Let S be a collection of (syntactically well-formed) predicate formulae without free variables and let C be a constant symbol not appearing in any of the formulae of S. For each formula H, let H(C-->x) denote the result of replacing each occurrence of C in H by an occurrence of x, where x designates a variable not otherwise used. Then, if S |- H, we have

```
S |- H(C-->x).
```

In intuitive terms, this lemma tells us that if the axioms S can be used to prove some statement about a constant which they never mention, they can be used to prove the same statement in which C is replaced by a variable.

Proof of Lemma 1: Suppose that Lemma 1 fails for some H. Then, proceeding inductively, we can suppose that Lemma 1 holds for all statements having proofs shorter than that of H. Without loss of generality, we can assume that the variable x is not used in the proof of H. Consider the final step in the proof of H. This must either be (i) a citation of a predicate axiom; (ii) a citation of some statement in S; (iii) a modus ponens step involving two formulae G and G *imp H proved earlier; (iv) a generalization step from a formula G proved earlier. Concerning case (i), if H is a predicate axiom so is H(C-->x). In case (ii), namely if H is a member of S, H cannot involve the constant C, so that H(C-->x) = H and therefore we plainly have S |- H(C-->x).
Next consider case (iii). Since in this case G and G *imp H both have shorter proofs than that of H, it follows by inductive assumption that S |- G(C-->x) and S |- (G *imp H)(C-->x), i.e. S |- G(C-->x) *imp H(C-->x). Therefore it follows by a modus ponens step that S |- H(C-->x).
Finally we consider case (iv). In this case G has a shorter proof than that of its generalization H = (FORALL z | G). Hence by inductive assumption S |- G(C-->x), so that, by the rule of generalization, S |- (FORALL z |G(C-->x)) and therefore S |- H(C-->x), since

```
H(C-->x) = (FORALL z | G)(C-->x) = (FORALL z | G(C-->x)),
```

proving our claim in case (iv) and thus completing our proof of Lemma 1. **QED**.

Next we prove the following consequence of Lemma 1.

Lemma 2: Let S be a collection of (syntactically well-formed) predicate formulae without free variables. Let F be a predicate formula involving the one free variable y. Let C be a constant symbol not appearing in any of the formulae of S or in F, and let F(y-->C) denote the formula obtained from F by replacing each occurrence of y by an occurrence of C. Suppose that

```
S |- (EXISTS y | F).
```

Let S' be the union of S and the statement F(y-->C). Then S' is a conservative extension of S.

Proof: Let H be a formula involving only the symbols appearing in S, so that in particular the constant C does not occur in H. Suppose that S' |- H. By the Deduction Theorem we have

```
S |- F(y-->C) *imp H.
```

By Lemma 1 this last formula yields

```
S |- (F(y-->C) *imp H)(C-->x),
```

where x is a variable not otherwise used. Therefore

```
S |- F(y-->x) *imp H,
```

since F(y-->C)(C-->x) = F(y-->x) and H(C-->x) = H. Applying the rule of generalization we obtain

```
S |- (FORALL x | F(y-->x) *imp H).
```

We have shown above that

```
((FORALL x | F(y-->x) *imp H) &     (EXISTS x | F(y-->x))) *imp (EXISTS x | H)
```

and

```
(EXISTS y | F) *eq (EXISTS x | F(y-->x))
```

are universally valid. Thus, by propositional reasoning,

```
S |- (EXISTS x | H).
```

But since the variable x does not occur freely in H, we have

```
|- (FORALL x | (not H)) *eq (not H)
```

by predicate axiom (iv), and so it follows propositionally that

```
|- not (FORALL x | (not H)) *eq H.
```

Predicate axiom (iii) then gives

```
|- (EXISTS x | H) *eq H
```

and so S |- H, proving that S' is a conservative extension of S. **QED**

The remainder of the proof: predicate consistencey principle

We will now complete our proof of the Gödel completeness theorem. For this, it is convenient to restate it in the following way.

Predicate consistency principle. Let S be a set of formulae, none containing free variables, such that S is *consistent*, i.e. S |- false is false. Then there exists a *model* for S, i.e. an interpretation framework (U,I,A) covering all the predicate and function symbols appearing in S, such that Val(I,A,F) = 1 for each F in S. Conversely if there is a model for S then S is consistent.

This is simply the statement that S |- false is false iff S |= false is false. For S |= false is false means that there is an interpretation framework (U,I,A) covering all the statements F in S such that Val(I,A,F) = 1 for each F in S, but nonetheless satisfying the (required) condition that Val(I,A,false) = 0.

It is an easy matter to see that the predicate consistency principle implies that for every set S of predicate formulae with no free variables and for every predicate formula F the following condition holds:

```
(*)   if   S |= F   then   S|- F.
```

Indeed, assume that S |= F holds and that S |- F is false. Then S |- (FORALL v_1,...,v_n | F), where v_1,...,v_n are the free variables of F, must also be false, because otherwise by repeated use of axiom (v) and the rule of modus ponens S |- F would follow. Let S' be the set of predicate formulae obtained by adding the formula not (FORALL v_1,...,v_n | F) to S. Then S' |- false must be false, because otherwise by the deduction theorem S |- not (FORALL v_1,...,v_n | F) *imp false would hold and therefore, by propositional reasoning, S |- (FORALL v_1,...,v_n | F) would hold. Therefore the predicate consistency principle implies that S' has a model, namely there exists an interpretation framework (U,I,A) covering all the statements G of S' and such that Val(I,A,G) = 1 for all such G. Thus, in particular, we have that Val(I,A,C) = 1 for all the formulae C in S and Val(I,A, not (FORALL v_1,...,v_n | F)) = 1. This last statement implies that there exists an assignment A' such that Val(I,A', F) = 0. Since all formulae in S have no free variables, it follows that Val(I,A',C) = Val(I,A,C) = 1 for each formula C in S, thus contradicting our initial assumption that S |= F holds, and thereby proving statement (*).

But the statement (*) implies, and indeed is a bit more general than, the Gödel completeness theorem. This shows that the Gödel completeness theorem will follow if we can prove the predicate consistency principle.

To this end assume first that S is not consistent. Then S |- false holds. But then, as was shown earlier, S |= false follows, so that S cannot have any model.

For the converse, assume that S is consistent, in which case we must show that S has a model. We can and shall suppose that all our formulae are in prenex normal form, since we have seen that given any set of formulae there is an equivalent set of prenex normal formulae. We proceed in a kind of 'algorithmic' style, to generate a steadily increasing collection of formulae known to be consistent. At the end of this process it will be easy to construct a model of the set S of statements using these formulae and a bit of purely propositional reasoning. The idea of the proof is to introduce enough new constants C to ensure that, for each original existentially quantified formula

```
(EXISTS x | F),
```

there exists a C for which

```
F(x->C)
```

is known to be true. To this end, we maintain the following lists and sets of formulae, along with one set of auxiliary constants. These lists and sets can be (countably) infinite and will steadily grow larger. In order to be certain that there exist only finitely many constants with names below any given length, it will be convenient for us to suppose that all constants have names like 'C', 'CC', 'CCC',.... The lists and sets we maintain are then:

SC: the set of all constants introduced so far.

SUF: the set of all universally quantified formulae generated so far.

SNQ: the set of all formulae containing no quantifiers generated so far.

LEF: the list of all existentially quantified formulae generated so far. This list is always kept in order of increasing length of the formulae on it. Formulae of the same length are arranged in alphabetical order. Each formula on the list LEF is marked either as 'processed' or 'unprocessed'.

These data objects are initialized as follows. SC initially contains all the constants appearing in functions of S. SUF contains all the formulae of S which start with a universal quantifier. SNQ contains all the formulae of S which contain no quantifiers. LEF contains all the formulae of S which start with an existential quantifier. These are arranged in the order just described. All the formulae on LEF are originally marked 'unprocessed'.

The auxiliary set FS consists of all function symbols appearing in formulae of S.

The following processing steps are repeated as often as they apply, causing our four data objects to grow steadily. Note that SC is always finite, becoming infinite only in the limit, but that SUF, SNQ, and LEF can be infinite during the process that we now describe.

(a) Whenever new constants are added to SC or new universally quantified formulae to SUF, all the constants on SC are combined in all possible ways with function symbols of FS to create new terms, and these terms are substituted in all possible ways for initial universally quantified variables in formulae of SUF (all the variables up to the first existentially quantified variable, if any), thereby generating new formulae, some starting with existential quantifiers (these are added to LEF if not already there, following which LEF is rearranged into its required order), others with no quantifiers at all (these are added to SNQ if not already there).

(b) After each step (a), or if no step (a) is needed, we examine LEF to find the first formula (EXISTS x | F) on it not yet marked 'processed'. For this formula, we generate a new constant symbol C, build the formula F(x-->C) produced by replacing each free occurrence of x in F by C, and add this formula to SUF or LEF or SNQ, depending on whether it starts with a universal quantifier, starts with an existential quantifier, or has no quantifiers at all, and finally add the new constant C to SC. It is understood that the list LEF must always be maintained in lexicographic order. Finally, the formula (EXISTS x | F) on LEF is then marked 'processed'.

Processing begins as if the set of constants appearing in the formulae of S have just been added to SC, and so with step (a). (If the are no such constants, we must generate one initial constant symbol C to start processing).

At the end of this (perhaps infinitely long) sequence of processing steps, we may have generated a countably infinite list of constants as SC, and put infinitely many formulae into both of the sets SUF and SNQ and on the list LEF. But we can be sure that it is never possible to prove a contradiction from our set of formulae. For otherwise a contradiction would result from some finite set of formulae, all of which would have been added to our collection at some stage in the process we have described. But by assumption our formulae are consistent to begin with. Moreover no step of type (a) can spoil consistency, since only predicate consequences of previously added formulae are added during such steps. Nor can steps of type (b) spoil consistency, since it was proved above that steps of this kind yield conservative extensions of the set of formulae previously present.

It follows that at the end of the process we have described the set SNQ of unquantified formulae that results is consistent, i.e. that every finite subset of this set of formulae is consistent. We have proved above that this implies that SNQ has a propositional model, i.e. that we can assign a 0/1 value Va(T) to each atomic formula T appearing in any of the formulae F of SNQ, in such a way that each such F evaluates to 'true' if the atomic formulae appearing in it are replaced by these values, and the standard rules for calculating Boolean truth values of propositional combinations are then applied. Note for use below that each of the atomic formulae T of the set AT of all such formulae appearing in any F has the form $P(t_1,...,t_k)$, where P is a predicate symbol and $t_1,...,t_k$ are 'constant' terms (i.e. terms devoid of variables).

Now we show that there exists a model whose universe is the set CT of all constant terms generated by applying the function symbols in FS to the constants in SC in all possible ways. (The resulting set of terms is the so called *free universe* FU generated by these constants and the function symbols in FS). Each k-adic function symbol f in FS is trivially associated with a mapping I(f) from the Cartesian product FU^k of k copies of FU into FU, namely we can put

$$I(f)(t_1,...,t_k) = f(t_1,...,t_k)$$

for all lists $t_1,...,t_k$ of terms. For this I and every possible assignment A it is immediate that

$$Val(I,A,t) = t$$

for each term t in FU. A 0/1 valued function on FU^k can now be associated with each predicate symbol P appearing in a formula of S, namely we can write

$$I(P)(t_1,...,t_k) = Va(P(t_1,...,t_k))$$

for each atomic formula $P(t_1,...,t_k)$ appearing in one of the formulae of SNQ, and define $I(P)(t_1,...,t_k)$ arbitrarily for all other atomic formulae; here 'Va' is the Boolean assignment of truth values described in the preceding paragraph. It is then immediate that for every assignment A we have

$$Val(I,A,F) = 1,$$

for each formula of SNQ. It remains to be shown that we must have Val(I,A,F) = 1 for the quantified formulae of SUF and LEF also and for every assignment A. Suppose that this is not the case. Then there exists a formula F with n > 0 quantifiers for which Val(I,A,F) = 0. Proceeding inductively, we may suppose that n is the smallest number of quantifiers for which this is possible. If F belongs to LEF, then it

has the form (EXISTS x | G), and by construction we will have added a formula of the form G(x-->C), with some constant symbol C, to our collection. Since G(x-->C) has fewer quantifiers than n, we must have Val(I,A,G(x-->C)) = 1, and so Val(I,A,F), which is the maximum over a collection of values including Val(I,A,G(x-->C)), must be 1 also.

It only remains to consider the case in which F belongs to SUF, and so has the form

```
(FORALL x1,...,xm | G)
```

for some G. In this case, all formulae $G(x_1\text{-->}t_1,...,x_m\text{-->}t_m)$, where $t_1,...,t_m$ are any terms in our universe, namely the set TERM of all constant terms generated by applying the function symbols in FS to the constants in SC in all possible ways, will have been added to our collection. All these formulae have fewer quantifiers than n, and so we must have

```
Val(I,A,G(x1-->t1,...,xm-->tm) = 1
```

for all these terms. Hence the minimum of all these values, namely

```
Val(I,A,(FORALL x1,...,xm | G))
```

must also have the value 1. This completes our proof of the predicate consistency principle and in turn of the Gödel completeness theorem. **QED**

The argument just given clearly leads to the following slightly stronger result.

Corollary. Let S be a set of formulae in prenex normal form, and let SNQ be the set of all unquantified formulae generated by the process described above. Then S is consistent, i.e. it has a model, if and only if SNQ, regarded as a collection of propositions whose propositional symbols are the atomic formulae appearing in SNQ, is propositionally consistent.

Proof: As shown above, the set of statements in SNQ must be consistent if S is consistent. The argument given above establishes the converse, i.e. it shows that S has a model if SNQ is propositionally consistent. **QED**

Immediate consequences of the Gödel completeness theorem

The preceding corollary implies that in situations in which we can be sure that the procedure described in the proof of the predicate consistency principle will produce sets SC, SUF, SNQ and a list LEF all of which remain finite, this procedure can be used as an algorithm to decide in a finite number of steps whether or not a given finite set S of prenex normal formulae (none of which involves free variables) is consistent. One case in which this remark applies is that of pure 'EXISTS...EXISTS FORALL...FORALL' formulae, as defined by the following conditions:

i. S is a finite set of formulae in prenex normal form not involving free variables.
ii. No formula in S involves function symbols of arity greater than zero (i.e., the only terms allowed in these formulae are variables and constant terms). Of course, any number of predicate symbols can be used.
iii. No existential quantifier can follow a universal quantifier in any formula of S.

Note that the condition (iii) implies that the sequence of quantifiers prefixed to any 'EXISTS...EXISTS FORALL...FORALL' formula has the form

```
(EXISTS y1...ym | (FORALL x1...xn |  ...
```

To see why in this case the procedure described in the proof of the predicate consistency principle must converge after a finite number of steps, note first of all that since there are no function symbol the only terms substituted for universally quantified variables in step (a) of that procedure are constants. These constants must either be present in our initial formulae or be generated in some step of the procedure described. But since all existential quantifiers precede all universal quantifiers, the aforesaid step (a) will never generate any new formula containing existential quantifiers. Hence the number of constants generated is no greater then the number of existential quantifiers contained in our original collection of formulae, and substitution of these for all the universally quantified variables present will generate no more than a finite set of formulae.

Decidability for the Bernays-Schönfinkel sentences. An interesting special case of the foregoing is that when we are given a finite set S of pure 'EXISTS...EXISTS FORALL...FORALL' formulae, involving no free variables, as described above, and one additional formula F of the same kind and in which no universal quantifier follows an existential quantifier, and we want to determine whether S |- F holds. Let S' be the set of formulae obtained by adding the formula 'not F' to S. Then we know that S |- F holds if and only S' is inconsistent. But by moving the connective not in 'not F' across the quantifier prefix of F, we obtain another set S^* which is equivalent to S' and is still a finite set of pure 'EXISTS-FORALL' formulae, whose consistency can be tested algorithmically in the manner just explained.

The Löwenheim-Skolem Theorem. The argument given in the proof of the predicate consistency principle allows us to derive another interesting fact, known as the Löwenheim-Skolem Theorem. This states that any consistent countable set of sentences has a countable model. Indeed, if S is countable (as was implicitly assumed in our proof of the predicate consistency principle) then all the sets SC, SUF, SNQ, FS, and the list LEF maintained by the process described in the proof of the predicate consistency principle are countable at each stage, and so must also be countable in the limit. Therefore the model constructed from SNQ using the technique seen above must also be countable.

The compactness theorem. A set S of predicate formulae is said to be *satisfiable* if it has a model. The Compactness Theorem states that if S is a set of predicate sentences such that every finite subset of S is satisfiable, then the whole infinite set S is satisfiable. This theorem is an easy consequence of the predicate consistency principle. Indeed, let S be a set of predicate sentences such that every finite subset of S has a model, and assume that S is not satisfiable. Then S |= false holds, so that by the predicate consistency principle we have S |- false also, i.e. there exists a proof of 'false' from S. Since any proof from S can involve at most finitely many formulae of S, there must exist a finite subset S' of S such that S' |- false holds, and so by the predicate consistency principle S' |= false must hold. That is, S' is not satisfiable, contradicting our initial hypothesis that every finite subset of S is satisfiable.

Some other consequences of the Gödel completeness theorem

Skolem Normal Form. Let S be a countable (i.e. finite or denumerable) collection of syntactically well-formed predicate sentences. Putting each of these formulae into prenex normal form gives an equivalent set S' of formulae, so that if S has a model (i.e. it is consistent) so does S'. We will now describe a second normal form, called the *Skolem normal form*, into which the formulae of S' can be put. We will see that if S^{**} denotes the set of formulae in Skolem normal form derived from S', then S^{**} is consistent if and only if S' (and S) is consistent. However the formulae of S^{**} are generally not equivalent to the formulae of S' from which they derive. Thus S^{**} and S' (and S) are only *equiconsistent*, not *equivalent*.

By definition, a formula in prenex normal form is in Skolem normal form if and only if its prefixed list of quantifiers contains no existential quantifiers. To derive the Skolem normal form of a formula F in S', which must already be in prenex normal form, suppose that F has the form

```
(*)  (FORALL x1,...,xk | (EXISTS y | G)).
```

Introduce a new function symbol f of k variables, along with a statement of the form

```
(**)    (FORALL x1,...,xk | G(y-->e)),
```

where G(y-->e) is derived from G by replacing every free appearance of the variable y in G by an appearance of the subexpression e = $f(x_1,...,x_k)$. Let S_1 be the result of adding (**) to S'. We have seen above that S_1 is a conservative extension of S'. Hence if S' |- false is false, so is S'_1 |- false, and conversely. That is, S' and S_1 are equiconsistent.

Let S^* be the set of statements obtained by dropping (*) from S_1. We shall show that S' and S^* are equiconsistent. But in S^* the existentially quantified statement (*) has been replaced by (**) which has one fewer existential quantifier. It should be clear that by repeating this step as often as necessary, we can eliminate all existential quantiifiers from our original set of statements, introducing function symbols in their stead. The resulting set of statements is the Skolem normal form of our original set. To prove that S' and S^* are equiconsistent, note first of all that, as we have already noted, S^* is consistent if S' is consistent. Suppose conversely that S^* is consistent. We can deduce G(y-->e) from (**) by k successive applications of predicate axiom (v) and the rule of modus ponens. More specifically, we have

```
(FORALL x1,...,xk | G(y-->e)) |- G(y-->e).
```

But since

```
|- (FORALL y | not G) *imp (not G(y-->e))
```

by the same axiom (v), it follows that

```
(FORALL x1,...,xk | G(y-->e)) |-  not (FORALL y | not G).
```

Thus by predicate axiom (iii) we have

```
(FORALL x1,...,xk | G(y-->e)) |-  (EXISTS y | G)
```

and so, by repeated application of the rule of generalization, we obtain

```
(FORALL x1,...,xk | G(y-->e)) |- (FORALL x1,...,xk | (EXISTS y | G)).
```

The deduction theorem now implies

```
|- (FORALL x1,...,xk | G(y-->e)) *imp (FORALL x1,...,xk | (EXISTS y | G))
```

so that

```
S* |- (FORALL x1,...,xk | (EXISTS y | G)).
```

This implies that exactly the same formulae can be derived from S_1 and S^*, so that these two sets of formulae are equiconsistent. Hence S' and S^* are equiconsistent, as required.

The Herbrand theorem. Herbrand's theorem, which gives a *semi-decision procedure* for the satisfiability of sets of predicate formulae given in Skolem normal form, can be stated as follows.

Theorem (Herbrand): Let S be a countable collection of predicate sentences, all having Skolem normal form. Let D be the set of all function symbols appearing in the formulae of S. Let SC be the set of individual constants (function symbols of zero variables) appearing in the formulae of S. (If there are no such constants, let SC consist of just one artificially introduced individual constant, distinct from all the other symbols in D). Let T be the set of all terms which can be generated from the constants in SC using the function symbols appearing in formulae of S. Let S' be the set of formulae generated from S by stripping off their quantifiers and substituting terms in T for the variables of the resulting formulae in all possible ways. Then the set S is consistent if and only if every finite subset of S' is consistent when regarded as a collection of propositional formulae in which two atomic formulae correspond to the same propositional variable if and only if they are syntactically identical.

Proof: This is just the Corollary of the Gödel completeness theorem stated above, in the special case in which the formulae of S have Skolem normal form, i.e. they contain no existential quantifiers. For in this case the construction we have used to prove that Theorem and Corollary generates no new constant symbols. **QED.**

Herbrand's theorem is often used as a technique for searching automatically for predicate calculus proofs. If none of the formulae concerned have any free variables, we can show that a predicate formula F follows from a set S of such formulae by adjoining the negative of F to S, then putting all the resulting formulae into Skolem normal form, and finally searching for the propositional contradiction of whose existence Herbrand's theorem assures us.

As a very simple example, consider the predicate theorem

```
(+)  (EXISTS y | (FORALL x | P(x,y))) *imp (FORALL x | (EXISTS y | P(x,y)))
```

whose negation is

```
(++)  (EXISTS y | (FORALL x | P(x,y))) & (EXISTS x | (FORALL y | not P(x,y))),
```

or, in Skolem normal form,

```
  (FORALL x | P(x,B)) & (FORALL y | not P(A,y))).
```

A substitution then gives the propositional contradictions P(A,B) & (not P(A,B)), showing the impossibility of the negated statement (++), and so confirming the universal validity of (+).

A very large literature has developed concerning optimization of searches of this kind. Some of the resulting search techniques will be reviewed in Chapter 3.

Predicate calculus with equality as a built-in

The simplicity of the equality relationship and its continual occurrence in mathematical arguments make it appropriate to extend the predicate calculus as defined above to a slightly larger version in which equality is a built-in. Syntactically we have only to make '=' a reserved symbol; semantically we need to introduce axioms for equality strong enough for the Gödel completeness theorem to remain valid. The following axioms suffice.

The axioms of the *equality-extended predicate calculus* are all the axioms of the (ordinary) predicate calculus, plus

(vi) Any formula of the form

```
(FORALL x,y,z | x = x & ((x = y) *imp (y = x))     & ((x = y & y = z) *imp (x = z))).
```

(vii) Any formula of the form

```
(FORALL x,y | (x = y) *imp (f(xj-->x) = f(xj-->y))),
```

where f is a k-adic functional expression $f(x_1,...,x_k)$, and $f(x_j\text{-->}x)$ (resp. $f(x_j\text{-->}y)$) is the result of replacing the j-th variable in it by an occurrence of x (resp. y).

(viii) Any formula of the form

```
(FORALL x,y | (x = y) *imp (P(xj-->x) *eq P(xj-->y))),
```

where P is a k-adic predicate expression $P(x_1,...,x_k)$, and $P(x_j\text{-->}x)$ (resp. $P(x_j\text{-->}y)$) is the result of replacing the j-th variable in it by an occurrence of x (resp. y).

No new rules of inference are added.

The notion of 'model' is extended to this slightly enlarged version of the predicate calculus by agreeing that

(xi) If the formula F is of the form '$t_1 = t_2$', then

```
Val(I,A,F) = if Val(I,A,t1) = Val(I,A,t2) then 1 else 0 end if,
```

for every interpretation framework (U,I,A).

That is, the predicate which models the equality sign is simply the standard predicate of equality.

As before we want to show that the added predicate axioms evaluate to 1 in every model. This is clear for (vi), since it simply states the standard properties of equality. Similarly, since replacement of the arguments of any set-theoretic mapping by an equal argument never changes the map value, (vii) and (viii) must evaluate to 1 in any model.

Additionally we can show that the Gödel completeness theorem carries over to our extended predicate calculus. For this, we argue as follows. If (U,I,A) is an interpretation framework covering a set S of sentences in our extended calculus, then it follows as previously that if Val(I,A,F) = 1 for each F in S, then Val(I,A,G) = 1 for every G such that S |- G. Hence, as previously, if such a set S has a model it is consistent. Suppose conversely that S is consistent. Add the equality axioms (vi-viii) to S (this preserves consistency since only axioms are added to S) and proceed as above to build the sets SC, SUF, SNQ and the list LEF. Then the collection of statements in SNQ must be propositionally consistent, and so must have a propositional model V for which every statement in SNQ takes on the value 'true'. It was seen above that this gives a model (U,I,A) of all the statements in our collection, with universe U equal to the set of all terms formed from the constants in SC using the function symbols appearing in formulae of S. This is not quite a model of S in the sense required when we take '=' as a built-in predicate symbol which must be modeled by the standard equality operator, since there may well exist formulae of the form $t_1 = t_2$ such that $Val(I,A,t_1 = t_2) = 1$ even though t_1 and t_2 are syntactically distinct. However, the binary relationship

```
(+)  R(t1,t2) = (Val(I,A,t1 = t2) = 1)
```

between terms of U must be an equivalence relation, since whenever terms t_1, t_2 and t_3 are generated we will have added all the assertions

```
t1 = t1 & ((t1 = t2) *imp (t2 = t1)) &      ((t1 = t2 & t2 = t3) *imp (t1 = t3))
```

to our collection. Moreover, since in the same situation statements like

```
(t1 = t2 *imp (f(..t1..) = f(..t2..))      and  (t1 = t2 *imp (P(..t1..) *eq P(..t2..)).
```

will have been added to our collection for all function and predicate symbols, the terms must always be equivalent whenever their lead function symbols are the same and their arguments are equivalent, and also we must have Val(I,A,P(..t_1..)) = Val(I,A,PP(..t_2..)) for atomic formulae when their lead function symbols are the same and their arguments are equivalent. Therefore we can form a model of our set of statements by replacing the universe U by the set U' of equivalence classes on it defined by the equivalence relation (+), and in this new model the symbol '=' is represented by the standard equality operation. This concludes our proof that the Gödel completeness theorem carries over to our extended predicate calculus. **QED.**

2.3. Set theory as an axiomatic extension of predicate calculus

In most of the present book we take a rather free version of set theory (perhaps this should be called 'brutal' set theory) as basic, and use it to hurry onward to our main goal of proving the long list of theorems found in Chapter 5. The standard treatment of set theory ties it more carefully to predicate calculus. Specifically, to ensure applicability of the foundational results presented earlier in this chapter, set theory is cast as a collection of predicate axioms. In this form it is customarily referred to as Zermelo-Fraenkel set theory (ZF) if no version of the axiom of choice is necessarily included, or ZFC if an axiom of choice is present. Here is the standard list of ZFC axioms.

Zermelo-Fraenkel theory with the axiom of choice

1. **(Axiom of extension)**

```
(FORALL s,t | s = t *eq (FORALL x | (x in s) *eq (x in t))).
```

2. **(Axioms of elementary sets)** There is an empty set 0; for each set t there is a set Singleton(t) whose only member is t; if s and t are sets then there is a set Unordered_pair(s,t) whose only members are s and t. That is, we have

```
(FORALL s | not (s in 0)),  (FORALL s,t | (s in Singleton(t)) *eq (s = t)),  (FORALL s,t,u |
```

3. **(Axiom of power set)** To every set A there corresponds a set pow(A) whose members are precisely the subsets of A:

```
(FORALL s,t | (s in pow(t)) *eq       (FORALL x | (x in s) *eq           (FORALL y | (y in x) *imp
```

4. **(Axiom of union)** To every set A there corresponds a set Un(A) whose members are precisely those elements belonging to elements of A:

```
(FORALL s,t | (s in Un(t)) *eq (EXISTS x | (x in y) & (s in x))).
```

5. **(Axiom of infinity)** There is at least one set Inf such that

```
0 in Inf & (FORALL s | (s in Inf) *imp (Singleton(s) in Inf).
```

6. **(Axiom of regularity)**

```
not (EXISTS x | (x /= 0) &           (FORALL y | (y in x) *imp (EXISTS z | (z in x) & (z in y)))).
```

7. **(Axiom schema of subsets)** If $F(y,z_1,...,z_n)$ is any syntactically valid formula of the language of ZF that has no free variables other than those shown, and neither x nor z occur in the list $y,z_1,...,z_n$, then

```
(EXISTS z | (FORALL y | (y in z *eq y in x & F(y,z1,...,zn))))
```

is an axiom. Here and below, a formula is said to be a formula of the language of ZF if it is formed using only the built-in symbols of predicate calculus (i.e. the propositional operators, FORALL, EXISTS, =) plus the membership operator. (Note that in stating this axiom, we mean to assert the formula which results by quantifying it universally over all the free variables $z_1,...,z_n$).

8. **(Axiom schema of replacement)** If $F(u,v,z_1,...,z_n)$ is any syntactically valid formula of the language of ZF that has no free variables other than those shown, and neither u nor v occur in the list $z_1,...,z_n$, then

```
(FORALL u,v1,v2 |       ((F(u,v1,z1,...,zn) & F(u,v2,z1,...,zn)) *imp v1 = v2) *imp           (FORALL
```

is an axiom. (Here again, in stating this axiom, we mean to assert the formula which results by quantifying it universally over all the free variables $z_1,...,z_n$).

This statement is obscure enough for a brief clarifying discussion of its equivalent in our informal version of set theory to be helpful. In that less formal system we would proceed by defining an auxiliary 'Skolem' function h satisfying

```
(FORALL x,z1,...,zn | (EXISTS y | F(x,y,z1,...,zn) *eq F(x,h(x,z1,...,zn),z1,...,zn))).
```

Then, since the replacement axiom assumes that $F(x,y,z_1,...,z_n)$ defines y uniquely in terms of x and $z_1,...,z_n$, we have

```
(FORALL x,y,z1,...,zn | F(x,y,z1,...,zn) *eq y = h(x,z1,...,zn)),
```

and so the set c whose existence is asserted by the axiom of replacement can be written in our 'working' version of set theory as

```
{h(x,z1,...,zn): x in b}.
```

This 'setformer' expression is the form in which such constructs will almost always be written.

9. **(Axiom of choice)**

```
(FORALL x |  (EXISTS f | (is_function(f) & domain (f) = x &        (FORALL y | (y in x & y /= 0) *
```

Note that this form of the axiom of choice is weaker than the assumption concerning 'arb' which our 'brutal' set theory uses in its place. Specifically, while 'arb' is a universal choice function applicable to any non-null set, the axiom of choice just stated provides a separate such choice function for each set of sets.

Most axioms appear in Skolemized version in the above list. Other authors prefer to write those in unskolemized form, e.g. to write our axiom `(FORALL s | not (s in 0))` in the form

```
(EXISTS z | (FORALL s | not(s in z)).
```

Similarly the axiom of union will often be written as

```
(FORALL s | (EXISTS u | (FORALL s | (s in u) *eq      (EXISTS x | (x in y) & (s in x))))).
```

The main respects in which the ZFC formulation of set theory differs from our 'brutal' version is that no built-in setformer construct is provided, nor are 'transfinite recursive' definitions like those freely allowed in our version of set theory. An issue of relative consistency therefore arises: can our version of set theory be reduced to ZFC in some standard way, or, if ZFC is assumed to be consistent, can it be demonstrated that our 'brutal' version is consistent also?

Concerning the consistency of ZFC and various interesting extensions of it

To open a discussion of this problem we first consider the general question of consistency for set-theoretic axioms like the ZFC axioms. Since equality can be treated as an operator of logic, these axioms involve only one non-logical symbol, the predicate symbol 'in'. The Gödel completeness theorem tells us that the ZFC axioms are consistent if and only if they have a model. How can such models be found? Are there many of them having an interesting variety of properties, or just a few? Since von Neumann's 1928 paper on the axioms of set theory and Gödel's 1938 work on the continuum hypothesis, many profound studies have addressed these questions. We can get some initial idea of the issues involved by looking a bit more closely at the hereditarily finite sets. We will see that these are of interest in the present context since they model all the axioms of set theory other than the axiom of infinity.

Basic facts concerning hereditarily finite sets

In intuitive terms, the 'hereditarily finite' sets s are those which can be constructed by using the pair formation operation {x,y} and union operation x + y repeatedly, starting from the null set {}. Any such set has a string representation r consisting of a properly matched arrangement of opening brackets '{' and closing brackets '}', 'properly matched' in the sense that there are equally many opening and closing brackets, and that no initial substring of r contains more closing than opening brackets. Moreover, the string representation r of any such set is indecomposable, in the sense that no initial substring of r is properly matched. Examples are

```
{}      {{}}     {{}{{}}}.
```

The 'height' of any such set is one less than the maximum depth of bracket nesting in its string representation. For example, the three sets just displayed have heights 0, 1, and 2 respectively. The general transfinite induction techniques described in the preceding section make it possible to prove that the hereditarily finite sets are precisely those sets which are finite and all of whose elements are themselves hereditarily finite; this point is discussed in greater detail in Chapters 3 and 4.

Hereditarily finite sets can be represented in many ways by computer data structures which allow the basic operations on them, namely {x,y}, x + y, and x in y, to be realized by simple code fragments, and therefore allow translation of setformer expressions and recursive function definitions of all kinds into computer programs. One way of doing this is to make direct use of string representations like those just displayed. To this end, note that each properly matched arrangement of brackets is a concatenation of one or more indecomposable properly matched arrangements of brackets, and that every indecomposable arrangement has the form {s} where s itself is properly matched. Moreover the decomposition of any properly matched arrangement of brackets into indecomposable properly matched substrings is unique. (The reader is invited to prove these elementary facts, and to describe an algorithm for separating any

properly matched arrangement of brackets into its indecomposable parts).

It follows from the facts just stated that each hereditarily finite set t has a string representation, itself indecomposable, of the form

```
(1)     {s1s2...sm},
```

where each of the s_j is properly matched and indecomposable, and where all these s_j, which are simply the string representations of the elements of t, are distinct. We can make this string representation unique by insisting that the s_j be arranged in order of increasing length, members having string representations of the same length then being arranged in alphabetical order of their representations. We can call a string representation (1) having these properties at every recursive level (and in which all the s_j are distinct at every level) a 'nicely arranged' properly matched arrangement of brackets.Then every hereditarily finite set has a unique string representation of this kind, and conversely every nicely arranged properly matched arrangement of brackets represents a unique set. Hence these arrangements give an explicit, 1-1 representation of the family of all hereditarily finite sets.

In this representation, the two elementary operations {x,y} and x + y which suffice for construction of all such sets have the following simple implementations. The representation of {x,y} is obtained by taking the representations s_x and s_y of x and y respectively, checking them for equality and eliminating one of them if they are equal, arranging them in order of length (or alphabetically if their lengths are equal), and forming the string $\{s_x s_y\}$ (or simply $\{s_x\}$ if s_x and s_y are identical). To compute the standard string representation of x + y, let $\{s_1 s_2 ... s_m\}$ and $\{t_1 t_2 ... t_n\}$ be the standard string representations of x and y respectively. Then form the concatenation

```
s1s2...smt1t2...tn,
```

rearrange its indecomposable parts in the standard order described above, eliminate duplicates, and enclose the result in an outermost final pair of brackets.

In this, or any other convenient representation, it is easy to construct a code fragment which will calculate the value of any setformer of the type we allow, for example

```
{e(x): x in s | P(x)},
```

provided that s is hereditarily finite, and that e is any set-valued expression and P(x) any predicate expression which can be calculated by procedures which have already been constructed. For this, we have only to set up an iterative loop over all the elements of s, and use an operation which calculates e(x) for each element x of s satisfying P(x) and then inserts all such elements into an initially empty set, eliminating duplicates.

The powerset operation pow(s) (set of all subsets of s) satisfies the recursive relationship

```
pow(s) = if s = {} then {{}} else pow(s - {arb(s)})          + {x + {arb(s)}: x in pow(s - {arb(s
```

which can be used to calculate pow(s) recursively for each hereditarily finite s. This makes it possible to calculate setformers of the second allowed form

```
{e(x): x *incin s | P(x)},
```

by translating them into

```
{e(x): x in pow(s) | P(x)}.
```

Setformers involving multiple bound variables, for example

```
{e(x,y,z): x in s, y in a(x), z in b(x,y) | P(x,y,z)},
```

can be calculated in much the same way using multiply nested loops, provided that all the sets which appear are hereditarily finite and that e, a, and b are set-valued expressions, and P(x,y,z) a predicate expression, which can be calculated by procedures which have already been constructed. Similar loops can be used to calculate existentially and universally quantified expressions like

```
(FORALL x in s, y in a(x), z in b(x,y) | P(x,y,z))
```

and

```
(EXISTS x in s, y in a(x), z in b(x,y) | P(x,y,z)),
```

or such simpler quantifiers as

```
(FORALL x in s | P(x))    and    (EXISTS x in s | P(x)).
```

Note however that the predicate calculus in which we work also allows quantifiers involving bound variables not subject to any explicit limitation, for example

```
(FORALL x | P(x))  and  (EXISTS x | P(x)).
```

Since translation of expressions of this form into a programmed loop would require iteration over the infinite collection of all hereditarily finite sets, we can no longer claim that the values of these unrestricted iterators are effectively calculable. Thus they represent a first step into the more abstract world of the actually infinite, where symbolic reasoning must replace explicit calculation.

All the kinds of definition we allow translate just as readily into computer codes as long as only hereditarily finite sets are considered. Algebraic definitions like

```
Un(x) := {z: y in x & z in y}
```

translate directly into procedures whose body consists of a single nested iteration. Recursive definitions like

```
enum(X,S) := if S *incin {enum(y,S): y in X} then S         else arb(S - {enum(y,S): y in X})
```

translate just as directly into recursive procedures. Thus, as long as we confine ourselves to hereditarily finite sets, the whole of the set theory in which we work (excepting only unrestricted quantifiers of the kind shown above) can be thought of both as a language for the description of mathematical relationships and as an implementable (indeed, implemented) programming language for actual manipulation of a convenient class of finite objects. This parallelism between language of deduction and language of computation will be explored more deeply in Chapter 4.

We can summarize the preceding discussion in the following way. All hereditarily finite sets can be given explicit finite representations, so that these sets constitute a 'universe of computation' in which all of the properties we assume for sets can be checked by explicit computation, at least in individual cases. We will see below that the collection of hereditarily finite sets models all the axioms of set theory, save one:

there is no infinite set, for example no hereditarily finite set t having the property

```
t /= {} & (FORALL x in t | {x} in t)
```

which we will use as our axiom of infinity. By including this statement in our collection of axioms we cross from the world of computation defined by the hereditarily finite sets into a more abstract world of objects which can no longer be enumerated explicitly but which are known only through the statements about them that we can deduce formally, i.e. as elements of a world of formal computation, whose main elementary property is simply its formal consistency. Nevertheless, mathematical experience has shown that the statements that we can prove about the objects of this abstract world are both beautiful and extremely useful tools for deriving many properties of hereditarily finite sets which it would be harder or impossible to prove if we refused to enlarge our universe of discourse to allow free reference to infinite sets.

Hereditarily finite sets: formal definition within general set theory

Hereditarily finite sets can be defined formally in either of two ways: either as all sets satisfying a predicate is_HF, or as all the members of a set HF. The predicate is_HF is defined in the following recursive way (we continue to designate the set of all integers by Z):

```
is_HF(x) :*eq (#x in Z & (FORALL y in x | is_HF(y))).
```

To define the corresponding set HF (thereby showing that the collection of all x satisfying is_HF(x) is really a set), a bit more work is needed. We proceed as follows. Begin with the following recursive definition (informally speaking, this defines the collection of all sets of 'rank x').

```
HF_(x) :=     if x = {} then {} else Un({pow(HF_(y)): y in x}) end if.
```

It is easily proved by recursion that

```
(FORALL y in HF_(x) | HF_(x) incs y).
```

Indeed, if there exists an x for which 'HF_(x) incs z' is false for some z in HF_(x), there exists a smallest such x, which, after renaming, we can take to be x itself. Then there is a u such that 'z in HF_(x)', 'u in z', 'u notin HF_(x)'. Since 'z in HF_(x)', we have

```
 z in Un({pow(HF_(y)): y in x}),
```

so z in pow(HF_(y)) for some y in x, i.e. z *incin HF_(y) for some y in x. Then u in HF_(y) for some y in x. Since x has no member y for which

```
(FORALL w in HF_(y) | HF_(y) incs w)
```

is false, it follows that HF_(y) incs u, so u in pow(HF_(y)), and therefore

```
u in Un({pow(HF_(y)): y in x}),
```

i.e. u in HF_(x), proving our claim. Note also that the function HF_ is increasing in its parameter, in the sense that if y in x, then HF_(x) incs HF_(y). Indeed if u is an element of HF_(y), then {u} in pow(HF_(y)), so

```
{u} in Un({pow(HF_(y)): y in x}),
```

and therefore {u} in HF_(x), so by what we have just proved u in HF_(x).

In what follows we also need the fact that

```
(FORALL n in Z | #HF_(n) in Z),
```

i.e. that all the sets HF_(n) are themselves finite. To prove this, suppose that it fails for some smallest n. Then

```
HF_(n) = Un({pow(HF_(m)): m in n}),
```

all the sets HF_(m) for which m in n are finite, and so are their power sets. Thus HF_(n) is the union of a sequence of sets, each of finite cardinality, over a domain of cardinality less than Z (i.e. of finite cardinality). Hence HF_(n) is itself finite, i.e. #HF_(n) belongs to Z, as asserted.

Now we can define the set HF by

```
(+)  HF := Un({HF_(n): n in Z}).
```

To come to the desired goal we must prove that

```
(FORALL y | is_HF(y) *eq y in HF).
```

This can be done as follows. Suppose that y in HF. Then we have y in HF_(n) for some n in Z. To prove that is_HF(y), suppose that this is false, and, proceeding inductively, that n is the smallest element of Z for which HF_(n) has an element y such that is_HF(y) is false. Then, since

```
y in Un({pow(HF_(m)): m in n}),
```

we have y in pow(HF_(m)) for some m in n. All the elements u of y are therefore elements of HF_(m), and so satisfy is_HF(u). We have also proved that HF_(m) is finite, so all its subsets are finite, and therefore #y in Z, proving that is_HF(y), a contradiction implying that

```
(y in HF) *imp is_HF(y)
```

for all y.

Suppose conversely that is_HF(x), and that x notin HF. Proceeding inductively, we can suppose that x is a minimal element with these properties, i.e. that y in HF for each y in x. Then it follows from (+) that for each y in x there is an n = n(y) in Z for which y in HF_(n(y)). But then since x is finite by definition of is_HF(x), the maximum m of all these n(y) is finite, so every y in x belongs to HF_(m) since the sets HF_(m) clearly increase with their parameter m. Therefore x in pow(HF_(m)), x in HF_(m+1), and x in HF, a contradiction implying that

```
is_HF(y) *imp (y in HF)
```

for all y, which leads to the desired conclusion.

It is easily seen that HF is a model of all the ZFC axioms *other than the axiom of infinity*. To show this, we simply need to check that all these axioms remain valid if we interpret all quantifiers as extending over the set HF rather than over the 'universe of all sets' that the initial ZFC axioms assume. This can be done as follows. (1) The axiom of extension remains true since HF is *transitive,* i.e. every member of a

member of HF belongs to HF. (2) The null set, singleton, and unordered pair constructions take elements of HF into themselves since they construct finite sets all of whose elements are drawn from HF. (3) The power set axiom remains valid since every subset of an hereditarily finite set is hereditarily finite, and for s in HF, pow(s) consists only of such elements and also is finite. (4) The union set axiom remains valid since every member of a member of Un(s), where s is an hereditarily finite set, is hereditarily finite, and for s in HF, Un(s) is the union of finitely many sets and so is finite. (5) The axiom of infinity fails. (6) The axiom of regularity clearly remains true, since each z in HF has the same members as an element of HF that it does as a set. (7) The axiom schema of subsets, which in informal terms asserts the existence of the set y = {u: u in x | F(x,z_1,...,z_n)} for every x and z_1,...,z_n, remains true since the y whose existence it asserts is a subset of the x which it assumes, and so must be hereditarily finite if x is hereditarily finite. (8) In informal terms, the axiom schema of replacement asserts the existence of the set y = {u: x in b | F(x,u,z_1,...,z_n)} for every b and z_1,...,z_n if the predicate F defines u uniquely in terms of x and z_1,...,z_n. This remains true if only hereditarily finite sets are allowed, since if b is finite and each u is required to be hereditarily finite the set of whose existence it asserts is a finite set of elements, each of which is hereditarily finite, and so must be hereditarily finite. (9) The axiom of choice remains true since the f whose existence it asserts is a single-valued map whose pairs have their first components in x and their second components in Un(x): assuming that x in HF, each such pair plainly belongs to HF and therefore, since f consists of finitely many such pairs, we conclude that f in HF. (If 0 in x, we can carry out a similar argument, after replacing the image f(0) by 0).

Large Cardinal axioms

The preceding observations concerning the set HF suggest that it may be possible to find a model of set theory, which would imply the consistency of set theory, by replacing Z, the smallest infinite cardinal, by something larger in the crucial formula (+) seen above. If this is done, the argument that we have given can be shown to go through almost without change for any cardinal having the two properties of Z used in the argument. The following definition gives names to these properties:

Definition: A non-null cardinal number N is *inaccessible* if (a) any set of cardinals, all less than N, which has a cardinality smaller than N also has a supremum less than N. (Cardinals having this property are called *regular* cardinals). (b) If M is a cardinal less than N then 2^M (which is #pow(M) by definition) is less than N. (Cardinals which have this property are called *strong limit* cardinals).

Note that the set Z of integers is inaccessible according to this definition. Intuitively speaking, a cardinal number N is inaccessible if it cannot be constructed from smaller cardinals using any 'explicit' set-theoretic operation, so that the very existence of N would seem to involve some new assumption, in the same way that assuming the existence of an infinite set takes a step beyond anything that follows from the properties of hereditarily finite sets x in HF.

If we make the following quite straightforward definition, which simply generalizes the preceding construction of HF to arbitrary cardinal numbers N,

Definition:

```
H(N) := Un({HF_(n): n in N}) for every cardinal number N.
```

then the preceding discussion shows that

Theorem: If N is an inaccessible cardinal larger than Z, then H(N) is a model of the ZFC axioms of set theory.

Corollary: It there exists any inaccessible cardinal larger than Z, then the ZFC axioms have a model, and so are consistent.

A theorem of Gödel to be proved in Chapter 4 shows that no system having at least the expressive power and proof capability of HF can be used to prove its own consistency. Thus the corollary just stated implies the following additional result:

Corollary: Adding the assumption that there exists an inaccessible cardinal larger than Z to the ZFC axioms allows us to construct a model of the ZFC axioms and hence implies that these axioms are consistent. Therefore the ZFC axioms cannot suffice to prove that there exists an inaccessible cardinal larger than Z.

The situation described by this last corollary is much like that seen in the case of HF. The ZFC axioms, which include the axioms of infinity, allow us to define the infinite cardinal number Z and so the model HF of the theory of hereditarily finite sets. The theory of hereditarily finite sets can be formalized by dropping the axiom of infinity (keeping the other axioms of ZFC, and adding a suitable principle of induction); but the resulting set of 'HF axioms' do not suffice to prove the existence of even one infinite set.

The technique for forming models of set theory seen in the preceding discussion, namely identification of some transitive set H in which the ZFC axioms remain true if we redefine all quantifiers to extend over the set H only, does not change the definition of ordinal numbers, since an element t of s is an ordinal (in the overall ZFC theory) iff its members are totally ordered by membership and each member of a member of t is a member of t. Since the collection of members of t remains the same in H, this definition is plainly invariant. Thus the ordinal numbers of the model H, seen from the vantage point of the overall ZFC universe, are just those ordinals which are members of H. But the situation is different for cardinal numbers, which are defined as those ordinals O which cannot be mapped to smaller ordinals by a 1-1 mapping, i.e. those which do not satisfy

```
not_cardinal(O) *eq     (EXISTS f | one_1_map(f) & domain(f) = O & range(f) in O).
```

When we cut the whole ZFC universe of sets down to the set H, the set of ordinals will grow smaller, but so will the set of 1-1 mappings ('one_1_maps') f appearing in the formula seen above, making it unclear how the collection of cardinals (relative to H), or the structure of this set, will change. The power set operation can also change, since for s in H the power set relative to H is the set pow(s) * H of the ZFC universe. Thus properties and statements involving the power set can change meaning also. But the union set Un(s) retains its meaning. (Note also that if f is a member of H, then the property one_1_map(f) holds relative to H if and only if it holds in the ZFC universe, since it is defined by a formula quantified over the members of f, and these are the same in both contexts).

However, in the particularly simple case in which we restrict our universe of sets to H(N) where N is an inaccessible cardinal, the property 'not_cardinal' does not change. This is because any one_1_map in the ZFC universe for which domain(f) in H(N) & range(f) in H(N) must itself belong to H(N), since it is a set of ordered pairs of elements all belonging to H(N), whose cardinality is at most that of domain(f), and so is less than N. It readily follows that the cardinals of H(N) are simply those cardinals of the ZFC universe which lie below N; likewise for the regular, strong limit, and inaccessible cardinals.

It follows that ZFC, plus the assumption that there are two inaccessible cardinals, allows us to construct a set H(N) in which there is one inaccessible cardinal (namely we take N to be the second inaccessible cardinal), and so implies the consistency of ZFC plus the axiom that there is at least one inaccessible cardinal. Generally speaking, axioms which imply the existence of many and large inaccessible cardinals

imply the consistency of ZFC as extended by statements only implying the existence of fewer and smaller inaccessible cardinals, but not conversely. Thus the addition of stronger and stronger axioms concerning the existence of large cardinal numbers exemplifies a basic consequence of the incompleteness theorems presented in Chapter 4, namely that no fixed set of axioms can exhaust all of mathematics, so that significant extension of consistent systems by the addition of new axioms will always remain possible. The fact that large cardinal axioms can be formulated independently of any detailed reference to the syntax of the language of set theory makes them interesting in this regard, and so has encouraged the study of axioms which imply the existence of more and more, larger and larger, cardinal numbers.

It is worth reviewing a few of the key definitions that have appeared in such studies:

> **Definition:** Let S be a set of cardinal numbers all of whose members are less than a fixed cardinal number N.
>
> (i) S is said to be *closed relative to N* if the union of every sequence of elements of S whose length is less than N is a member of S.
>
> (ii) S is said to be *unbounded in N* if every cardinal less than N is also less than some member of S.
>
> (iii) S is said to be *thin in N* if there exists a closed unbounded set relative to N which does not intersect S.

Definition: A nonempty set F of non-empty subsets of a set S is called a *filter* on S if the intersection of any two elements of F is an element of F and any superset, included in S, of an element of F is an element of S. A filter F is an *ultrafilter* if whenever the union of finitely many subsets of S belongs to F, one of these subsets belongs to F. Given a cardinal number N, a filter F is said to be *N-complete* if whenever the union of fewer than N subsets of S belongs to F, one of these subsets belongs to F. An ultrafilter F is said to be *nontrivial* if it is not the collection of all sets having a given point p as member.

Note that if F is an N-complete filter on S, the intersection IT of any collection T of sets in F such that #T is less than N belongs to F. Indeed, S belongs to F, and if G belongs to F then S - G is not in F, since otherwise F would contain the null set G * (S - G). But now S is the union of IT and the collection of all complements S - G for G in T, and since #T is less than N and F is N-complete, the union of all these complements must lie outside T, so IT must belong to F.

The following definition lists two of the various kinds of large cardinal numbers that have been considered in the literature.

Definition: (i) A cardinal number N is a *Mahlo* cardinal if it is inaccessible and the set of regular cardinals less than N is not thin.

(ii) A cardinal number N is *measurable* if there is a nontrivial N-complete ultrafilter for N.

Note that if there is a Mahlo cardinal N, then the number of inaccessible cardinals below N must be at least N. For if there were fewer, then since N is inaccessible the supremum M of all these cardinals would also be less than N. But then the set SLC of all strong limit cardinals between M and N is unbounded and closed, contradicting the assumption that N is Mahlo. Indeed, for each K between M and N, the supremum of the sequence 2^K,2^{2^K},... must be a strong limit cardinal, showing that SLC is unbounded in N. Also the supremum L of any collection of strong limit cardinals must itself be a strong limit cardinal, since any L_1 less than L must plainly be less than some cardinal of the form 2^K. This shows that SLC is

closed. Now, no member K of SLC can be regular, since if it were it would be inaccessible, contradicting the fact that M is the largest inaccessible below N. This shows that the set of regular cardinals below N is thin, contradicting the assumption that N is Mahlo, and so completes our proof of the fact that every Mahlo cardinal N must be the N-th inaccessible.

It follows that the assumption that there is a Mahlo cardinal is much stronger than the assumption that there is an inaccessible cardinal, since it implies that there are inaccessibly many inaccessible cardinals.

Suppose next that the cardinal number N is measurable, and let F be an N-complete nontrivial ultrafilter on N. Then any set consisting of just one point p must lie outside F (or else F would be the trivial ultrafilter consisting of all sets having p as member). Since F is N-complete, it follows that every subset of N having fewer than N points lies outside F, and therefore so does every union of fewer than N such sets. Hence every measurable cardinal is regular. We will now show that if K is a cardinal less than N, then 2^K is less than N also, showing that every measurable cardinal is inaccessible. Suppose the contrary, so that there exists a collection CF of monadic-valued functions f(j) defined for all j in K, but having cardinality N, and so standing in 1-1 correspondence with N. This correspondence maps F to an N-complete nontrivial ultrafilter F' on CF. For each j in K, let a(j) be that one of the two Boolean values {0,1} for which the set of functions {f in S | f(j) = a(j)} belongs to F'. Then, since F' is N-complete, it follows, as was shown above, that the intersection of all the sets {f in S | f(j) = a(j)} must belong to F', and so F' contains a singleton and must therefore be trivial, contrary to assumption.

This proves that any measurable cardinal N is inaccessible. Jech proves the much stronger result (Lemma 28.7 and Corollary, p. 313) that N must be Mahlo, and in fact must be the N-th Mahlo cardinal. He goes on to define yet a third class of cardinals, the *supercompact* cardinals (p. 408), and to show that each supercompact cardinal N must be measurable, and in fact must be the N-th measurable cardinal (Lemma 33.10 and Corollary, p. 410). [As a general reference for this area of set theory, see Thomas Jech, Set Theory, 2nd edn., Springer Verlag, 1997.]

In light of the preceding, we can say that various axioms implying the existence of very many large inaccessible cardinals have been considered in the literature, with some hope that they can be used to define consistent extensions of the axioms of set theory.

The preceding discussion suggests the following transfinite recursive definition, which generalizes some of the properties of very large cardinals considered above:

```
(+)  Px(N) :*eq iff x = {} then Is_inaccessible(N) else    (FORALL y in x | #{M: M in N | Py(M)} = N)
```

Thus $P_0(N)$ is true iff N is inaccessible, $P_1(N)$ is true iff N is the N-th inaccessible (which we have seen to be true for Mahlo cardinals), $P_1(N)$ is true iff N is the N-th cardinal having property P_1 (which we have seen to be true for measurable cardinals), etc. So the axiom

```
(FORALL x | ord(x) *imp (EXISTS N | Px(N)))
```

implies the existence of many and very large cardinals. And, if one likes, one can repeat this construction after replacing the predicate 'Is_inaccessible' in (+) by

```
(EXISTS K | (FORALL x in K | ord(x) *imp (EXISTS N | Px(N)))).
```

These particular statements do not seem to have been studied enough for surmises concerning their consistency or inconsistency to have developed. But if they are all consistent, there will exist *inner models* of set theory, in the sense described in the next section, in which any finite collection of them are

true. This will allow theories containing such axioms to be covered by 'axioms of reflection' of the kind described below. Of course, all of this resembles the play of children with large numbers: *'a thousand trillion gazillion **plus one'.***

More general 'inner' models of set theory

A predicate model of the Zermelo-Fraenkel axioms must provide some set U as universe and assign a two-variable Boolean function E on U to represent the non-logical symbol 'in'. The most direct (but of course not the only) way of doing this is to choose a set U having appropriate properties and simply to define E as

```
E(x,y) = if x in y then 1 else 0 end if,
```

which can be written more simply as

```
E(x,y) *eq (x in y)
```

if we agree to represent predicates by true/false valued, rather than 0/1 valued, functions. (An element A(x) of U must be assigned to each free variable x appearing in a function whose value is to be calculated). Using this convention, and noting that the ZFC axioms involve no function symbols and so they do not require formation of any terms, we can write our previous recursive rules for calculating the value associated with each predicate expression F in the following slightly specialzed way:

(i) If the expression F is just an individual variable x, then Val(A,F) = A(x).

(ii) If F is an atomic formula having the form 'x in y', then Val(A,F) is the Boolean value A(x) in A(y).

(iii) If the formula F is an atomic formula having the form (FORALL $v_1,...,v_k$ | e), then Val(A,F) is

```
(FORALL x1,...,xk | (v1 in U & ... & vk in U) *imp          Val(A(x1,...,xk),e)),
```

where $A(x_1,...,x_k)$ assigns the same value as A to every free variable of e, but assigns the value x_j to each v_j, for j from 1 to k.

(iv) If F is a formula having the form (EXISTS $v_1,...,v_k$ | e), then Val(A,F) is

```
(EXISTS x1,...,xk | (v1 in U & ... & vk in U) &          Val(A(x1,...,xk),e)),
```

where $A(x_1,...,x_k)$ assigns the same value as A to every free variable of e, but assigns the value x_j to each v_j, for j from 1 to k.

(v) If the formula F has the form 'G & H', then Val(A,F) is Val(A,G) & Val(A,H).

(vi) If the formula F has the form 'G or H', then Val(A,F) is Val(A,G) or Val(A,H).

(vii) If the formula F has the form 'not G', then
Val(A,F) = (not Val(A,G)).

(viii) If the formula F has the form 'G *imp H', then Val(A,F) is Val(A,G) *imp Val(A,H).

(ix) If the formula F has the form 'G *eq H', then Val(A,F) is Val(A,G) *eq Val(A,H).

The set U defines a model of ZFC if and only if each of the ZFC axioms evaluates to 'true' under these rules. We shall now list a set of conditions on U sufficient for this to be the case.

We first suppose that U is *transitive*, i.e. that each member of a member of U is also a member of U. Then the first axiom of ZFC evaluates to

```
(FORALL s,t | (s in U & t in U) *imp     (s = t *eq (FORALL x | (x in U) *imp ((x in s) *eq (x in
```

This formula clearly has the value true. Indeed, if s = t, then (x in s) *eq (x in t) for every x in U, so clearly

```
(+)  (FORALL x | (x in U) *imp ((x in s) *eq (x in t)))
```

must be true. Suppose conversely that s /= t. Then by the ZFC axiom of extensionality, one of these sets, say s, has a member x that is not in the other. Since U is transitive we have x in U, so (+) must be false.

ZFC axiom (vi) (axiom of regularity) evaluates to

```
not (EXISTS x | (x in U) & (x /= 0) &            (FORALL y | ((y in U) & (y in x)) *imp
```

and this also must be true. Indeed, if x in U is non-null, then by the ZFC axiom of regularity it must have an element y which is disjoint from it, and since U is transitive this y is also in U.

Chapter 3. More on the Structure of the Verifier System

In this chapter we describe our verifier and its underlying design in more detail. The chapter falls into three parts: (i) An account of the general syntax and overall structure of proofs acceptable to the verifier. (ii) An extended survey of inference mechanisms which are candidates for inclusion in the verifier's initial *endowment*. (iii) A listing of the mechanisms actually chosen for inclusion in this endowment. We explain the syntax used to invoke each of the verifier's built-in inference mechanisms, and note the efficiency considerations which limit the complexity of the sets of statements to which each inference mechanism can be applied.

3.1. Introduction to the general syntax and overall structure of proofs

The syntax of proofs

Our verifier ingests bodies of text, which it either certifies as constituting a valid sequence of definitions, theorems, and auxiliary commands, or rejects as defective. When a text is rejected the verifier attempts to pinpoint the location of trouble within it, so that the error can be located and repaired. The bulk of the text normally submitted to the verifier will consist of successive theorems, some of which will be enclosed within theories whose external conclusions these internal theorems serve to justify.

The verifier allows input and checking of the text to be verified to be divided into multiple *sessions.*

Each theorem is labeled in the manner illustrated below. As seen in the following example, each theorem label is followed by a syntactically valid logical formula called the *conclusion* of the theorem.

```
Theorem 19: (enum(X,S) = S & Y incs X) *imp (enum(Y,S) = S).
```

The statement of each theorem should be terminated by a final period (i.e. '.') and be followed by its *proof,* which must be introduced by the keyword 'Proof:', and terminated by the reserved symbol 'QED'. A theorem's proof consists of a sequence of *statements* (also called *inferences),* each of which consists of a 'hint' portion separated by the sign ' ==> ' from the *assertion* of the statement. An example of such a statement is

```
ELEM ==> car([x,y]) in {x} & cdr([x,y]) = y & car([z,w]) notin {x} ,
```

where the 'hint' is 'ELEM' and the assertion is

```
car([x,y]) in {x} & cdr([x,y]) = y & car([z,w]) notin {x}.
```

As this example illustrates, the 'hint' portion of a statement serves to indicate the inference rule using which the 'assertion' is derived (from prior statements, theorems, definitions, or assumptions). The 'assertion' must be a syntactically well-formed statement in our set-theoretic language.

An example of a full proof is

```
Theorem 1a: arb({{X},X}) = X. Proof:    Suppose_not(c) ==> arb({{c},c}) /= c    {{c},c} --> Ax_ch ==> (({
```

Each 'hint' must reference one of the basic inference mechanisms that the verifier provides, and may also supply this inference mechanism with auxiliary parameters, including the *context* of preceding statements in which it should operate. The following table lists many of the most important of the inference mechanisms provided.

ELEM ==> ... Proof by extended elementary set-theoretic reasoning.

Suppose ==> ... Introduces hypothesis, available in local proof context, to be 'discharged' subsequently.

Discharge ==> ... Closes proof context opened by last previous 'Suppose' statement, and makes negative of prior supposition available.

Suppose_not ==> ... Specialized form of 'Suppose', used to open proof-by-contradiction arguments.

(e_1,..,e_n) --> Stat_label ==> ... Substitutes given expressions or newly quantified constants into a prior labeled statement.

Defmemb ==> ... Expands prior membership or non-membership statement into its underlying meaning.

EQUAL ==> ... Makes deduction by substitution of equals for equals, possibly in universally quantified form.

SIMPLF ==> ... Makes deduction by removal of set-former expressions nested within other setformers or quantifiers.

Use_Def(symbol) ==> ... Expands a defined symbol into its definition.

(e_1,..,e_n) --> Theorem_number ==> ... Substitutes given expressions into prior universally quantified theorem.

APPLY.. ==> ... Draws conclusions from theory established previously.

ALGEBRA ==> ... Deduces algebraic consequence from statements proved or assumed previously.

Statement conclusions and parts of compound conclusions connected by the conjunction sign '&' can be labeled for explicit subsequent reference within the same proof by appending a reserved notation of the form 'Stat_nnn:' to them, where '_nnn' designates any integer. (An example of such a label is 'Stat3:'). These are the labels used in hints of the form

$(e_1,..,e_n)$ --> Stat_label ==> ...,

as shown in the table above.

The context of a hint defines the collection of preceding statements, within the theorem in which the hint appears, which the inference mechanism invoked by the hint should use in deducing the assertion to which the hint is attached. Since in some cases the efficiency of an inference mechanism may degrade very rapidly (e.g. exponentially or worse) with the size of the context with which it is working, appropriate restriction of context can be crucial to successful completion of an inference. Inferences which the verifier cannot complete within a reasonable amount of time are abandoned with a diagnostic message 'Abandoned...', or with the more specific message 'Failure...' if the inference method is able to certify that the attempted inference is impossible. Hint directives like ELEM, EQUAL, SIMPLF, and ALGEBRA which do not automatically carry context indications can be supplied with such indications by prefixing them with a statement label, or a comma-separated list of such labels, as in the examples

```
(Stat3)ELEM ==> s notin {x *incin o | Ord(x) & P(x)}
```

and

```
(Stat3,Stat4,Stat9)ELEM ==>       s notin {x *incin o | Ord(x) & P(x)}.
```

The first form of prefix defines the context of an inference to be the collection of all statements in the proof, back to the point of last previous occurrence of the statement label appearing in the proof (but not within ranges of the proof that are already closed in virtue of the fact that they are included between a preceding 'Discharge' statement and its matching 'Suppose' statement; see below). The second form of prefix defines the context of an inference to be the collection of statements explicitly named in the prefix. If no context is specified for an inference, then its context is understood to be the collection of all preceding statements in the same proof (not including statements enclosed within previously closed 'Suppose/Discharge' ranges) form its context. This default is workable for simple enough inferences in short enough proofs.

The automatic treatment of built-in functions like the cons-car-cdr triple by the methods described later in this chapter often poses efficiency problems, since the method used adds multiple implications which may force extensive branching in the search required. For example, automatic deduction of

```
([x,[y,z]] = [x2,[y2,z2]]) *imp       ((x = x2) & (y = y2) & (z = z2))
```

takes about 40 seconds on an 400Mhz Macintosh G4. For this reason the verifier provides a few efficiency-oriented variants of the ELEM deduction primitive. These are invoked by prefixing the keyword ELEM with a parenthesized label in the manner described above, which may be preceded with various special characters whose significance will be explained later. Including the character '*' just before the closing ')' of the prefix suppresses the normal internal examination of special functions like cons,car,cdr, i.e. it treats these as unknown functions whose occurrences must be 'blobbed'. (Appended characters like '*' are not regarded as parts of other labels contained in the same parentheses). This treats statements like

```
([x,[y,z]] = [x2,[y2,z2]] & [x,[y,z]] = [x3,[y3,z3]] &       [x,[y,z]] = [x4,[y4,z4]])
```

as if they read

```
(xyz = xyz_2 & xyz = xyz_3 & xyz = xyz_4),
```

and so makes deduction of

```
[x2,[y2,z2]] = [x3,[y3,z3]]
```

from the formula shown easy. Without modification of the ELEM primitive's operation this same deduction would require many minutes. This coarse treatment is of course incapable of deducing the implication

```
([x,[y,z]] = [x2,[y2,z2]]) *imp       ((x = x2) & (y = y2) & (z = z2))
```

which it sees as

```
(xyz = xyz_2) *imp ((x = x2) & (y = y2) & (z = z2)).
```

In such cases we must simply allow a more extensive search than is generally used. (The verifier normally cuts off ELEM deduction searches after about 10 seconds). Including the character '+' instead of '*' in a prefix attached to ELEM raises this limit to 40 seconds. Note that an empty prefix, i.e. '( )',can be used to indicate that a statement is to be derived without additional context, i.e. that it is universally valid as it stands. Therefore the right way of obtaining the implication just displayed by ELEM deduction is to write it as

```
(+)ELEM ==> ([x,[y,z]] = [x2,[y2,z2]]) *imp       ((x = x2) & (y = y2) & (z = z2)).
```

Within the body of a proof all free variables of formulae are treated as logical constants, i.e. none are understood to be universally quantified, and so no inference can be made by substituting any expression or different variable for such a variable, unless an equality or equivalence allowing such replacement as an instance of equals-for-equals substitution is available in the relevant context.

A somewhat different syntax is used to abbreviate the statements of theorems. In that setting, every variable that is not otherwise quantified (or defined) is understood to be universally quantified. For example, the theorem

```
Theorem 1: arb({X}) = X
```

should be understood as reading

```
(FORALL x | arb({x}) = x).
```

Variables used in this way are capitalized for emphasis.

Our verifier's 'Suppose' and 'Discharge' capabilities make a convenient form of 'natural deduction' available. Any syntactically well-formed formula can be the 'conclusion' of a 'Suppose' statement, i.e. you can suppose what you like. For example,

```
Suppose ==> 2 *PLUS 2 = 4
```

and

```
Suppose ==> 2 *PLUS 2 = 5
```

are both perfectly legal. However, all the assumptions made in the course of a theorem's proof must be Discharged before the end of the proof. This is accomplished by matching each 'Suppose' statement with a following 'Discharge' statement. The matching rule used is the same as that for opening and closing parentheses. A Discharge statement of the form

```
Discharge ==> some_conclusion
```

constructs its conclusion as p *imp q, where p is the 'conclusion' of the matching Suppose statement and q is the conclusion of the last inference preceding the Discharge. For example, the following sequence of 'Suppose' and 'Discharge' statements proves the propositional tautology P *imp ((P *imp Q) *imp Q).

```
Suppose ==> P    Suppose ==> P *imp Q    ELEM ==> Q    Discharge ==> (P *imp Q) *imp Q    Discharge ==>
```

Dividing long proof verifications into multiple separate 'sessions'

Several seconds of computer time may be required to certify conclusions dependent on contexts that are at all complex. For this reason, it is often appropriate to divide the verification of lengthy sequences of proofs into multiple successive verifier sessions. The following verifier mechanism makes this possible. Two special verifier directives 'SAVE(file_name)' and 'RESTART(file_name1,file_name2)' are provided. In both of these commands, 'file_name' should name some file available in the file system of the computer on which the verifier is running. When encountered, SAVE(file_name) writes all the theorems, definitions, and theories established prior to the point at which it is encountered. These are written to the named file along with one half H_1 of a cryptographically secure checksum for the file. The other half H_2 of the checksum is retained by the verifier in a hidden data structure that allows H_2 to be retrieved if H_1 is given. The file names of any session record written in this way can be passed to the RESTART(file_name1,file_name2) command as its first parameter. The second parameter 'file_name2' should be the name of a text file of purported proofs of additional theorems which are to be verified. The verifier then reads all the definitions, theorem statements, and theory descriptors previously written to file_name1, which it can accept as valid without additional verification once the fact that the text in the file conforms to the two available checksum halves is verified. These definitions, theorem statements, and theories then become available for use in the session opened by the RESTART(file_name1,file_name2) statement. Once some or all of the new text supplied in file_name2 has been brought to the point at which it will verify, a new 'SAVE(file_name)' statement can be executed to store the newly certified definitions, theorem statements, and theory descriptors. In this way large libraries of theorems can be accumulated through multiple verifier sessions. Note that proof files written by the SAVE(file_name) operation can be copied without losing their validity, and so can be made available over the Web as community resources.

A few supplementary commands are provided to increase the flexibility of the verifier's multi-session capability. The commands

```
DELETE_THEOREM(theorem_label1,.., theorem_labeln)
```

and

```
DELETE_THEORY(theory_label1,.., theory_labeln)
```

delete comma-separated lists of labeled theorems and theories respectively. The command

```
DELETE_DEFINITION(symbol1,..,symboln)
```

deletes the definition of all labeled symbols, along with all theorems and further definitions in which any symbol with a deleted definition appears. The parameter of the command

```
RENAME(old_symbol1,new_symbol1;..;old_symboln,new_symboln)
```

must be a semicolon-separated list of symbol pairs delimited by commas. The new_symbols which appear must be predicate and function symbols never used before. This command replaces each occurrence of every old_symbol_j in every theorem, definition, and theory known at the point of the RENAME command by the corresponding new_symbol_j.

The RESTART command is available in the generalized form

```
RESTART(file_name1,..,file_namen,file_namen + 1).
```

Here file_name$_1$,..,file_name$_n$ must be a list of files, each written by some preceding SAVE(file_name) command, and file_name$_{n+1}$ should be the name of a text file of purported proofs of additional theorems which are to be verified. After examining the checksums of file_name$_1$,..,file_name$_n$ to ensure their validity, the contents of these files are scrutinized to verify that all symbols defined in more than one of these files have identical definitions in all the files in which they are defined, and that all theorems and theories with identical labels are completely identical. If the files pass this test, their contents are combined and the new-text file file_name$_{n+1}$ is then processed in the normal way.

The syntax and semantics of definitions

Definitions introduce new predicate and function symbols into the ken of our verifier. Predicate definitions have the syntactic form

```
P(x1,x2,...,xn) :*eq pexp.
```

Function definitions have the form

```
f(x1,x2,...,xn) := fexp.
```

In both these cases, $x_1,x_2,...,x_n$ must be a list of distinct variables; only these variables can occur unbound on the right of the definition, and P (resp. f) must be a predicate (resp. function) symbol that has never been defined previously. In the first (resp. second) case pexp (resp. fexp) must be a syntactically well-formed predicate expression (resp. function expression). Two cases of each form of definition, the non-recursive and the recursive, arise. In non-recursive predicate (resp. function) definitions, pexp (resp. fexp) can only contain previously defined predicate and function symbols, plus the free variables $x_1,x_2,...,x_n$ (and, of course, any other bound variables). In recursive definitions the predicate (resp. function) symbol being defined is allowed to appear on the right-hand side of the definition, but then other syntactic conditions must be imposed to guarantee the legality of the definition. More specifically, in the function case, we allow recursive definitions of the general form

$$f(s,x_2,...,x_n) := d(\{g(f(x,h_2(s,x,x_2,...,x_n),h_3(s,x,x_2,...,x_n),...,h_n(s,x,x_2,...,x_n)),s,x,x_2,...,x_n): x \text{ in } s \mid P(f(x,h_2(s,x,x_2,...,x_n),h_3(s,x,x_2,...,x_n),...,h_n(s,x,x_2,...,x_n)),s,x,x_2,...,x_n)\},s,x_2,...,x_n).$$

Here g, d, and $h_2,...,h_n$ must be previously defined functions of the indicated number of arguments, and P must be a previously defined predicate of the indicated number of arguments.

The following informal argument indicates why it is reasonable to expect definitions of the general form displayed above to specify a function that is well defined for each possible argument list $s,x_2,...,x_n$. If the initial argument s is the null set { }, the definition reduces to

$$f(\{\},x_2,...,x_n) := d(\{\},\{\},x_2,...,x_n),$$

i.e. to an ordinary set-theoretic definition in which the function being defined does not appear on the right. Since, in intuitive terms, we can think of the collection of all sets as being arranged in a members-first order, we can suppose that $f(x,y_2,...,y_n)$ is known for each x in s and for all $y_2,...,y_n$ before the value $f(s,x_2,...,x_n)$ is required. But then the definition shown above clearly specifies $f(s,x_2,...,x_n)$ in terms of (i) values of f which are already known, (ii) known functions and predicates, along with (iii) a single setformer operation.

Although it is not hard to convert this informal line of reasoning into a more formal argument involving transfinite induction, we shall not do so, but will simply allow free use of inductive definitions of the form shown above.

In the predicate case, the same line of reasoning shows that we can allow recursive definitions of the form

$$P(s,x_2,...,x_n) :*eq\ d(\{g(s,x,x_2,...,x_n)\colon x \text{ in } s \mid P(x,x_2,...,x_n)\},s,x_2,...,x_n) = \{\},$$

where again g and d must be previously defined functions of the indicated number of arguments. In the special case in which the function d has the form

$$d(t,s,x_2,...,x_n) = \{x\colon x \text{ in } t \mid (\text{not } Q(s,x,x_2,...,x_n))\},$$

where Q is some previously defined predicate, the recursive predicate definition seen above can be recast in the form

$$P(s,x_2,...,x_n) :*eq\ (\text{FORALL } x \text{ in } s \mid P(x,x_2,...,x_n)\ *imp\ Q(s,x,x_2,...,x_n)).$$

Accordingly, we allow recursive predicate definitions of this latter form also.

To illustrate the use of recursive definitions, we show how one can define functions on sets which, when they are restricted to natural numbers in the von Neumann representation, become the usual operations of unitary incrementation and decrementation, addition, multiplication, subtraction, quotient, remainder, and greatest common divisor (for this, we use an auxiliary operation 'coRem(X,Y)', which finds the maximum multiple of Y less than or equal to X):

```
next(W) := W + {W},    prec(V) := arb({ w: w in V | next(w)=V }),    plus(X,Y) := X + Un({ next(plus(X,
```

An alternative definition of the greatest common divisor, syntactically equally convenient but more procedural in flavor (indeed, inspired by the classical Euclid algorithm), can be given as follows:

```
gcd(X,Y) := if Y=0 then X else gcd(Y,rem(X,Y)) end if.
```

Auxiliary verifier commands

3.2. A survey of inference mechanisms

In addition to the discourse-manipulation mechanisms described earlier in this chapter, the verifier depends critically on a collection of routines which work by combinatorial search. These are able to examine certain limited classes of logical and set-theoretic formulae and determine their logical validity or invalidity directly. Together they constitute the verifier's inferential core. In the following paragraphs we will examine a variety of candidate algorithms of this kind. While all of these (plus many others too complex to be described here) are interesting in their own right, not all are worth including in the verifier's initial endowment of deduction procedures, since some are too inefficient to be practical, while others are too specialized to be applied more than rarely in ordinary mathematical discourse. The selection actually made in the verifier will be detailed once the collection of candidates that suggest themselves has been reviewed. We begin this review by discussing one of the most elementary but important decision procedures, the *Davis-Putnam* technique for deciding the validity of sets of propositional formulae.

The Davis-Putnam propositional decision algorithm

The Davis-Putnam algorithm works on collections C of propositional formulae, each supposed to be a disjunction of the form

```
(+)   P1 or P2 or ... or Pn
```

with n>=1, where each P_j is either a propositional symbol or its opposite. It determines, for each such collection, whether it is *satisfiable,* i.e. whether there exists an assignment of truth values to the propositional symbols appearing in the

statements of C which makes all these statements true, or unsatisfiable.

The flavor of the collections of propositional formulae (+) which the Davis-Putnam procedure takes as input can best be understood by moving all the negated symbols P_j = (not Q_j) to the left side of each formula and then rewriting it as

(++) $(Q_1 \ \& \ Q_2 \ \& \ \ldots \ \& \ Q_k) \ *\text{imp} \ (P_{k+1} \ \text{or} \ \ldots \ \text{or} \ P_n)$,

where now all propositional symbols are non-negated. This allows us to recognize Davis-Putnam input disjunctions (+) as implications in which multiple conjoined hypotheses Q_j imply one of several alternate conclusions P_i. We see at once that sets of clauses of this type are quite typical for ordinary mathematical discourse, and that most typically they will contain just one conclusion P_i rather than several alternative conclusions. We also are forewarned that if many of the clauses in our input set C contain multiple alternative conclusions, the argument necessary to analyse C's satisfiability will probably involve inspection of an exponentially growing set of possible cases.

The Davis-Putnam procedure is designed to work very efficiently on sets of clauses which can be written as implications containing no or few alternative conclusions. It works as follows in an input set of formulae (+).

> (1) If possible, find a formula F in C consisting of just one propositional atom Q, either negated (i.e. F is 'not Q') or non-negated (i.e. F is Q). Assign Q the value 'false' if it occurs negated; otherwise assign it the value 'true'.
>
> (2) If step (1) succeeds, remove F from C, along with every formula G in which Q occurs with the same sign as in F. This reflects the fact that all these G are already satisfied, since 'H or true' is propositionally equivalent to 'true' for every proposition H. Also, remove the negation of F from every formula G in which Q occurs with sign opposite to that seen in F. This reflects the fact that 'H or false' is propositionally equivalent to H for every proposition H.
>
> If step (2) ever generates an empty set of propositions, then the whole initial set is clearly satisfied by the sequence of truth values assigned. If it ever generates an empty disjunction (resulting from the fact that two opposed propositions Q and 'not Q' have been seen), then the search ends in failure, since a propositional contradiction has been found.
>
> (3) If step (1) fails, we can find no propositional symbol whose truth value is immediately evident. In this case, we proceed nondeterministically, by choosing some symbol Q that appears in one of the formulae remaining in C, and guessing it to have one of the two possible truth values 'true' and 'false'. Guessing that Q is true amounts to adding to C the formula F consisting of Q alone, and guessing that Q is false amounts to inserting the negation of Q into C. Thus, in either case, the recursive execution of step (1) is enabled. If this eventually leads to truth values satisfying all the remaining propositions of C we are done; otherwise we backtrack to the (last) point at which we have made a nondeterministic guess, and try the opposite guess. If both guesses fail, then we fail overall. A chain of failures back to the point of our very first guess implies that the input set C of propositions in not satisfiable.

It is easily seen that if we think of a set of Davis-Putnam input clauses as having the form (++), then the maximum number of nondeterministic trials that can occur in steps (3) is at most the product K of the numbers n of possible alternative conclusions appearing in clauses of the input. Although this can be exponentially large in the worst possible case, it will not be large in typical mathematical situations. Thus we can generally rely on the Davis-Putnam algorithm to handle the propositional side of our verifier's work very effectively.

The Davis-Putnam algorithm can easily be adapted to generate the set of *all* truth-value assignments which satisfy a given set C of input clauses. For this, we search as above, until a satisfying assignment is found, then collect this assignment into a set of all such assignments, but signal the algorithm to behave as if search has failed, so that it will backtrack in the

manner described above until it has found the next possible assignment. When no more satisfying assignments can be found, we have collected the set TVA of all truth-value assignments which satisfy all the clauses in C. Note that the argument given in the previous paragraph shows that the number of elements in TVA can be no larger than the product K considered there.

If we are using the Davis-Putnam algorithm simply to search for one truth-value assignment satisfying the set of clauses C, rather than searching for the set of all such assignments, then it can be improved by including the following step (2b) immediately after the step (2) seen above:

> (2b) If any propositional symbol Q occurs in all remaining statements of C with the same sign (that is, either always negated or always non-negated), then give Q the corresponding truth-value (i.e. 'false' if it always occurs negated, 'true' otherwise), and remove all the clauses containing Q from C.

This must work since if our clauses have any satisfying assignment, we can change the assignment to give Q the truth value specified by rule (2b), since all clauses not containing Q will clearly still be satisfied, but equally clearly the clauses not containing Q will be satisfied also.

Horn formulae and sets of formulae

A propositional formula

```
(+)   P1 or P2 or ... or Pn,
```

is called a *Horn formula* if at most one of the propositional symbols in it occurs non-negated, and a set C of such formulae is called a *Horn set.* It is easily seen that any such set C which does not contain (either the empty disjunction or) at least one 'linked' positive 'unit' formula A (i.e. a formula consisting of just the single propositional symbol A that also occurs negated in some other formula) must be satisfiable. For clearly if we give the value 'true' to every symbol A that appears as a positive unit clause of C, and 'false' to every symbol that occurs negated in a formula of C, all the formulae in C will be satisfied. It follows from this that in the case of an unsatisfiable set C of Horn clauses the Davis-Putnam algorithm will never run out of unit clauses before deducing an empty clause, and so need never use its recursive step (3). In this case, the algorithm will run in time linear in the total length of its input.

For later use it is worth noting that we can look at such 'Horn' cases in a different, somewhat more 'algebraic', way. The non-negated unit formulae A can be considered to be 'inputs', and the formulae

```
    (not B1) or (not B2) or ... or (not Bm)
```

which only consist of negated propositional symbols to be 'goals'. The remaining clauses, which must all have the form

```
   (A1 & A2 & ... & An) *imp B,
```

can be seen as 'multiplication rules' which allow collections $A_1,A_2,...,A_n$ of inputs to be combined to generate new inputs B. Proof of unsatisfiability results once a sequence of multiplications leading to the opposites B_j of all constituents 'not B_j' of a goal formula is found. Note that this observation shows that a Horn set is unsatisfiable if and only if some one of its subsets obtained by dropping all but one of its goal formulae is unsatisfiable.

Reducing collections of propositional formulae to collections of standardized disjunctions

Since ordinary mathematical statements generally have the form

```
multiple_hypotheses *imp single_conclusion,
```

most of the propositional inferences arising in ordinary mathematical practice convert very readily into the disjunctive Horn form favorable for application of the Davis-Putnam algorithm as soon as their non-propositional elements are reduced ('blobbed down') to propositional symbols. Other formulae can be converted into collections of disjunctions using the following straightforward procedure:

1. Express all other propositional operators in the given collection of propositional formulae by their expressions in terms of the operators &, 'or', and 'not'.
2. Move all the negations down in the syntax trees of these formulae by using de Morgan's rules: 'not (a & b)' is equivalent to '(not a) or (not b)', etc. Use the rule (not (not a)) *eq a to eliminate all double negations.
3. Use the fact that disjunction is distributive over conjunction to 'multiply out' wherever a disjunction of conjunctions is encountered, thereby reducing each formula to a conjunction of disjunctions, each such disjunction involving only propositional atoms and their opposites.

Although in most cases encountered in ordinary mathematical practice this recipe will work well, in some cases its third step can expand one of the initial formulae into exponentially many conjunctions. This will, for example, be the case if we multiply out a formula of the form

$(a_1 \ \& \ b_1) \ or \ (a_2 \ \& \ b_2) \ or \ ... \ or \ (a_n \ \& \ b_n).$

In such cases we can use an alternative, equally easy, approach, which however replaces our original set of propositional formulae, not by logically equivalent formulae, but by equisatisfiable formulae (since new variables are introduced). This alternative method is guaranteed to increase the length of our original collection by no more than a constant factor. It works as follows: after applying the above steps (1) and (2), progressively reduce the syntax tree of each of the resulting collection of formulae by working progressively upwards in the tree, replacing each conjunction 'a & b' and each disjunction 'a or b' introducing a new variable c which replaces 'a & b' (resp. 'a or b'), along with a conjoined clause 'c *eq (a & b)' (resp. 'c *eq (a or b)'), which we can write as

```
((not a) or (not b) or c) & ((not c) or a) & ((not c) or b)
```

in the first case and as

```
((not c) or a or b) & ((not a) or c) & ((not b) or c)
```

in the second. After elimination of double negatives, the resulting collection of formulae clearly has the asserted properties, proving our claim.

A reduction technique very similar to this reappears in the following discussion of the decidability of the elementary unquantified theory of Boolean set operators, where it will be called *secondary decomposition.*

Elementary Boolean theory of sets

Now we move on from the easily decidable statements of the purely propositional calculus to a somewhat larger but still practicable case, namely that of statements formed using the propositional operators plus the elementary Boolean operators and comparators of set theory: *, +, -, incs, *incin, and '='. It is convenient to allow the null set { }, as a constant. Simple examples of statements that can be formed using these operators are

```
(a incs b & b incs c) *imp (a incs c)
```

and

```
(a incs b & b * c = {}) *imp (a - c incs b),
```

both of which are universally valid.

Statements of this general form can be considered in either of two possible settings, that in which quantifiers are forbidden (as in the examples seen above), and that in which quantifiers are allowed, as in the example

```
(FORALL a | (not (a * b = {})) *imp (a incs b)).
```

If quantifiers are forbidden we describe the language which confronts us as being *unquantified;* in the opposite case we speak of the *quantified* case. Both cases are decidable, but unsurprisingly the quantified case (which is analyzed in a later section of this chapter) is substantially more complex. Indeed, the last formula displayed is readily seen to be equivalent to #b = 1 or #b = 0. This hints at the fact that analysis of such quantified statements must involve consideration of the number of elements in the sets which appear, a perception which we will see to be true when we come to analyse this case. For this reason we confine ourselves in this section to the much more elementary unquantified case.

This case is quite easy, and can be handled in any one of a number of ways. With an eye on what is to follow, we choose to follow an approach based on the notion of *place,* which can be described as follows. Given a collection of unquantified statements formed using propositional connectives and the elementary set operators and comparators listed above, and having the goal of testing these statements for satisfiability, we can begin by using the Davis-Putnam algorithm (or any other propositional-level algorithm of the same kind) to determine all the propositional-level truth-value assignments which would verify all the statements in our collection. Each of these truth-value assignments gives rise to some collection of negated and non-negated atomic formulae of our language, no longer containing any propositional operators. These collections of formulae must then be tested for satisfiability. If any such collection is found to be satisfiable, then so are our original formulae. If no truth-value pattern satisfying our original formulae at the propositional level gives rise to a collection of atomic formulae which can be satisfied at the underlying set-theoretic level, then our original formula collection is plainly unsatisfiable. We shall refer to this preliminary propositional level step as *decomposition at the propositional level.*

We can equally readily eliminate all compound expressions such as a + (b * c) formed using the available operators *, +, -, by introducing new auxiliary variables t and equalities like t = b * c, which allows compound expressions like a + (b * c) to be rewritten as a + t. Similarly, inequalities like not(a = b + c) can be reduced to inequalities of the simpler form not(a = t) by introducing auxiliary variables t and replacing not(a = b + c) by the equisatisfiable pair of statements t = b + c, not(a = t). Once simplifications of this second kind, which we will call *secondary decomposition,* have been applied systematically, what remains is a collection of atomic formulae, each having one of the forms

```
x = y * z ,    x = y + z ,    x = y - z ,   x = {} ,    x = y ,    x incs y ,
```

together with statements of the form 'not (x = y)'. Note that all uses of the comparator *incin can be eliminated, since 'x *incin y' is just 'y incs x'.

Next we make use of the following concept.

Definition: A *place* p for a collection C of atomic statements formed using the null set constant {} and the operators and comparators *, +, -, incs, and '=', is a Boolean-valued map p(x) defined on all of the set-valued variables appearing in propositions of C for which we have

p(x) *eq (p(y) & p(z)) whenever x = y * z appears in C ,

p(x) *eq (p(y) or p(z)) whenever x = y + z appears in C ,

p(x) *eq (p(y) & (not p(z))) whenever x = y - z appears in C ,

p(x) *eq p(y) whenever x = y appears in C ,

p(x) *eq false whenever x = { } appears in C ,

p(y) *imp p(x) whenever x incs y appears in C .

Note that this notion depends only on the subcollection of non-negated formulae in C.

Definition: A collection S of places for C is *ample* if, for each negated statement not (x = y) in C, there exists a p in S such that not(p(x) *eq p(y)).

Theorem: A collection C of atomic statements formed using the operators and comparators *, +, -, incs, and '=', and the null set constant { } is satisfiable if and only if it has an ample set A of places.

Proof: First suppose that C is satisfiable, so that it has a model M, i.e. there exists an assignment M(a) of an actual set to each variable a appearing in the statements of C, such that replacement of each of these variables by the corresponding set M(a) makes all the statements of C true. Let U be the 'universe' of this model, i.e. the union of all the sets M(a), and let x belong to U. Then, for each point u in U, the formula

```
(+)  pu(x) *eq (u in M(x))
```

defines a place. Indeed, if x = y * z appears in C, we have M(x) = M(y) * M(z), so $p_u(x)$ *eq ($p_u(y)$ & $p_u(z)$), and similarly if x = y + z appears in C, etc. For negated statement in C like 'not (x = y)' we must have M(x) /= M(y), and so there must exist a point u in U such that u in M(x) and u in M(y) have different truth values, that is, not($p_u(x)$ *eq $p_u(y)$). Hence the set of places deriving from M via the formula (+) is ample.

Conversely let A be an ample set of places. Then we can build a model M with universe A by setting

```
    M(x) = {p: p in A | p(x)}.
```

The conditions on places displayed above clearly imply that M is a model of all the positive statements in C. But since A is ample, we have M(x) /= M(y) whenever a statement 'not(x = y)' is present in C, so that the negative statements in C are modeled correctly also. **QED.**

Note that the places p deriving via formula (+) from a model M of any set C of statements serve to classify the points u in the universe of the model into subsets s = {x in U | p(x)} which are either contained in or disjoint from each of the sets M(x). Conversely if we assign disjoint sets M_p to the places p in an ample set A of places in any way, then the union set

```
(++)     M(x) = Un({Mp: p in A | p(x)})
```

is a model of the statements in C. Hence altogether, we see that all models of statements in C have this form. This observation will be applied just below.

The technique used in this section, of simplifying collections of statements whose satisfiability is to be determined, first by removing all propositional operators using a preliminary decomposition step, and then reducing all compound expressions by introducing auxiliary variables, will be used repeatedly and implicitly in what follows.

Elementary Boolean theory of sets, plus the predicates 'Finite' and 'Countable'

We now generalize the unquantified language considered in the preceding section by allowing two additional predicates on sets, namely Finite(s), which states that s is finite, and Countable(s), which states that s is either finite or denumerably

infinite. (As usual this allows us to write the corresponding negated predicates 'not Finite(x)' and 'not Countable(x)'). In this expanded language we can test candidate statements like

```
(*)  (a + b incs c & Countable(a) & Countable(b)) *imp Countable(c)
```

for satisfiability.

To see how statements in this expanded language can be tested for satisfiability, we have only to use the formula (++) shown above. We saw above that any model M of a collection C of statements involving only Boolean operators and comparators can be analyzed into this form. Let fi (resp. co) be the set of all places p for which M_p is finite (resp. countably infinite), and let Fi and Co be the two union sets

```
    Fi = Un({Mp: p in fi}),     Co = Un({Mp: p in fi + co}).
```

Then plainly any set x for which a statement Finite(x) (resp. Countable(x)) is present in C must satisfy

```
    Fi incs x   (resp. Co incs x).
```

Also, any set x for which a statement 'not Finite(x)' (resp. 'not Countable(x)') is present in C must satisfy

```
     not(Fi incs x)   (resp. not(Co incs x)).
```

Conversely, suppose that we are given any collection of statements C involving Boolean operators and comparators only, along with assertions of the forms Finite(x), Countable(x), not Finite(x), and not Countable(x) for some of the sets x mentioned in the statements of C. Introduce two new variables Fi and Co, and for these variables introduce the following statements:

```
(**)  Co incs Fi;    for each x for which a statement Finite(x) is present,         a statement Fi incs x;
```

Then drop all statements of the forms Finite(x), Countable(x), not Finite(x) not Countable(x).

It is plain from what was said above that if our original collection of statements has a model, so does our modified collection. Conversely, if this modified collection has a model, then we can assign disjoint sets M_p to the places p associated with this model according to the following rule:

```
  if p(Fi), then let Mp be some single element;  otherwise, if p(Co), then let Mp be some countably infinite
```

It then follows from the collection of statements (**) that M(x) is finite (resp. countable) for each variable x for which a statement 'Finite(x)' (resp. 'Countable(x)') was originally present. Moreover if a statement 'not Finite(x)' was originally present, we must have not(Fi incs x), so there must exist a place p for which p(Fi) is false and p(x) is true, and then plainly M(x) is not finite. Since much the same argument can be used to handle statements 'not Countable(x)' originally present, it follows that our original set of statements has a model if and only if the modified version described above has a model. As an example, note that the negative of the statement

```
(*)  (a + b incs c & Countable(a) & Countable(b)) *imp         Countable(c)
```

considered above is

```
    a + b incs c & Countable(a) & Countable(b) &         (not Countable(c)).
```

The procedure we have described transforms this into

```
a + b incs c & Co incs a & Co incs b & (not(Co incs c)).
```

Since this is clearly unsatisfiable, the universal validity of our original statement follows.

Elementary Boolean operators on sets, with the cardinality operator and additive arithmetic on integers

In the present section we generalize the results described in the preceding section by introducing a different type of variable n, now denoting integers and a set-to-integer operation #s. For variables of integer type we allow the operations n + m (integer addition) and n - m (integer subtraction); also the integer comparator n > m, and a constant designating the integer 0.

Quantified predicate formulae involving predicates of one argument only

Quantified formulae of the predicate calculus involving only predicates of a single argument and no function symbols can be decided rather easily as for satisfiability by relating them to elementary set-theoretic formulae of the kind considered above. This can be done as follows. Let F be any such formula. First remove all propositional '*imp' and '*eq' operators by replacing them with appropriate combinations of the operators &, 'or', and 'not'. Then introduce a set name p for each predicate name P appearing in the original formula, and using these rewrite each atomic formula P(x) as 'x in p'. This step is justified since if the original formula has a model M with universe U, then M will associate a Boolean-valued function M(P) with each predicate name P appearing in F, and we can simply interpret each corresponding p as the set

```
{x: x in U | M(P)(x)}.
```

Next, working upward in the syntax tree from its twigs toward its root, process successive quantifiers in the following way, so as to remove them. (The approach we are using is accordingly known as *quantifier elimination*).

(i) Rewrite universal quantifiers '(FORALL x |...)' as the corresponding existential 'not (EXISTS x | not ...)'.

(ii) Use the algebraic rules for the operators &, 'or', 'not' to rewrite the *body* of each existential (EXISTS x |...) (i.e. the part of it following the sign '|') as a disjunction of conjunctions, that is, in the form

```
(A1 & A2 & ... & Ai) or (B1 & B2 & ... & Bj) or ...,
```

where of each elementary subpart A,B,... which appears is either of the form 'x in P', or of the negated form 'not (x in P)', or is a subformula not involving x as a free variable. Then use the predicate rules

```
(EXISTS x | A(x) or B(x)) *eq ((EXISTS x | A(x)) or         (EXISTS x | B(x)))
```

and

```
(EXISTS x | A(x) & C) *eq ((EXISTS x | A(x)) & C)
```

(where x has no free occurrences in C) to reduce the existential quantifier being processed to the form

```
(EXISTS x | A1 & A2 & ... & An),
```

where each A_i appearing is either of the form 'x in P' or 'x notin P'. This confronts us with an existential formula of the form

```
(EXISTS x | x in P1 & ... & x in Pm & x notin Pm + 1       & ... & x notin Pn),
```

which we can rewrite as

```
P1 * ... * Pm * (U - Pm + 1) * ... * (U - Pn) /= {}.
```

It is clear that we can apply this procedure until no quantifiers remain, at which point we will have derived a formula F' of the unquantified language of elementary Boolean set operations considered previously which is equisatisfiable with our initial quantified formula F. By testing F' for satisfiability using the method described above, we therefore can determine whether F is satisfiable. Note that clauses

```
U incs Pj
```

and a clause U /={} implying that the universe U is non-null and includes all the other sets which appear in our formula must be added just before the final satisfiability check is applied.

Note also that this procedure converts our original collection of quantified formulae into a collection of purely Boolean statements about the sets {x: x in U | P(x)}, which can however involve arbitrary intersections of these sets and their complements.

As an example of this procedure, consider the formula

```
(+)  (EXISTS x | (EXISTS y | P(y)) *imp P(x))
```

examined in an earlier section. The negation of this is

```
  not (EXISTS x | (not (EXISTS y | P(y)) or P(x))).
```

Processing this as above we get

```
  not (P = {} or P /= {}) & U incs P & U /= {}.
```

which is clearly unsatisfiable. Hence (+) is universally valid.

Various somewhat more general quantified cases can be reduced to the case just treated. For example, suppose that as above we take quantified formulae of the predicate calculus involving only predicates of a single argument, but now also allow function symbols of a single variable. If the function symbols sometimes appear compounded within predicates, as in the example P(f(g(h(x)))), we can introduce auxiliary new predicate symbols P^f and P^{fg} along with defining clauses

```
  (FORALL x | Pf(x) *eq P(f(x)))  and (FORALL x | Pfg(x) *eq Pf(g(x))),
```

and then rewrite P(f(g(h(x)))) as $P^{fg}(h(x))$.

Suppose that there exists a model M with universe U of the collection of statements, which must therefore model all the predicates P and functions f in such a way as to make all the quantified statements in our original collection C of statements true. Associate the set

```
  SP = { x in U | P(x) }
```

with each predicate P, and the set

```
  SPf = { x in U | P(f(x)) }
```

with each predicate symbol P and function symbol f. Then S_{Pf} is the inverse image of S_P under the map M(f) modeling f. Let $P_1,...,P_n$ be all the predicate symbols inside of which f appears (as $P_j(f(x))$ for some variable x), let

```
(i)  SP(1) * SP(2) *...* SP(k) - (SP(k+1) *...* SP(n))
```

be some intersection of the sets S_{P_j} and their complements, and let

```
(ii)  SP(1)f * SP(2)f *...* SP(k)f - (SP(k+1)f *...* SP(n)f)
```

be the corresponding intersection of the sets S_{P_jf}. It follows that if the first of these sets is empty so is the other, and conversely. Hence, if a model M for our collection of quantified statements exists, there must exist a model for the collection of sets S_{P_j} and S_{P_jf} which satisfies all the conditions

```
(iii)  SP(1) * SP(2) *...* SP(k) - (SP(k+1) *...* SP(n)) = {} *eq       SP(1)f * SP(2)f *...* SP(k)f - (SP(k+1)f *...* SP(
```

Earlier in this section we developed a systematic method for converting every collection of quantified statements involving only predicates of the form P(x) to an equisatisfiable collection C' of statements about the sets $S_P=\{x \mid P(x)\}$, together with their intersections and complements. If we employ this procedure in the present case, we get a collection C" of statements about the sets $S_P=\{x|P(x)\}$ and $S_{Pf}=\{x \mid P(f(x))\}$, together with their intersections and complements, which must be satisfied even if the conditions (iii) are added. Conversely, suppose that we can find a set theoretic model for the collection C"+(iii) of statements. Then we can define the predicates P(x) as 'x in S_P', and the predicates P(f(x)) as 'x in S_{Pf}'. To be sure that these predicates can derive from some model of these same predicates in which there do exist maps for which 'x in S_{Pf} *eq f(x) in S_P', we can argue as follows. In the assumed model M' of the sets S_P, any two sets of the form (i) will be disjoint if the pattern of intersections and complements defining them are different. Hence we can map the whole of each non-null set (i) into some selected point p of the (also non-null) set (ii). This plainly maps each set S_P into the set S_{Pf}, establishing that we do have a model of the original collection of quantified statements.

The following formula illustrates the technique just described:

```
(*)  ((FORALL x | (P(x) & P(f(x))) *imp P(f'(x))) &         (FORALL x | P(f(x)) *imp P(x)) &          (EXISTS
```

The negative of this is the conjoined collection of formulae

```
  (FORALL x | (P(x) & P(f(x))) *imp P(f'(x))),      (FORALL x | P(f(x)) *imp P(x)),       (EXISTS x | P(f(x))
```

The transformed set C' of formulae derived from this in the manner described above is

```
  (FORALL x | (P(x) & Pf(x)) *imp Pf'(x)),      (FORALL x | Pf(x) *imp P(x)),       (EXISTS x | Pf(x)), not(EX
```

If we now consider the predicate symbols to designate sets this gives

```
 pf' incs p * pf & p incs pf & pf /= {} & pf' = {}.
```

Here there appear two sets p_f and $p_{f'}$ derived from predicate terms involving function symbols, one for each of the function symbols f and f'. The additional conditions which need to be added to guarantee equisatisfiability are

```
  pf /= {} *imp p /= {}, U - pf /= {} *imp U - p /= {},  pf' /= {} *imp p /= {}, U - pf' /= {} *imp U - p /=
```

together with conditions stating that all other sets are included in U, so that U must designate the universe of any model. Since the conjunction of all these Boolean conditions is clearly unsatisfiable, formula (*) must be universally valid.

We can allow the use of both the MLSS constructs defined in the next section and of quantified predicates P(x), Q(y) of a single variable, under the very restrictive but easy-to-check condition that no quantified variable x can appear in any set-theoretic expression or relationship other than atomic expressions of the form

```
x = e      or   x in e     or   P(x)
```

where the expression e does not involve any quantified variable. As explained above, a nominal set p can be associated with each predicate P, and P(x) then written as x in p. The reductions described above apply easily to the somewhat generalized statements that result. Note that a quantified expressions like

```
(EXISTS x | x = e & x in p1 & ... & x in pm &     x notin pm + 1 & ... & x notin pn)
```

can be rewritten as

```
e in p1 & ... & e in pm & e notin pm + 1 & ... &     e notin pn,
```

while

```
(EXISTS x | (not (x = e)) & x in p1 & ... & x in pm &     x notin pm + 1 & ... & x notin pn)
```

can be rewritten as

```
(U - {e}) * p1 * ... * pm * (U - pm + 1) * ... * (U - pn) /= {},
```

so that removal of quantifiers in the manner explained always generates statements belonging to MLSS.

Certain limited classes of statements involving setformers reduce to the kinds of statements considered above. For example, the inclusion

```
{x in s | P(x)} incs {e(y): y in t | Q(y)}
```

can be written as

```
(FORALL y | (y in t & Q(y)) *imp (e(y) in s & P(e(y)))).
```

On the other hand, the converse inclusion

```
   {x in s | P(x)} *incin {e(y): y in t | Q(y)}
```

translates into

```
(FORALL x | (EXISTS y | (x in s & P(x)) *imp (x = e(y) & y in t & Q(y))))
```

which involves the binary equality operator and so is not covered by the preceding discussion.This indicates that statements involving setformers can only be handled by the method just described in particularly favorable cases.

MLSS: Multilevel syllogistic with singletons

MLSS is the (unquantified) extension of the elementary Boolean theory of sets obtained by allowing the membership relator 'x in y' and the singleton operator {x}in addition to the elementary operators and relators *, +, -, incs, and '='. Given a collection C of statements in this language, we begin as usual by applying decomposition at the propositional level, and then secondary decomposition. This allows us to assume that C consists of statements each having one of the

forms

```
s = t + u, s = t * u, s = t - u, s = t, not (s = t), s in t,          not(s in t), t = {s}.
```

We then eliminate all the statements s = t by selecting a representative of any group of set variables known to be equal, and replacing each occurrence of a variable in the group by its selected representative.

Next we prepare C for the analysis given below by enlarging it, but in a manner preserving satisfiability. This is done by collecting all the variables s which appear in statements of the form 's in t' or 't = {s}', along with their associated variables t. (We will call these s the *left-hand* variables). Then, for each pair s_1, s_2 of such variables, of which s_1 appears in a statement '$t_1 = \{s_1\}$' and s_2 appears in a statement '$t_2 = \{s_2\}$' or in a statement 's_2 in t_2', we add an implication

```
(t1 = t2) *imp (s1 = s2).
```

Since this last statement evidently follows both from the pair of statements

```
t1 = {s1}, t2 = {s2}
```

and from the pair of statements $t_1 = \{s_1\}$, s_2 in t_2, these additions evidently preserve satisfiability. We also add statements

```
s1 = s2 or not(s1 = s2)
```

for each pair of lefthand variables. After adding the indicated statements, we apply decomposition at the propositional level once more, and again eliminate all statements s = t by selecting representatives in the manner described above. This leaves us with a modified collection C of statements, each having one of the forms

```
(+)   s = t + u, s = t * u, s = t - u, not (s = t), s in t,          not(s in t), t = {s}.
```

But now, after the steps of preparation we have described, we can be sure that for any two distinct left-hand variables s_1 and s_2, an explicit inequality 'not ($s_1 = s_2$)' is present in C.

Now suppose our collection C of statements has a model M with universe U. As in our previous discussion of the elementary Boolean case the set of places p_a defined by $p_a(x)$ *eq a in M(x), where a ranges over the points of U, must be ample for the subcollection of elementary Boolean statements in C, namely those not of the form s in t, not(s in t), or t = {s}. The points M(s) corresponding to the variables s appearing in C define places p_s (via our standard formula $p_s(x)$ *eq M(s) in M(x)), which plainly must have the following properties

```
ps(t) is true if a statement 's in t' appears in C;  ps(t) is false if a statement 'not(s in t)' appears i
```

We call a place p_s having these three properties a *place at s.* Some of the places corresponding to points in the model M will be places at s for some variable s in the set C of statements, others will not.

We now look a bit more closely at the structure of the model M, with an eye toward accumulating enough properties of its places to guarantee the existence of at least one model. Note first of all that since set theory forbids all cycles

```
x1 in x2 in ... xn in x1
```

of membership, it must be possible to arrange the sets M(x) of our model into an order for which the variable x comes before y whenever M(x) is a member of M(y). We will call any such order an *acceptable ordering* of the variables of C. Note that for any acceptable ordering, and any variables s and t, $p_s(t)$ can only be true if s precedes t in this ordering.

For each point p of the model we can let M_p be the collection of all points q of the model such that (p in M(s)) *eq (q in M(s)) for every variable s appearing in a statement of C , minus all points having the form M(s) for some left-hand variable s. This allows us to write each set M(s) of the model in the following way for each variable s in the set Lvars of all left-hand variables appearing in C:

```
(*)  M(s) = {M(x): x in Lvars | px(s)} + Un({Mp: p in places | p(s)}).
```

The sets M_p are clearly disjoint for distinct p, i.e. $M_p * M_q = \{\}$ if p /= q. If a statement 't = {s}' appears in C, then M(t) must be a singleton, so that p_s must be the only place p of the model M for which p(t) is true, and also M_p must be null.

The following theorem shows that the conditions on the collection of places of M that we have just enumerated are sufficient to guarantee the existence of a model of C, and so gives us a procedure for determining the satisfiability of C.

Theorem: Let C be a collection of statements of the form (+), and suppose that if s_1, s_2 are two distinct variables appearing in C, and that s_1 and s_2 are two distinct left-hand variables of C, an inequality 'not ($s_1 = s_2$)' is present in C.

Then the following conditions are necessary and sufficient for C to be satisfiable, i.e. to have a model M:

> (i) There exists an ample set A of places p for the subcollection of elementary Boolean statements in C.
>
> (ii) For each variable s appearing in a statement of C, there is a place p_s at s in A. Moreover, the variables appearing in the statements of C can be arranged in an order O such that $p_s(t)$ is false unless t precedes s in this order.
>
> (iii) If a statement 't = {s}' appears in C, then p_s is the only place in A for which p(t) is true.

Proof: We saw above that the conditions (i-iii) are necessary. Suppose conversely that they are satisfied. For each place p in A choose a set M_p in such a way that all these sets are disjoint and non-null; however if a statement 't = {s}' appears in C (so that s is a left-hand variable) we take M_{p_s} to be null. We also suppose that each member of M_p has larger cardinality than the total number V of variables appearing in C, plus #A * K, where K is the largest cardinality of any set M_p. (One way of doing this is to let the non-null sets M_p be distinct singletons {u}, where each u has a number of members exceeding V + #A). Then use formula (*) to define M(s) for each variable s appearing in C. This is possible since the condition (ii) can be arranged in an order for which all the M(x) appearing in the definition (*) of M(s) have been defined before (*) is used to define M(s). Note that the cardinality condition we have imposed ensures that every one of the sets

```
{M(x): x in Lvars | px(s)}
```

appearing first on the right of any formula (*) is disjoint from every one of the sets

```
Un({Mp: p in places | p(s)}),
```

appearing second on the right of any formula (*), every set M(s) has cardinality at most V + #A*K, while all the members of a set Un({Mp: p in places | p(s)}) must be members of some M_p, and hence must have cardinality greater than V + #A*K.

We now show that all the statements 'not (s = t)' are correctly modeled by the function M defined by (*). This is clear if there exists any M_p /= {} for which p(s) and p(t) are different, since in this case it follows from (*) that M(p) will be a subset of M(s) and will be disjoint from M(t) (since the first and second terms of (*) are always disjoint). But we must prove it in general.

Suppose that our claim is false, and let s be the first variable, in the ordering O mentioned in condition (ii), for which there exists some statement 'not (s = t)' in C such that M(s) = M(t). Since the set A of places is ample, there must exist a place p in A such that one of p(s), p(t) is true and the other is false. Suppose for definiteness that p(s) is true, so p(t) is false. If M_p were nonempty the second term of (*) would be distinct from the second term of

```
(**)   M(t) = {M(x): x in Lvars | px(s)} + Un({Mp : p in places | p(t)}),
```

and since all these first and second terms are disjoint it would follow tat M(s) /= M(t), contrary to assumption. Hence M(p) = { }, so that p must be of the form $p = p_u$, where u is some left-hand variable. Then $p_u(s)$ is true, so M(u) belongs to M(s) by (*). Hence M(u) belongs to M(t) also. But M(u) cannot belong to the second term of (**), since if it did it would belong to some M_p, and all the members of all M_p have cardinality larger than any M(u). Therefore M(u) must belong to the first term of M(t), i.e. must be identical with some M(v) for which $p_v(t)$ is true. Both u and v must be left-hand variables, and so if they are distinct C must contain a clause 'u /= v'. But now M(u) /= M(v) contradicts our assumption that s is the first variable in the order O for which there exists a t such that M(s) = M(t). This contradiction proves our claim that M(s) /= M(t) whenever a clause 'not(s=t)' is present in C, and so shows that all such clauses are correctly modeled by M.

Next we show that all other statements of C are correctly modeled also. For statements t = {s} this follows immediately from condition (iii) of our theorem and the fact that $M(p_s) = \{\}$ for each variable s appearing in such a statement. Statements 't in s' are correctly modeled since the presence of such a statement implies that M(t) must belong to the first term of (*). Statements 'not (t in s)' are correctly modeled, since by its cardinality a set of the form M(t) can only belong to the first term of (*); but since all the M(t) are distinct for distinct left-hand variables, M(t) will only belong to the first term of (*) if $p_t(s)$ is true, which is impossible if 'not(t in s)' appears in C.

Statements s = t + u are correctly modeled since

```
M(s) = {M(x): x in Lvars | px(s)} +          Un({M(p): p in places | p(s)})       = {M(x): x in Lvars |
```

Similarly, for statements s = t * u we have

```
M(t * u) = {M(x): x in Lvars | px(t) & px(u)} +          Un({M(p): p in places | p(t) & p(u)})       = ({M
```

since all the sets M(p) are disjoint, no M(x) belongs to any of them, and all the sets M(x) for x in Lvars are distinct. The same argument handles the case of statements 's = t - u', completing the proof of our theorem. **QED.**

MLSS plus the predicates 'Finite' and 'Countable'

We can easily generalize MLSS by allowing the two additional set predicates Finite(s) and Countable(s) studied above. Much as before, we can introduce two new variables Fi and Co, and for these variables introduce the following statements:

```
(**)  Co incs Fi.  For each x for which a statement Finite(x) is present,     a statement Fi incs x.  For e
```

Then drop all statements of the form Finite(x), Countable(x), not Finite(x), not Countable(x).

It is plain from what was said above that if our original collection of statements has a model, so does our modified collection. Conversely, if this modified collection has a model, then as above there must exist an ample set of places and to these places we can assign disjoint sets M_p according to the following rule:

```
if p is of the form ps for some variable s        appearing in a statement t = {s}, let Mp be null;  other
```

We also suppose, as in the preceding discussion of MLSS, that each member of M_p has larger cardinality than V+#A*K, where V, A, and K are as in that discussion, and then use (*) to define a model M. The analysis given in the preceding section shows that this M correctly models all statements not involving the predicates 'Finite' and 'Countable'. It is plain that M(Fi) is finite and M(Co) is countable; hence all statements Finite(x) and Countable(x) originally present are correctly modeled also.

If any statement 'not Finite(x)' is present in C, then there exists a p such that p(x) is true and p(Fi) is false. p cannot have the form p_s for any variable s appearing in any statement 't = {s}' appearing in C, since if it did then the fact that p_s(t) must be true and the statement 'Fi incs t' added to C would imply that p_s(Fi) is true. Hence M_p is infinite and so by (*) M(x) is infinite also. This shows that all statements 'not Finite(x)' are correctly modeled. The case of statements 'not Countable(x)' can be handled in much the same way, showing that our original and modified sets of statements are equisatisfiable.

Elementary Booleans plus map primitives

Next we consider another unquantified generalization of the elementary Boolean language of sets with which we started. This introduces variables designating maps between sets, which to ensure decidability we treat here as objects of a kind different from sets, designated by variables of a syntactically different, recognizable kind. (For convenience we will write set variables as letters s,t etc. taken from the initial part of the alphabet, and designate maps by letters like f, g from the later part of the alphabet). In addition to the elementary Booleans operators and comparators, the unquantified language we now wish to consider allows the map primitives

```
range(f) = s, domain(f) = s, f | s = g (map restriction),   Svm(f) (f is a single-valued map),   and Sinc
```

We will show that this language is decidable by reducing collections of statements in it to equisatisfiable collections of statements in which all variables designating maps, and all map-related operations, have been removed. As usual, we begin by applying decomposition at the propositional level, and then secondary decomposition, to the collection of statements originally given us. This means that we have only to deal with collections of statements each having one of the allowed elementary forms s = t + u, s = t, not (s = t), range(f) = s, f | s = g, Svm(f), Singinv(f), f = g, not (f = g), etc. Now we proceed as follows.

(i) All equalities between sets or between maps are removed by selecting a representative of any group of set or map variables known to be equal, and replacing each occurrence of a variable in the group by its selected representative.

(ii) We replace each statement not(f = g) by a statement of the form

```
not ((range(f | s_new) = range(g | s_new)).
```

This reflects the fact that if two maps are different, there must exist a set s on which their ranges are different. (For example, this can be a singleton whose one member either belongs to the domain of one of the maps but not the other, or to both domains, but at which the functions have different values).

(ii) All the map-related statements which remain at the end of step (ii) have one of the forms range(f) = s, domain(f) = s, f | s = g, Svm(f), and Singinv(f). We now proceed in the following way to eliminate all statements of the form (f | s) = g. We enumerate all the sets $s_1,...,s_k$ which appear in statements of the form f | s_j = g, and form the collection of all their 'Venn pieces'. These 'Venn pieces' are newly introduced symbols $V_{i_1,...,i_k}$ for all intersections of the sets s_j or their complements, with the obvious relationships defining the $V_{i_1,...,i_k}$ in terms of s_j and vice-versa. More specifically, the subscripts $i_1,...,i_k$ of the Venn pieces are all possible sequences of 0's and 1's of length k, distinct Venn pieces are disjoint, and each s_j is the union of all the Venn pieces

$V_{i1,\ldots,i_{j-1},1,i_{j+1},\ldots,i_k}$.

(iii) Next we introduce the 'Venn pieces' of the maps f. These are symbols f_i for all restrictions f | V_i, which we introduce with symbols r_i and d_i for their ranges and domains respectively, and statements expressing each f | s_j in terms of these r_i and d_i. Moreover, we add all relationships f_i = g expressing all the initial relationships f | s_j = g and the statements r_i /= { } *eq d_i /= { }, for each symbol f_i .

This eliminates all statements of the form f | s = g, leaving only simple equalities f = g, which can be eliminated by closing them transitively and choosing a representative of each class. Then only statements range(f) = s and domain(f) = s remain. Drop these, keeping only the corresponding

r_i /= { } *eq d_i /= { }, getting a set S' of elementary Boolean statements.

If S' has a model so does the original S (ignoring statements Svm(f) and Singinv(f)) since we can construct the f_i as either single-valued or non-single-valued maps of each non-null r_i onto the corresponding d_i, making all these sets countable.

To model a collection of statements Svm(f) and (not Svm(f)) we need only assign a truth value to each condition Svm(f_i), insisting that Svm(f) be equivalent to the disjunction of all the statements Svm(f_i), extended over all the Venn pieces of f.

To model a collection of statements Singinv(f) and (not Singinv(f)) we must add conditions r_i * r_j = { } for all the distinct pieces r_i into which each original range(f) is decomposed, since then the union map of the Venn pieces f_i of f can have a single-valued inverse or not, as desired. We must also assign a truth value to each condition Svm(f_i), and insist that Svm(f) be equivalent to the disjunction of all the statements Svm(f_i), extended over all the Venn pieces of f.

Various commonly occurring decidable extensions of MLSS

The decision algorithm for MLSS presented above can be extended in useful ways by allowing otherwise uninterpreted function symbols subject to certain universally quantified statements to be intermixed with the other operators of MLSS. Note however that the statements decided by the method to be described remain unquantified; the quantified statements to which we refer appear only as implicit 'side conditions'.

The 'pairing' operator 'cons' and the two associated component extraction operators 'car' and 'cdr' exemplify the operator families to which our extension technique is applicable. As noted earlier, these operators can be given formal set-theoretic definitions:

```
cons(x,y) := {{x},{{x},{{y},y}}},   car(p) := arb(arb(p)),  cdr(p) := arb(arb(arb(p - {arb(p)}) - {arb(p)
```

However, in most settings, the details of these definitions are irrelevant. Only the following properties of these operators matter:

The object cons(x,y) can be formed for any two sets x,y.

Both of the sets x,y from which cons(x,y) is formed can be recovered uniquely from the single object cons(x,y), since car(cons(x,y)) = x and cdr(cons(x,y)) = y.

Almost all proofs in which the operators 'cons', 'car', and 'cdr' appear use only these facts about this triple of operators. That is, they implicitly treat these operators as a family of three otherwise uninterpreted operators, subject only to the conditions

```
(FORALL x,y | car(cons(x,y)) = x) &       (FORALL x,y | cdr(cons(x,y)) = y).
```

The treatment of 'cons', 'car', and 'cdr' throws away information about these operators (e.g. cons(x,y) has cardinality 2 and car(x) is always a member of a member of x) that may become relevant in unusual situations, but this very rarely makes any difference.

Even though the underlying definitions are not always so strongly irrelevant as in the case of 'cons', 'car', and 'cdr', similar remarks apply to many other important families of operators. We list some of these, along with the universally quantified statements associated with them:

(i) *arb*:

```
(FORALL x | (x = {} & arb(x) = {})    or (arb(x) in x & arb(x) * x = {})) ;
```

(ii) pairs of *mutually inverse functions* on a set w:

```
(FORALL x in w | f(x) in w & g(x) in w    & f(g(x)) = x & g(f(x)) = x) ;
```

(iii) *monotone functions*:

```
(FORALL x,y | (x incs y) *imp (f(x) incs f(y))) ;
```

(iv) monotone functions having a known order relationship:

```
(FORALL x,y | (x incs y) *imp (f(x) incs f(y))) & (FORALL x,y | (x incs y) *imp (g(x) incs g(y
```

(v) monotone functions of several variables:

```
(FORALL x,y,u,v |      (x incs y & u incs v)          *imp (f(x,u) incs f(y,v))) ;
```

(vi) *idempotent* functions on a set w:

```
(FORALL x in w | f(x) in w & f(f(x)) = f(x)) ;
```

(vii) *self-inverse* functions on a set:

```
(FORALL x in w | f(x) in w & f(f(x)) = x) ;
```

(viii) *total ordering relationships* on a set:

```
(FORALL x in w, y in w | (R(x,y) or R(y,x)) & R(x,x)) &     (FORALL x in w, y in w, z in w |
```

(ix) (multiple) functions with *known ranges* v_j *and domains* w_j:

```
(FORALL x in vj | fj(x) in wj) ,
```

for multiple indices j and k.

These are all mathematically significant relationships, as the existence of names associated with them attests.

These cases can all be handled by a common method under the following conditions. Suppose that we are given an unquantified collection C of statements involving the operators of MLSS plus certain other function symbols f,g of various numbers of arguments. After decomposing compound terms in the manner described earlier, we can suppose that all occurrences of these additional symbols are in simple statements of forms like y = f(x), y = g(x,z), etc. From these

initially given statements we must be able to draw a 'complete' collection S of consequences, involving the variables which appear in them, along with some finite number of additional variables that it may be necessary to introduce. The resulting collection of formulae, comprising S and some 'residue' of the original C, will be entirely within the language of MLSS. 'Completeness' means that any model of the translated formula can be extended to include the original function symbols f, g, etc. in such a way that their interpretation Model(f), Model(g), etc. actually satisfies the desired properties (monotonicity, etc.).

In all cases listed above, S will include at least *single-valuedness conditions* x = u *imp y = v for all pairs y = f(x), v = f(u) originally present in C, so S will consist of these statements plus others appropriate to the case being considered, as detailed below. Call these added statements S the *extension conditions* for the given set of functions. We must find extension conditions comprising S which encapsulate everything which the appearance of the functions in question tells us about the set variables which also appear.

If extension conditions can be found, satisfiability can be determined by replacing all the statements y = f(x), y = g(x,z) in our original collection by the extension conditions derived from them.

This gives us a systematic way of reducing various languages extending MLSS to pure MLSS. As we will see, this approach can be exploited, to some extent, with predicates too, thanks to the fact that certain properties of predicates can be represented using associated functions.

Note that this 'extension conditions' technique can be applied even if the recipe for removing universal quantifiers by adding compensating extension clauses is not complete, as long as it is sound, i.e. all the clauses added do follow from known properties of the functions or predicates removed.

Take Case (iii) above (the 'monotone functions' case) as an example. Here the extension conditions can be derived as follows. Let the function symbols known to designate monotone functions be f, etc. Replace all the statements y = f(x), v = f(u) originally present by statements

```
(*)  x incs u *imp y incs v.
```

(Note that this implies the single-valuedness condition for f). The added clauses ensure that if a model exists, the set of pairs [Model(x),Model(y)], formed for all the x and y initially appearing in clauses y = f(x), defines a function F which is monotone on its domain. This can be extended to a function F' defined everywhere by defining F'(s) as the union of all the F(t), extended over all the elements t of the domain of F for which s incs t. It is clear that the F' defined in this way is also monotone and extends F. This proves that the clauses (*) express the proper extension condition in Case (iii). Note that the number of clauses (*) required is roughly as large as the square of the number of clauses y = f(x) originally present.

To make this method of proof entirely clear we give an example. Suppose that we need to prove the implication

```
(+)  f(f(x + y)) incs f(f(x))
```

under the assumption that the function f is monotone. By decomposing the compound terms which appear in this statement, we get the collection

```
  z = x + y, u = f(z), w = f(u), u' = f(x),       v' = f(u'), not(w incs v'),
```

which we must prove to be unsatisfiable. The four statements u = f(z), w = f(u), u' = f(x), v' = f(u') in this collection give rise to the 12 extension conditions

```
  (z incs u) *imp (u incs w), (z incs x) *imp (u incs u'),   (z incs u') *imp (u incs v'), (u incs z) *imp
```

which replace the four initial statements. It now becomes possible to see that

```
z = x + y, (z incs x) *imp (u incs u'),   (u incs u') *imp (w incs v'), not(w incs v')
```

is an unsatisfiable conjunction, proving the validity of (+).

Extension conditions in the other cases listed above. We shall now describe the extension conditions applicable in the remaining cases listed above. In Case (i) (the 'arb' case) the extension conditions are simply

```
(**)    (x = {} & arb(x) = {}) or (arb(x) in x & arb(x) * x = {})            & (x = u *imp arb(x) = arb(u))
```

(This last clause is the condition of 'single-valued functional dependence'). Suppose now that we model a collection of MLSS clauses, plus statements of the form x = arb(y), after first replacing all the y = arb(x), v = arb(u) originally given by the derived clauses

```
(x = {} & y = {}) or (y in x & y * x = {}) & (x = u *imp y = v)).
```

Then plainly the set of pairs [Model(x),Model(y)], formed for all the x and y appearing in the statements 'y = arb(x)' originally present, defines a single-valued function A on its finite domain which satisfies

```
(s = {} & A(s) = {}) or (A(s) in s & A(s) * s = {}),
```

for all the elements of its domain. We can extend this to a function A' defined everywhere by writing

```
A'(s) = if s in domain(A) then A(s) else arb(s) end if,
```

where 'arb' is the built-in choice operator of our version of set theory. A' then satisfies the originally universally quantified condition for arb, verifying our claim that the clauses (**) are the proper extension conditions.

Case (iv) (monotone functions having a known order relationship) can be treated in much the same way as the somewhat simpler case (iii) discussed above. Given two such f, g, where it is known that f(x) incs g(x) is universally true, first force the known part of their domains to be equal by introducing a u satisfying g(x) = u for each initially given clause f(x) = y and vice-versa. Then proceed as in case (iii), but now add inclusions

```
x = v *imp y incs u
```

for every pair g(v) = u, f(x) = y of clauses present. It is clear that the extensions of g and f defined in our discussion of the simpler case (iii) stand in the proper ordering relationship.

Case (v) (monotone functions of several variables) is also easy. We can proceed as follows. Given a function f(x,y) which is to be monotone in both its variables, and also a set of clauses like z = f(x,y), w = f(u,v), introduce clauses

```
(x incs u & y incs v) *imp (z incs w).
```

Then plainly the set of pairs [[Model(x),Model(y)],Model(z)], formed for all the x,y,z initially appearing in clauses z = f(x,y) defines a function F of two arguments which is monotone on its domain. This can be extended to a function F' defined everywhere by defining F'(s,t) as the union of all the F(p,q), extended over all the pairs p, q of the domain of F for which s incs p and t incs q.

The related case of additive functions of a set variable can also be treated in the way which we will now explain (but the very many clauses which this technique introduces hints that 'additivity' is a significantly harder case than 'monotonicity'). A set-valued function f of sets is called 'additive' if f(x + y) = f(x) + f(y) for all x and y. Given an otherwise uninterpreted

function f which is supposed to be additive, and clauses y = f(x), introduce all the 'atomic parts' of all the variables x which appear in such clauses. These are variables representing all the intersections of some of these sets x with the complements of the other sets x. In terms of these intersections, which clearly are all disjoint, express each x in terms of its atomic parts, namely as '$x=a_{j_1}+...+a_{j_k}$'. Likewise, after introducing clauses $b_j = f(a_j)$ giving names to the range elements $f(a_j)$, write out all the relationships '$y = b_{j_1}+...+b_{j_k}$' that derive from clauses y = f(x). Finally, writting {} and f({}) for uniformity as a_0 and b_0, add statements '$a_j = a_0$ *imp $b_j = b_0$' and 'b_0 *incin b_j', along with statements

```
aj * ai = {}       (with i /= j)
```

which express the disjointness of distinct sets a_j. Now suppose that the set of clauses we have written has a model in which the a_j, b_j, x, y, etc. appearing above are represented by sets a'_j, b'_j, x', y', etc. and for each s, define the set-valued function F(s) to be the union of all the sets b'_j for which s intersects a'_j. The function F defined in this way is clearly additive. It is also clear that if a clause y = f(x) is present in our initial collection, and the variables x and y are represented by sets x' and y', then y' = F(x'). Hence F can represent f in the model we have constructed, so f can be represented by an additive function, proving that the clauses we have added to our original collection are the appropriate extension conditions.

Cases (vi) (idempotent functions on a set) and (vii) (self-inverse functions on a set) are also easy. In the case of idempotent function we can proceed as before, but adding a clause y = f(y) whenever a clause y = f(x) is present. Then we add implications

```
w = x *imp z = y
```

whenever two clauses y = f(x), z = f(w) are present, and remove all the clauses y = f(x). The added clauses ensure that if a model M exists, the mapping F which sends Mx to My for each clause initially present is single-valued, and since a clause y = f(y) has been added whenever a clause y = f(x) is present this mapping is clearly idempotent where defined. It can be extended by mapping all elements not in the domain of F to any selected element of the range of F.

The self-inverse function case (vii) can be handled in much the same way. Here one adds a clause x = f(y) whenever the clause y = f(x) is present, and then adds all the implications needed to force a model of the pairs [x,y] deriving from clauses y = f(x) initially present to define a single-valued map which can model the original f. In the resulting model f is self-inverse on its domain, which is the same as its range. f can then be extended to a mapping defined for all x by writing f(x) = x for all elements not in its domain/range.

Predicates representable by functions in one of the classes analyzed above can be removed automatically by first replacing them by the functions that represent them, and then removing these functions by writing the appropriate extension conditions. For example, equivalence relationships R(x,y) can be written as f(x) = f(y) using a representing function f; f only needs to be single-valued. Partial ordering relationships can be written as f(x) incs f(y) where f only needs to be single-valued. f is monotone iff the ordering relationship R(x,y) is compatible with inclusion, in the sense that

```
(FORALL x,y | (x incs y) *imp R(x,y)).
```

Monadic predicates P(x) satisfying the condition

```
(FORALL x,y | (P(x) & P(y)) *imp P(x + y)) &     (FORALL x,y | (P(x) & x incs y) *imp P(y))
```

can be written in the form P(x) *eq (p incs x). The predicates Finite(x), Countable(x), and Is_map(x) illustrate this remark.

Case (viii) (total ordering relationships on a set) can be handled in the following way, which derives from the preceding remarks. Let R be such a relationship. Introduce a representing function f for it, i.e. f(x) incs f(y) *eq R(x,y). Then R is a total ordering iff the range elements f(x) all belong to a collection of sets totally ordered by inclusion. So write a clause 'y

incs v or v incs y' for each pair of clauses y = f(x), v = f(u), and also write the conditions needed to ensure that f is single-valued. In the resulting model f plainly maps its domain into a collection of sets totally ordered by inclusion, and then f can be extended to all other sets by sending them to {}.

Case (ix) (multiple functions with known ranges and domains) is also very easy. For clarity, we will consider the special subcase of this in which two functions f, g are given, along with two domain sets d_1, d_2, and two range sets r_1, r_2. The universally quantified conditions which must be satisfied are

```
(a)    (FORALL x in d1 | f(x) in r1)(b)    (FORALL x in d2 | g(x) in r2),
```

along with some collection of unquantified clauses of MLSS.

We proceed as follows. For any two clauses y=f(x), y'=f(x') present in our set S of clauses write a condition

```
(*)    x = x' *imp y' = y',
```

and similarly for g. As usual, these reflect the single-valuedness of f and g. For any clause y=f(x) in S, write a condition

```
(**)    x in d1 *imp y in r1,
```

and similarly for g, d_2, and r_2. Finally, write the conditions

```
        d1 /= {} *imp r1 /= {},(***)         d2 /= {} *imp r2 /= {}.
```

Then seek a model of the resulting set C of clauses, which must plainly exist if our original set of clauses is consistent.

Conversely, suppose that the clauses C have a model M. Define a preliminary function F (resp. G) as the set of all pairs [M(x),M(y)] for which a clause y=f(x) (resp. y=g(x)) is present in S. The clauses (*) plainly imply that F is single-valued on its domain, and the clauses (**) ensure that F maps the intersection of its domain with d_1 into r_1. If $M(d_1)=\{\}$ the quantified condition (a) is automatically satisfied. If $M(d_1)/=\{\}$, the clause (***) ensures that $Mr_1/=\{\}$, so we can extend F to map all elements of d_1 not in its initial domain to any element of r_1 we choose. Repeating this construction for g, d_2, and r_2 plainly gives us a model of all our clauses in which f and g are represented by single-valued functions satisfying (a) and (b). Hence the clauses (*), (**), and (***) we have added are the extension conditions we require.

The case of mutually inverse functions. Extension conditions for Case (ii) (pairs of mutually inverse functions f, g on a set w) can be formulated as follows. Write the clauses, described above, that force f and g to be single-valued. To these, add clauses

```
y = v *imp x = u
```

derived from all the given statements y = f(x), v = f(u). These force f to be 1-1 on the collection of elements x known to be in its domain. (Note that this much also handles the case of functions known to be 1-1). Do the same thing for g. Then add clauses

```
y = u *eq x = v
```

derived from all the statement pairs y = f(x), v = g(u). Then, in the resulting model M, the model functions F and G of f and g must both be 1-1 on their domains (e.g. for F this is the collection of sets M(x) modeling points x for which some clause y=f(x) appears in our original set of statements), and G must be the inverse of F on domain(G) * range(F). Since G is 1-1 on its domain, it follows that the range of G on domain(G) - range(F) must be disjoint from domain(F). Indeed, if a set s is in domain(F) * range(G) it must have the form s=M(x) where clauses y=f(x) and x=g(u) both appear in our original set of statements. But then M(u)=M(y) is implied by an added clause, and hence M(u) is in the range of F. Similarly the

range of F on domain(F) - range(G) must be disjoint from domain(G). F can therefore be extended to

```
range(G | domain(G) - range(F)) (the range on the restriction)
```

as the inverse of G, and similarly G extended to

```
range(F | domain(F) - range(G))
```

as the inverse of F. Let F' and G' be these extensions. Then plainly domain(F') = domain(F) + range(G), and so range(G') = range(G) + domain(F) = domain(F') and vice-versa. Hence the extensions F' and G' are mutually inverse with domain(F') = range(G') and vice-versa. F' and G' can now be extended to mutually inverse maps defined everywhere by using any 1-1 map of the complement of domain(F') onto the complement of range(F'). This shows that the clauses listed above are the correct extension conditions for case (ii).

The extension conditions for the important car, cdr, and cons case can be worked out in similar fashion as follows. Regard cons(x,y) as a family of one-parameter functions $cons_x(y)$ dependent on the subsidiary parameter x. The ranges of all the functions $cons_x$ in the family are disjoint (since cons(x,y) can never equal cons(u,v) if x /= u). For the same reason, each $cons_x$ is 1-1, and cdr is its (left) inverse, i.e. $cdr(cons_x(y)) = y$. Also, $car(cons_x(y)) = x$ everywhere. The extension conditions needed can then be stated as follows:

(i) 'cons' must be 'doubly 1-1' and well defined: add clauses

```
(not ((x = u) & (y = v))) *imp (not (z = w))
```

and

```
((x = u) & (y = v)) *imp (z = w)
```

derived from all pairs of initial clauses z = cons(x,y), w = cons(u,v).

(ii) car and cdr must stand in the proper inverse relationship to cons: add clauses

```
u = z *imp x = v
```

derived from all pairs z = cons(x,y), v = car(u), and all clauses

```
u = z *imp y = v
```

derived from all pairs z = cons(x,y), v = cdr(u) of initial statements.

Various other cases which can be handled by the 'extension conditions' technique, e.g. uninterpreted commutative functions of two variables, having the property

```
(FORALL x,y | f(x,y) = f(y,x)),
```

can readily be handled by this technique. It might be possible to treat associativity also, possibly based on a prior MLSS-like theory of the concatenation operator.

Because of their special importance the treatment of 'arb' and of the 'cons-car-cdr' group is built into ELEM. The use of supplementary proof mechanisms for handling other extended ELEM deductions like those described above is switched on in the following way. Each of the cases listed above is given a name, specifically (ii) INVERSE_PAIR, (iii) MONOTONE_FCN, (iv) MONOTONE_GROUP, (v) MONOTONE_MULTIVAR, (vi) IDEMPOTENT, (vii)

SELF_INVERSE, (viii) TOTAL_ORDERING, (ix) RANGE_AND_DOMAIN. To enable the use of supplementary inferencing for a particular operator belonging to one of these named classes, one writes a verifier command of a form like

```
ENABLE_ELEM(class_name; operator_list)
```

where class_name is one of the names in the preceding list, and operator_list lists the operator symbols for which the designated style of inferencing is to be applied. An example is

```
ENABLE_ELEM(MONOTONE_FCN; Un)
```

which states that during ELEM inferencing the 'union of elements' operator Un is to be treated as an otherwise uninterpreted symbol for a monotone increasing set operator. The operator_list parameter of an 'ENABLE_ELEM' command must consist of the number of operators appropriate to the class_name used, e.g. IDEMPOTENT calls for a single operator as its operator list but MONOTONE_GROUP and INVERSE_PAIR each call for a list two operators f,g.

The ENABLE_ELEM command scans the list of all currently available theorems for theorems of form suitable to the type of inference defined by the class_name parameter. For example, MONOTONE_FCN calls for a theorem of the form

```
(FORALL x,y | (x incs y) *imp (f(x) incs f(y)))
```

where f is the function symbol that appears as operator_list in this case; IDEMPOTENT calls for a theorem of the form

```
(FORALL x,y | f(f(x)) = f(x))).
```

Thus, for example, the command ENABLE_ELEM(MONOTONE_FCN; Un) calls for the theorem

```
(FORALL x,y | (x incs y) *imp (Un(x) incs Un(y))).
```

Cardinality is another example; the command ENABLE_ELEM(MONOTONE_FCN; #) calls for the theorem

```
(FORALL x,y | (x incs y) *imp (#x incs #y)).
```

If the required theorem is not found an error message is issued; otherwise the declared style of inferencing becomes available for the operator or operators listed.

Since extension of ELEM inferencing is not without its efficiency costs, one may wish to switch it on and off selectively. To switch off extended ELEM inferencing of a specified kind for specified operators one uses a command

```
DISABLE_ELEM(class_name, operator_list)
```

whose class_name parameter must reference one of the names which could occur in an ENABLE_ELEM(class_name;...) directive. This disables use of the ELEM extensions described above for the indicated operators. Of course, a subsequent ENABLE_ELEM command can switch this back on.

Limited predicate proof

In some situations, we can combine the ELEM style of unquantified proof described in the preceding pages with predicate reasoning, provided that we hold down the computational cost of proof searches by imposing artificial limitations on the information used. An example of such a situation is that in which a deduction is to be made by combining a collection of statements in the unquantified language of MLSS with one or more universally quantified statements like

```
(FORALL s,t | (Ord(s) & Ord(t)) *imp       (s in t or t in s or s = t),
```

where 'Ord(s)' is the predicate stating that s is an ordinal. Although in the full context of set theory use of such statements opens a path to very many subsequent deductions, and so has consequences that are quite undecidable, the special case of universally quantified statements which contain no symbols designating operators and only uninterpreted predicates is more tractable. This limited case can be handled in the following way. Suppose that we deal with a collection C of unquantified statements of the language MLSS, together with a collection U of universally quantified statement of the form

```
(+)  (FORALL x1,...,xn | P),
```

where P is built from some collection of uninterpreted predicates $Q(x_1,...,x_n)$ and contains no function symbols. Gather all the variables s that appear in the statements of C, substitute them in all possible ways for the bound variables of (+), and decompose the resulting collection of statements at the propositional level. To the original collection C this would add a finite number of statements of the form

```
    Q(s1,...,sn),
```

some of which may be negated. But instead of adding these statements, which involve predicate constructions, proceed as follows. For each such $Q(s_1,...,s_n)$ introduce a unique propositional symbol $Q^{s^1,...,s^n}$ and add $Q^{s^1,...,s^n}$, negated in the pattern inherited from the $Q(s_1,...,s_n)$ instead of the $Q(s_1,...,s_n)$ to C. Then, for all pairs of argument tuples $s_1,...,s_n$ and $t_1,...,t_n$ which appear in such statements (with the same Q) add an implication

```
(++)  (s1 = t1 & ... & sn = tn) *imp (Qs1,...,sn = Qt1,...,tn).
```

This gives a collection C' of statements, all of which are in MLSS. It is clear that C' is satisfiable if C and U are simultaneously satisfiable. Conversely, let C' have a model M. The conditions (++) that we have added to C imply that the Boolean values $Q^{s^1,...,s^n}$ derive from a single -valued predicate function via the relationship

```
   Qs1,...,sn = Q(s1,...,sn).
```

Let D be the collection of all the elements of the model M that correspond to symbols which appear in statements belonging to C. Then plainly

```
(+++)  (FORALL x1 in D,...,xn in D | P).
```

Choose some s_0 in D and let r be the idempotent map of the entire universe of sets onto D defined by

```
  r(x) = if x in D then x else s0 end if.
```

If we show the dependence of the predicate P on its free variables $x_1,...,x_n$ by writing it as $P(x_1,...,x_n)$, then (+++) is clearly equivalent to

```
(*)  (FORALL x1,...,xn | P(r(x1),...,r(xn))).
```

Extend each of the predicates Q^M from its restriction to the Cartesian product D * D *...* D to a universally defined predicate Q_+^M by taking

```
   Q+M(x1,...,xn) = Q(r(x1),...,r(xn)).
```

Then it is clear that the predicates Q_+^M model both the statements of C and the universally quantified statement (+). This shows that the collection C' has a model if and only if the union of C and U has a model, proving that the satisfiability of C + U is decidable.

Given any collection of universally quantified statements U and collection C of unquantified statements of MLSS, we can treat them as if the predicates appearing in the statements of U were uninterpreted, i.e. had no known properties except those given explicitly by the statements in U. Even though this throws away a great deal of information that can be quite useful, there are many situations in which it achieves an inference step needed for a particular argument. Note that the inference mechanism described need not treat predicates like 'x in y' and 'x incs y' present in a universally quantified statement as uninterpreted predicates if they contain no operator signs not available in MLSS, even though the preceeding argument fails if this is not done: the inference method used remains sound nevertheless. However compounds like '#t *incin #s' must be treated as uninterpreted multiparameter predicates, just as if they read $Q_{\#}(s,t)$. Similarly a compound like 'Finite(domain(f))' must be treated as if it involved a special predicate $F_d(f)$. Any information that this loses lies out of reach of the elementary extension of MLSS described in the preceding paragraphs.

Our verifier provides an inference mechanism, designated by the keyword THUS, which extends ELEM deduction in the manner just explained. To make a universally quantified statement available to this mechanism, one writes

```
  ENABLE_THUS(statement_of_theorem),
```

for example

```
   ENABLE_THUS((Ord(S) & T in S) *imp Ord(T)).
```

To disable use of a theorem by 'THUS' inferencing, one can write

```
  DISABLE_THUS(statement_of_theorem),
```

The following list shows some of the commonly occurring theorems suitable for use with the 'THUS' inferencing mechanism.

```
ENABLE_THUS((FORALL s,t |     (Ord(s) & Ord(t)) *imp (s incin t or t incin s)))     ENABLE_THUS((FORALL s,t
```

Besides using all the MLSS statements available in the context in which it is invoked, the inference mechanism invoked by the keyword 'THUS' makes use of all the explicit and implicit universally quantified statements found in that context, including nonmembership statements like

```
 b notin {e(x): x in s | P(x)},
```

which are equivalent to

```
   (FORALL x | not(b = e(x) & x in s & P(x))).
```

This extends the reach of the automatic substitution mechanism invoked by 'THUS'.

Proof by equality

Proof by equality tests two expressions for equality or two atomic formulae for equivalence, by standardizing their bound variables and then descending their syntax trees in parallel until differing nodes are found. These differing nodes are then examined to determine if the context of the equality proof step contains theorems which imply that the syntactically different constructs seen are in fact equal or equivalent. Suppose, for example, that an assertion

```
   {g(e(x),f(y)): x in s, y in t | P(x,y)} = a
```

has been proved, and that

```
{g(e'(x),f'(y)): x in s, y in t | P'(x,y)} = a
```

is to be deduced from it. Syntactic comparison reveals the differences between e and e', f and f, P and P'. Our verifier's proof by equality procedure will then generate the three statements

```
(FORALL x in s | e(x) = e'(x))    (FORALL y in t | f(y) = f'(y))    (FORALL x in s, y in t | P(x,y) *eq
```

and attempt to find all of them in the available context. If this succeeds, the proof by equality inference will be accepted. If not, the equality procedure will go one step higher in the syntax tree of these two formulae, generate the pair of statements

```
(FORALL x in s, y in t | g(e(x),f(y)) = g(e'(x),f'(y)))    (FORALL x in s, y in t | P(x,y) *eq P'(x,y))
```

and search for them in the available context. This gives a second way in which proof by equality can succeed.

Proof by equality uses the equalities available in its context transitively. Since the inner suboperations of the proof by equality routine are either purely syntactic or are simple searches, this kind of inference is quite efficient.

Proof by monotonicity

Our verifier includes a 'proof by monotonicity' feature which keeps track of all operators and predicates for which monotonicity properties have been proved, and also of all relationships of domination between monadic operators and predicates. This mode of inference uses an efficient, syntactic mechanism and so works quite rapidly when it applies. Proof by monotonicity allows statements like

```
(n incs k & m incs j) *imp       ((#{[x,0]: x in n} + {[x,1]: x in m})          incs #{[x,0]: x in k} + {
```

and

```
(n incs k & m incs j) *imp       ((#{[x,y]: x in n, y in m}) incs #{[x,y]: x in n, y in m})
```

to be derived immediately. Since the formulae appearing on the right are essentially the definitions of the cardinal addition and multiplication operators respectively, this easily gives us the formulae

```
(n incs k & m incs j) *imp ((n *PLUS m) incs (k *PLUS j))
```

and

```
(n incs k & m incs j) *imp ((n *TIMES m) incs (k *TIMES j)),
```

which can then be used as the basis for further inferences by monotonicity.

Proof by monotonicity works in the following way. The monotonicity properties of all of the verifier's built-in predicates and operators are known *a priori.* For example, 'x in s' is monotone increasing in its second parameter, whereas 's incs t' is monotone increasing in its first parameter and monotone decreasing in its second parameter. 's + t' and 's * t' are monotone increasing in both their parameters; 's - t' is monotone increasing in its first parameter and monotone decreasing in its second. Quantifiers and setformers like

```
(FORALL x,y in s | P)  and  (EXISTS x,y in s | P)
```

and

```
(FORALL x,y in s | P)
```

depend in known monotone fashion on the sets which restrict their bound variables, and preserve the monotonicity properties of their qualifying clauses P. The same remark applies to setformers like

```
{e(x,y): x in s,y *incin t | P).
```

The propositional operators &, or, not, *imp transform the monotonicity properties of their predicate arguments in known ways. 'a & b' and 'a or b' are monotone increasing in both their parameters; 'not a' is monotone decreasing. 'a *imp b' is monotone increasing in its second parameter and monotone decreasing in its first parameter.

These rules allow the monotonicity properties of compound expressions like

```
(+)    {e(x,y): x in s,y *incin t |            (FORALL z,w | ([[z,x],[w,y]]) in u *imp (z in v))}
```

to be calculated directly by a procedure which processes its syntax tree bottom up and assigns a dependency characteristic to each node encountered. For example, the expression just displayed is monotone increasing in s,t, and v, but monotone decreasing in u.

Besides the properties 'monotone increasing' and 'monotone decreasing', there is one other property which it is easy and profitable to track in this way. As previously explained, an operator f(x,...) of one or more parameters is said to be *additive* in a parameter x if

```
f(x + y,...) = f(x,...) + f(y,...)
```

for all x and y, and a predicate P(x,...) is said to be additive if

```
P(x + y,...) *eq (P(x,...) & P(y,...)
```

Using this notion we can easily see that an example like (+) is additive in s, but not necessarily in its other parameters.

Many of the operators and predicates which appear repeatedly in the sequence of theorems and proofs to which the second half of this book is devoted have useful monotonicity properties. These include

is_map additive
domainadditive

rangeadditive

is_mapdecreasing

Svmdecreasing

1_1_mapdecreasing

#increasing

*ONadditive in both parameters

Finiteadditive

*PLUSincreasing in both parameters

*TIMESincreasing in both parameters

powincreasing

*MINUSincreasing in first parameter, decreasing in second

Unadditive

*OVERincreasing in first parameter, decreasing in second

The three commands

```
ENABLE_ELEM(MONOTONE_FCN; operator_and_predicate_list)  ENABLE_ELEM(MONOTONE_GROUP; operator_and_predicat
```

discussed in the previous section can be used to make the monotonicity properties of other operators available for use in proof-by-monotonicity deductions once these properties have been proved. This enlarges the class of expressions which can be handled automatically. For example, it follows immediately that

```
#pow(Un(domain(f) + range(f)))
```

is monotone increasing in f.

Many of the monotonicity properties which appear in the table shown above follow readily using proof by monotonicity. For example, from the definition of the predicate is_map, namely

```
is_map(f) :*eq f = {[car(x),cdr(x)]: x in f}
```

it is not hard to show that

```
is_map(f) *eq (FORALL x in f | x = [car(x),cdr(x)])
```

But the predicate on the right is obviously monotone decreasing in f, and so it follows that is_map(f) has this same property. The facts that the predicates Svm(f) (f is a single-valued function) and 1_1_map(f) are also monotone decreasing then follow immediately from the definitions of these predicates, which are

```
Svm(f) :*eq is_map(f) &     (FORALL x in f, y in f | (car(x) = car(y)) *imp (x = y))
```

and

```
1_1_map(f) :*eq Svm(f) &     (FORALL x in f, y in f | (cdr(x) = cdr(y)) *imp (x = y)).
```

Similarly the fact that 'f ON a' is additive in both its parameters follows immediately from its definition, which is

```
f ON a := {p in f | car(p) in a}.
```

Many small theorems used later in this book follow more or less immediately using proof by monotonicity. Some of these are

```
Theorem: ((G *incin F) & is_map(F)) *imp is_map(G)  Theorem: ((G *incin F) & Svm(F)) *imp Svm(G)  Theorem
```

The verifier's proof-by-monotonicity mechanism can examine statements whose topmost operator (after explicit or implicit universal quantifiers have bee stripped off) is '*imp' to see if the conclusion of the implication found is an inclusion derivable from the implication's hypotheses via proof by monotonicity. This allows one-step derivation of statements like the

```
(n incs k & m incs j) *imp       (#({[x,0]: x in n} + {[x,1]: x in m}) incs #({[x,0]: x in k}           +
```

considered above.

Examples of decidable sublanguages

Various predicate statements with restricted quantifiers.

More decidable sublanguages

MLS with 'ordinal', Z, **ee**, **¿j**

MLS with Un(s): the union of all elements of s

MLSS with pow(s) (the set of all subsets of s), and with the predicate Finite(s) which asserts that a set is finite.

The Un operator is interesting because

```
s /= {} & Un(s) = s
```

is satisfiable, but only by an infinite model.

Presburger's decidable quantified language of additive arithmetic

In *pres,* Presburger showed that the language of quantified statements whose variables all represent integers, and in which the only operations allowed are arithmetic addition and subtraction and the comparators n > m and n >= m, has a decidable satisfiability problem. (We will see in Chapter 4 that if the multiplication operator is added to this mix the class of formulae that results admits of no algorithm for testing satisfiability).

The technique used by Presburger is progressive elimination of quantifiers by replacement of existentially quantified set expressions by equivalent unquantified expressions of the same kind. This method of 'quantifier elimination' applies to a language L if, given any formula

```
(1)  (EXISTS x | P(x))
```

formed using just one quantifier, together with the operators allowed by the language, also the bound variable x and various free variables a_1,...,a_n, we can find an equivalent unquantified formula of the language, involving only the free variables a_1,...,a_n, which is equivalent to (1). (Note that universally quantified subformulae can always be reduced to existentially quantified form by use of the de Morgan rule

```
(FORALL x | P(x)) *eq (not (EXISTS x | not P(x))).
```

If an unquantified formula equivalent to (1) always exists, we can work systematically through the syntax tree of any formula, from bottom to top, replacing all quantified subformulae with equivalent unquantified formulae, until no quantifiers remain. For (1) to be equivalent to an unquantified formula of the language L, it may be necessary to enlarge L by adding some finite collection of supplementary operators and predicates. If quantification *à la* (1) of formulae written

using every such operator collection requires the introduction of still more operators, quantifier elimination will fail; otherwise it can be applied.

A typical means of re-expressing (1) in unquantified form is to show that if (1) has a solution at all, some one of a finite collection of unquantified expressions $e_1,...,e_k$ ('canonical solutions') written in terms of the free variables of (1) must be a solution. This allows (1) to be rewritten as the disjunction

```
P(e1) or ... or P(ek).
```

in which the quantified variable x has been eliminated.

To apply these ideas to the Presburger language of additive arithmetic formulae described above, we need to introduce one additional operator into the language. This is the divisibility operator, which we will write in the next few paragraphs as c|n. In such expressions c will always be a positive integer constant, and n an integer-valued variable or expression.

In considering 'innermost' existentially quantified Presburger-formulae (EXISTS n | P(n)) (that is, quantified formulae not containing any quantified subformulae) we can expand the (unquantified) 'body' P(x) into a disjunction of conjunctions, and then use the predicate rule

```
(EXISTS x | P(x) or Q(x)) *eq ((EXISTS x | P(x)) or (EXISTS x | Q(x)))
```

to move the existential quantifier in over the 'or' operators. In the resulting formulae each P is a conjunction of literals, and can therefore be written as

(2) $(\text{EXISTS } n \mid \&_{k=1}^{I} (a_k*n >= A_k) \ \& \ \&_{k=1}^{J} (b_k*n <= B_k) \ \& \ \&_{k=1}^{L} (c_k \mid (d_k*n + C_k)))$

where the a_k, b_k, c_k, and d_k are positive integer constants, '*' and "+" designate integer multiplication and addition respectively, and A_k, B_k, and C_k are well-formed Presburger terms not containing n. Suppose for the sake of definiteness that I > 0 in (2), and that (2) admits a solution m.

Then among these solutions, all of which exceed the largest among the quotients A_k/a_k, there must exist a smallest m_0. This m_0 will have the form $(A_{k_0} + j)/a_{k_0}$, for some k_0 and some non-negative integer j. Let c'_k denote the quotient $c_k/GCD(c_k,d_k)$, for k = 1,... ,L. Since m_0 is smallest it must be impossible to subtract any multiple of $a_{k_0} * lcm(c'_1,...,c'_L)$ from j and still have a non-negative integer. Hence

```
0 <= j < Lk0 = ak0 * lcm(c'1,...,c'L).
```

Thus (2) is equivalent to the following finite disjunction:

(3) $\text{OR}_{i=1}^{I} \ \text{OR}_{j=0}^{L_i-1} \ (\&_{k=1}^{I} (a_k*(A_i + j) >= a_i*A_k) \ \& \ \&_{k=1}^{J} (b_k*(A_i + j) <= a_i*B_k) \ \& \ \&$

Note that (3) has essentially the same form as (2), but has one less existential quantifier. In passing from (2) to (3) we have essentially 'solved' for n: n is $(a_i + j)/a_i$, where (3) serves to locate i and j within the finite set $\{[i,j]: 1 <= i <= I, 0 <= j < L_i\}$.

Treatment of the case I = 0 is similar; details are left to the reader. Decidability of the satisfiability problem for Presburger's language of quantified purely additive arithmetic now follows in the manner explained above.

A decidable quantified theory involving ordinals

Various interesting algebraic operations can be defined on the collection of all ordinals, in the following way. A set s is said to be *well ordered* if it is ordered by some ordering relationship x > y for which x > y is incompatible both with x = y and y > x, and which is such that every nonempty subset t of s contains a smallest element x, which we can write as Smallest(t). If we make the recursive definition

```
Enu(x) := if {Enu(y) : y in x} *incin s then s else Smallest(s - {Enu(y): y in x}) end if,
```

it is not hard to see that for any two ordinals x and y we have

```
(x incs y) *imp Enu(x) > Enu(y) or s = {Enu(z): z in x},
```

and from this that if s is a well ordered set, then 'Enu' is a one-to-one, order preserving mapping of some unique ordinal n onto s (where, as usual, ordinals are ordered by inclusion, or, equivalently, membership). The ordinal n derived from s in this way is called the *order type* of s, and it can easily be seen that

```
n = Min{α in Ord : {Enu(y): y in α}} .
```

The algebraic operations alluded to above are then defined by forming various totally ordered sets from pairs of ordinals and taking the order types of these sets.

Perhaps the easiest case is that of the Cartesian product {[x,y]: x in s_1, y in s_2}, with s_1 and s_2 cardinal numbers, which can be ordered lexicographically. The order type of this product is the so-called *ordinal product*, which we will write as s_1 [*] s_2, where the s_i are ordinals. In much the same way we can order the set

```
{[x1,x2,...,xk]: x1 in s1, x2 in s2,...,xk in sk}
```

of k-tuples lexicographically, thereby defining the k-fold ordinal product s_1 [*] s_2 [*]...[*] s_k, where $s_1,s_2,...,s_k$ are ordinal numbers. Since there is an evident order isomorphism (i.e. 1-1, order-preserving map) between s_1 [*] s_2 [*] s_3 and each of the ordered sets

```
{[[x1,x2],x3]: x1 in s1, x2 in s2,x3 in s3}
```

and

```
{[x1,[x2,x3]]: x1 in s1, x2 in s2,x3 in s3},
```

it follows that ordinal multiplication satisfies the associative law (s_1 [*] s_2) [*] s_3 = s_1 [*] (s_2 [*] s_3).

Given any two ordinals s_1 and s_2, we can form a well-ordered set by ordering the collection {[0,x]: x in s_1} + {[1,y]: y in s_2} of pairs lexicographically. The order-type of this set is called the *ordinal sum* of s_1 and s_2, which we will write as s_1 [+] s_2. It is not hard to see that if s_3 is a third ordinal, then both (s_1 [+] s_2) [+] s_3 and s_1 [+] (s_2 [+] s_3) have the order type of the set

```
{[0,x]: x in s1} + {[1,y]: y in s2} + {[2,y]: y in s3},
```

ordered lexicographically. Hence ordinal addition is also associative, i.e. (s_1 [+] s_2) [+] s_3 = s_1 [+] (s_2 [+] s_3). Note however that ordinal addition is not commutative, e.g. Z [+] 1 is larger than Z, but 1 [+] Z is easily seen to be Z. Note also that n [+] 1 is easily seen to be the successor ordinal of n for each ordinal n, and so is always strictly larger than n.

The smallest ordinals are the finite integers 0,1,2,... , followed by the set Z of all integers, which is the smallest infinite ordinal. From these, we can form other ordinals using the operations just introduced: Z [+] 1, Z [+] 2,...,Z [+] Z = 2 [*] Z, 3 [*] Z,...,Z [*] Z, Z [*] Z [*] Z,... . We shall now have a look at the ordering and ordinal arithmetic relationships between these and related ordinals.

Suppose that we indicate the dependence of the Enu(x) function described above on the well-ordered set s appearing in its definition by writing Enu(x) as $Enu_s(x)$. Then it is easily proved by (transfinite) induction that if t is a well-ordered set and t incs s we have $Enu_s(n) >= Enu_t(n)$ for ordinal n. (Hint: first prove by induction that

```
t - {Enut(y): y in n} incs s - {Enus(y): y in n}
```

for every ordinal n). It follows that the order type of any subset s of an ordinal n is the image under the 'Enu' function of an ordinal no larger than n. Since, as seen above, any well-ordered set is order-isomorphic to some ordinal, it follows at once that the order type of a subset of a well-ordered set s can be no larger than the order type of s.

Using this last result it is easy to see that both addition and multiplication are nondecreasing functions of both their arguments. For example, if n_1, n_2, m_1, and m_2 are all ordinals, with n_1 incs m_1 and n_2 incs m_2, then n_1 [*] n_2 is the order type of the lexicographically ordered Cartesian product C of n_1 and n_2, and m_1 [*] m_2 is the order type of the Cartesian product of m_1 and m_2, which is a subset of C and has the same lexicographic order. Hence n_1 [*] n_2 is an ordinal no smaller than m_1 [*] m_2, showing that the operation of ordinal multiplication is monotone in both its arguments. The proof of the corresponding statement for ordinal addition, which is similar, is left to the reader.

Ordinal multiplication is right-distributive over ordinal addition. That is, we have

```
(n1 [+] n2) [*] m = (n1 [*] m) [+] (n2 [*] m)
```

whenever n_1, n_2, and m are ordinals. To see this, note that (n_1 [+] n_2) [*] m is easily seen to be the order type of the set

```
{[0,x,y]: x in n1, y in m} + {[1,x,y]: x in n2, y in m}
```

and (n_1 [*] m) [+] (n_2 [*] m) can be identified with equal ease with the same set. This implies that the ordinal sum n [+] n [+]...[+] n of k copies of an ordinal n is the same as k [*] n. On the other hand, the corresponding right distributive law fails for infinite ordinals: although 2 [*] Z is Z [+] Z (the order type of two copies of the integers, the second positioned after the whole of the first), Z [*] 2 is the order type of the lexicographically ordered set of pairs

```
{[x,0]: x in Z} + {[x,1]: x in Z},
```

which is order-isomorphic to Z by the (integer arithmetic) mapping [x,i] :-> 2 * x + i.

A kind of subtraction can be defined for ordinals. More specifically, if s_1 and s_2 are ordinals and s_1 incs s_2, then we can write s_1 as an ordinal sum $s_1 = s_2$ [+] s_3. (Conversely, by the result proved in the preceding paragraph, s_2 [+] s_3 can never be less than s_2, since s_2 can be written as s_2 [+] 0). Indeed, s_1 is the union of s_2 and $s_1 - s_2$, which appear successively in s_1 [+] 1, from which it is easily seen that the order type of s_1 is the ordinal sum of the order types of s_2 and $s_1 - s_2$.

Using the ordinal subtraction operation just described we can now show that the ordinal addition operation m [+] n is strictly monotone in its second (though not in its first) argument. Indeed, if n' > n, then n' can be written as n [+] k for some non-zero ordinal k, and so m [+] n' = m [+] n [+] k is larger than m [+] n.

For any two ordinals s_1 and s_2, of which the first is at least 2 and the second is non-zero, the ordinal product s_1 [*] s_2 is strictly larger than s_2. Indeed we have (s_1 [*] s_2) incs (2 [*] s_2) = (s_2 [+] s_2) >= (s_2 [+] 1) > s_2.

The equation a [+] b = b, of which 1 [+] Z = Z is a special solution, is worth studying more closely. Note first of all that if b >= Z [*] a, then using ordinal subtraction we can write b as Z [*] a [+] c for some ordinal c, so that a [+] b = a [+] Z [*] a [+] c = (1 [+] Z) [*] a [+] c = Z [*] a [+] c = b. That is, we must have a [+] b = b whenever b >= Z [*] a. Conversely, if a [+] b = b, then 2 [*] a [+] b = (a [+] a) [+] b = a [+] (a [+] b) = a [+] b = b, and so inductively (k [*] a) [+] b = b for every finite integer k, and so k [*] a <= b for every finite integer k. It follows from this that Z [*] a <= b. For if not then we must have b < Z [*] a, so b is the proper initial segment {x: x in t | x in b} of the order type t of the Cartesian product C of Z and a, and therefore b is the order type of a proper initial segment s of C. Let [m,x] = Smallest(C-s) with m in Z and x in a. Then s is a proper initial segment of the Cartesian product of m and a, whose order type is m [*] a. Thus b < m [*] a, which contradicts the inequality m [*] a <= b derived above. Therefore Z [*] a <= b, as stated. Together all this proves that a [+] b = b if and only if b >= Z [*] a, i.e. if and only if b is 'substantially' larger than a, in this sense. Note that our argument also proves that if n and m are ordinals, and m >= k [*] m for every finite integer k, then m >= Z [*] m.

Write the k-fold product of any ordinal n with itself as n [**] k. The associative law for ordinal multiplication implies that (n [**] j) [*] (n [**] k) = n [**] (j + k) (where j + k denotes the integer sum of j and k). If n is greater than 1, and in particular if n = Z, then the sequence of powers n, n [**] 2, n [**] 3,... is strictly increasing. Indeed we have

```
n [**] (i + 1) = n [*] (n [**] i) >= 2 [*] (n [**] i) =      (n [**] i) [+] (n [**] i) >= (n [**] i) [+]
```

We will call an ordinal n a *polynomial ordinal* if it has the form

```
ck [*] (Z[**]k) [+] ck-1[*](Z[**](k-1))      [+] ck-2[*](Z[**](k-2)) [+]...[+] c1[*]Z [+] c0,
```

where all the coefficients c_i are finite integers. These ordinals, which we shall write as $Pord(c_k,c_{k-1},...,c_0)$, have the following properties: (i) Two polynomial ordinals are distinct if their coefficient sequences $c_k,c_{k-1},...,c_0$ are distinct. (ii) Two polynomial ordinals compare in the lexicographic order of their coefficients. (If one of the sequences of coefficients is shorter, it should be prefixed with zeroes to give it the length of the other sequence of coefficients.) (iii) the ordinal sum of two polynomial ordinals $p = Pord(c_k,c_{k-1},...,c_0)$ and $p' = Pord(c'_{k'},c'_{k'-1},...,c'_0)$ with k >= k' is given by the following rule: Locate the leftmost position i in which the second argument has a nonzero coefficient; take the coefficients of the first argument to the left of this position; in the i-th position, add c_i and c'_i; in later positions take the coefficients of the second argument. This rule is expressed by the formula

```
Pord(ck,ck-1,...,c0) [+]  Pord(c'k',c'k'-1,...,c'0) =      Pord(ck,ck-1,...,ci + 1,ci + c'i,c'i - 1,...,c'0)
```

for the sum of these two polynomial ordinals, where $c'_{k'}=c'_{k'-1}=...=c'_{i+1}=0$ and $c'_i \neq 0$.

To prove (i-iii), note that (ii) implies (i), so that only (ii) and (iii) need be proved. (ii) can be proved as follows. Consider two ordinals, both having the general form

```
(a)    Pord(ck,ck-1,...,c0),
```

and suppose that the first difference between their coefficients occurs at the position i. Since it is obvious from the definition of (a) and by associativity that $Pord(c_k,c_{k-1},...,c_0) = Pord(c_k,c_{k-1},...,c_{i+1},0,...,0)$ [+] $Pord(c_i,c_{i-1},...,c_0)$, and since the ordinal addition operator is strictly monotone in its second argument, we can suppose without loss of generality that i = k, and therefore need only prove that if two polynomial ordinals (*) differ in their first coefficient, the one with the larger first coefficient is larger. This is simply a matter of proving that $Pord(c_k,c_{k-1},...,c_0) < (c_k + 1)$ [*] Z[**]k, i.e. that $Pord(c_{k-1},...,c_0) < Z$[**]k. But it is easily seen that $Pord(c_{k-1},...,c_j)$ [+] Z[**]k = $Pord(c_{k-1},...,c_{j+1})$ [+] Z[**]k for every j <= k, from which it follows inductively that $Pord(c_{k-1},...,c_0)$ [+] Z[**]k = Z[**]k. It is easily seen from this that $Pord(c_{k-1},...,c_0) < Z$[**]k, as claimed, thus proving (ii).

To calculate the sum of two polynomial ordinals $Pord(c_k,c_{k-1},...,c_0)$ and $Pord(c'_i,c'_{i-1},...,c'_0)$ (both written with nonzero leading coefficients) we can note first of all that if k < i then we can show, as at the end of the preceding paragraph, that

Pord(c_{k-1},...,c_j) [+] c'$_i$[*]Z[**]i = c'$_i$Z[**]i. From this, (iii) follows immediately by associativity of ordinal addition in the special case in which k < i. Now suppose that k >= i. Then by associativity of ordinal addition we have

```
(b)     Pord(ck,ck-1,...,c0) [+] Pord(c'i,c'i-1,...,c0)      = Pord(ck,ck-1,...,ci + 1) [+] ci[*]Z[**]i [+] Pord(ci-
```

proving (iii).

By the rule (ii) stated above, Pord(c_k,c_{k-1},...,c_0) < Z[**](k + 1). Conversely, we will show that if n is any ordinal such that n < Z[**](k + 1), then n is a polynomial ordinal of the form Pord(c_k,c_{k-1},...,c_0). To see this, argue inductively on k, and so suppose that our statement is true for all k' < k. Then find the largest integer c such that n >= c[*](Z[**]k); this must exist since we have seen above that if n >= c[*](Z[**]k) for all integers c, it would follow that n >= Z[**](k + 1), which is impossible. By the subtraction principle stated above, we can write n = c[*](Z[**]k) [+] m for some ordinal m. If m >= Z[**]k, then n >= c[*](Z[**]k) [+] (Z[**]k) = (c + 1)[*](Z[**]k), contradicting the definition of c. It follows by induction that m is a polynomial order and can be written as Pord(c_{k-1},...,c_0), from which it follows immediately that n = Pord(c,c_{k-1},...,c_0), as asserted.

It follows that the smallest ordinals are precisely the polynomial ordinals, and that the first positive ordinal larger than all the polynomial ordinals is the union of all the powers Z[**]k for integer k. This is the order type of the collection of all infinite sequences [...,n_i,n_{i-1},...,n_0] which begin with infinitely many zeroes, lexicographically ordered.

We will say that an ordinal n is *post-polynomial* if, whenever m < n and p is a polynomial ordinal, m [+] p < n also. The first post-polynomial ordinal is the zero ordinal { }; this is the only post-polynomial ordinal which is also polynomial. Moreover the sum n_1 [+] n_2 of any two post-polynomial ordinals is itself post-polynomial. For if i is an ordinal such that i < n_1 [+] n_2 and p is a polynomial ordinal, then if i < n_1 we have i [+] p < n_1 also, and therefore i [+] p < n_1 [+] n_2. On the other hand, if i >= n_1, we can write i = n_1 [+] j for some ordinal j, and by the strict monotonicity of ordinal addition in its second argument we must have j < n_2, so j [+] p < n_2, and therefore

```
  i [+] p = n1 [+] j [+] p < i [+] p < n1 [+] n2
```

proving that i [+] p < n_1 [+] n_2 in all cases.

We shall now show that any ordinal n can be decomposed uniquely as an ordinal sum n = m [+] p, where m is post-polynomial and p is a polynomial ordinal. Moreover, in this decomposition, ordinals n have exactly the lexicographic ordering of the corresponding pairs [m,p]. To show this, note first of all that the union u of all the elements of any set s of post-polynomial ordinals must itself be post-polynomial. Indeed, if k is an ordinal < u, then k is a member of u and hence of some j in u, so that k + p < j for all polynomial ordinals p, and hence k + p < u. It follows that the union m of all the post-polynomial ordinals not greater than n is itself post-polynomial. Clearly m is the largest post-polynomial ordinal <= n. By the subtraction principle for ordinals stated above there exists an ordinal x such that n = m [+] x. x cannot be >= the first nonzero post-polynomial ordinal f, since if it were, then we would have n >= m [+] f, but we have seen above that m [+] f is post-polynomial, and since it is clearly greater than m we have a contradiction. Hence x is less than f, and so is polynomial, proving that n can be decomposed as an ordinal sum n = m [+] p. Uniqueness is proved in the next paragraph.

The decomposition n = m [+] p of an ordinal n into the sum of a post-polynomial and a polynomial ordinal is unique, since if m [+] p = m' [+] p' for distinct post-polynomial m, m', then one of these two, say m', must be larger than the other. But then m > m' [+] p', contradicting m [+] p = m' [+] p'. Similarly, if m [+] p > m' [+] p', we must have m >= m', and if m = m', then p > p' by the monotonicity of ordinal addition. This shows that the lexicographic ordering of the pairs [m,p] corresponds exactly to the standard ordering of the corresponding ordinals n = m [+] p.

In what follows we shall say that a polynomial ordinal is of *degree k* if it has the form Pord(c_k,c_{k-1},...,c_0) with either c_k /= 0 or k =0. The function Cf$_j$(p) is defined to return the j-th coefficient of the polynomial ordinal p, or, if j exceeds the degree of p, to return 0. We extend this function to all ordinals by writing Cf$_j$(m [+] p) = Cf$_j$(p) if p is a polynomial ordinal

and m is post-polynomial. If p = Pord(c_k,c_{k-1},...,c_0) is a polynomial ordinal and j an integer, we let $Hi_j(p)$ be Pord(c_k,c_{k-1},...,c_{j+1}) and $Low_j(p)$ be Pord(c_j,c_{j-1},...,c_0). These operations are extended to general ordinals in the same way we extended Cf_j. Using these functions, we define three auxiliary functions x [-] p, x [^] p, and x [~] p for use below. These are defined for any ordinal x and polynomial ordinal p: If p is of degree d and c is its leading coefficient, then Hi_d(x [-] p) = Hi_d(x); Cf_j(x [-] p) is Cf_j(x) - c if this is positive, otherwise 0; and Low_{d-1}(x [-] p) = Low_{d-1}(p). Similarly Hi_d(x [^] p) = Hi_d(x); Cf_j(x [^] p) is 0; and Low_{d-1}(x [-] p) = Low_{d-1}(p). Finally Hi_d(x [~] p) = Hi_d(x) and Low_d(x [~] p) = Low_d(p).

We will also need to use various properties of these operators, as used in combination with each other and in combination with the comparators '>' and '='. These are as follows (where y, z are arbitrary ordinals, and p, q are polynomial ordinals of degrees d and d' and leading coefficients c and c' respectively):

```
(i) (y [+] p) [+] q = y [+] (p [+] q)

(ii) (y [+] p) [-] q = if d > d' then y [+] (p [-] q)        elseif d' > d then y [-] q         else

(iii) (y [+] p) [^] q = if d > d' then y [+] (p [^] q)        else y [^] q end if

(iv) (y [+] p) [~] q = if d > d' then y [+] (p [~] q)        else y [~] q end if

(v) (y [-] p) [+] q = if d > d' then y [-] (p [+] q)        elseif d' > d then y [+] q         elsei

(vi) (y [-] p) [-] q = if d > d' then y [-] (p [-] q)        elseif d' > d then y [-] q         else

(vii) (y [-] p) [^] q = if d > d' then y [-] (p [^] q)        else y [^] q end if

(viii) (y [-] p) [~] q = if d > d' then y [-] (p [~] q)        else y [~] q end if

(ix) (y [^] p) [+] q = if d > d' then y [^] (p [+] q)         elseif d' > d then y [+] q         els

(x) (y [^] p) [-] q = if d > d' then y [^] (p [-] q)         elseif d' > d then y [-] q         else

(xi) (y [^] p) [^] q = if d >= d' then y [^] (p [^] q)         else y [^] q end if

(xii) (y [^] p) [~] q = if d >= d' then y [^] (p [~] q)         else y [~] q end if

(xiii) (y [~] p) [+] q = if d > d' then y [~] (p [+] q)         elseif d' > d then y [+] q         e

(xiv) (y [~] p) [-] q = if d > d' then y [~] (p [-] q)         elseif d' > d then y [-] q         el

(xv) (y [~] p) [^] q = if d >= d' then y [~] (p [^] q)         else y [^] q end if

(xvi) (y [~] p) [~] q = if d >= d' then y [~] (p [~] q)         else y [~] q end if

(xvii) ((y [+] p) > z) *eq (y >= (z [+] r')         or (y >= (z [^] r) & ((Cfd(z) < c)          or (y

(xviii) ((y [-] p) > z) *eq (y >= (z [+] r') or     (y >= (z [^] r) &      ((Cfd(y) <= c & p [^] p >

(xix) ((y [^] p) > z) *eq if (p [^] p) > Lowd(z) then y >= z [^ ] r        else y >= z [+] r' end if

(xx) ((y [~] p) > z) *eq z *eq if p > Lowd(z) then y >= z [^ ] r        else y >= z [+] r' end if

(xxi) ((y [+] p) > z) *eq (y >= (z [+] r')         or (y >= (z [^] r) & ((Cfd(z) < c)          or (y

(xxii) ((y [-] p) >= z) *eq (y >= (z [+] r') or     (y >= (z [^] r) &      ((Cfd(y) <= c & p [^] p >=
```

```
(xxiii) ((y [^] p) >= z) *eq if (p [^] p) >= Low_d(z) then y >= z [^ ] r        else y >= z [+] r' end

(xxiv) ((y [~] p) >= z) *eq if p >= Low_d(z) then y >= z [^ ] r        else y >= z [+] r' end if

(xxv) Cf_j(y [+] p) =     if j > d then Cf_j(y) else Cf_j(y) + Cf_j(p) end if

(xxvi) Cf_j(y [-] p) = if j > d then Cf_j(y)     elseif Cf_j(y) >= Cf_j(p) then Cf_j(y) - Cf_j(p) else 0 end

(xxvii) Cf_j(y [^] p) =     if j > d then Cf_j(y) else Cf_j(p') end if,        where p' is the polynomia

(xxviii) Cf_j(y [~] p) = if j > d then Cf_j(y) else Cf_j(p) end if
```

These rules have the following proofs. (i) is a consequence of the associative law for ordinal addition. For (ii), note that if d > d' then in the range of coefficients relevant to the formation of (y [+] p) [-] q the coefficients of y will have been replaced, in y [+] p, by those of p, from which the first case of (ii) follows immediately. On the other hand, if d' > d, then the difference between y and y [+] p is irrelevant to the formation of (y [+] p) [-] q, and thus the second case of (ii) follows. Finally, if d' = d, then the coefficient $Cf_j((y [+] p) [-] q)$ is $Cf_j(y) + (Cf_j(p) - Cf_j(q))$ if p has a larger leading coefficient than q. However, if q has a larger leading coefficient than p, then $Cf_j((y [+] p) [-] q)$ is $Cf_j(y) - (Cf_j(q) - Cf_j(p))$, or 0 if this difference is negative. In both these cases, all lower coefficients are those of q, proving rule (ii) in the remaining cases.

In regard to rule (iii), note that if d >= d' then in the range of coefficients relevant to the formation of (y [+] p) [^] q the coefficients of y will have been replaced (in y [+] p) by those of p, from which the first case of (iii) follows immediately. On the other hand, if d' > d, then the difference between y and y [+] p is irrelevant to the formation of (y [+] p) [-] q, and thus the second case of (iii) follows. The proofs of (iv), (vii), (viii), (xi), (xii), (xv), and (xvi) are essentially the same, so we leave details to the reader.

The proofs of the first two cases of case of (v), (vi), (ix), (x), (xiii), and (xiv) are much the same as that of the corresponding cases of rule (ii) and are also left to the reader. In the remaining cases of these rules, p and q have the same degree d. In all these cases, the coefficients Cf_j of the result being formed are always those of q for j < d; only the coefficients Cf_d requires closer consideration. In regard to the d = d' case of rule (v), note that in this case if the leading coefficient c of p is larger than the corresponding coefficient of y, y [-] p will have a zero d-th coefficient, so (y [-] p) [+] q will simply be y [~] q. But if c is not larger than the corresponding coefficient of y, then the d-th coefficient of (y [-] p) [+] q will be $Cf_d(y) + c - c'$, i.e. is that of y [-] (p [-] q) if c >= c', but that of y [+] (p [-] q) otherwise. Since the remaining coefficients of (y [-] p) [+] q are those of q in any case, Rule (v) follows.

The d = d' case of rule (vi) follows in the same way since the d-th coefficient of (y [-] p) [-] q is always that of y [-] (p [+] q), and the remaining coefficients of (y [-] p) [-] q are those of q. In the d = d' case of rule (ix), the d-th coefficient of y [^] p is zero, hence the d-th coefficient (y [^] p) [-] q is that of q, while the remaining coefficients are those of q, proving rule (ix) in this case. The d = d' cases of rules (x), (xiii), and (xiv) follow by similar elementary observations, whose details are left to the reader.

Rules (xxv-xxvii) follow directly from the definitions of the operators [+], [-], [^], and [~] and the coefficient functions Cf_j. Their proofs are left to the reader.

To prove rule (xvii), note first of all that (y [+] p) > z will hold either if $Hi_d(y) > z$, in which case the values of Low_d(y [+] p) and $Low_d(z)$ are all irrelevant, or otherwise if $Hi_d(y) = Hi_d(z)$ (which in this case we can write as $Hi_d(y) >= Hi_d(z)$), in which case we must have $Low_d(y [+] p) > Low_d(z)$. But $Hi_d(y) > z$ is equivalent to y > z [+] r', and $Hi_d(y) >= z$ is equivalent to y >= z [^] r, where r and r' are as in (xvii). (This last remark applies in the proofs of all the rules (xvii-xxv)). In the y >= z [^] r case of (xvii), if $c > Cf_d(z)$ then (y [+] p) > z is certainly true, while if $c <= Cf_d(z)$ then we must have both $Hi_{d-1}(y) >= Hi_d(z [-] p)$ and $Low_{d-1}(p) >= Low_{d-1}(z)$. The final clauses in (xvii) merely restate these conditions, by rewriting $Hi_{d-1}(y) >= Hi_d(z [-] p)$ as $z [-] r^d$ and $Low_{d-1}(p) >= Low_{d-1}(z)$ as $p [^] p > Low_{d-1}(z)$.

The proofs of rules (xviii-xiv) generally resemble that just given for rule (xvii), and in some cases are distinctly simpler. To prove rule (xviii), we note as above that (y [-] p) > z will hold either if $Hi_d(y) > z$, or otherwise if (y [-] p) >= z & $Low_d(y [-] p) > Low_d(z)$. If $Cf_d(y) <= c$ then $Low_d(y [-] p) = p [-] p$; otherwise $Low_d(y [-] p) > Low_d(z)$ is equivalent to

$Cf_d(y) > c$ or $(Cf_d(y) = c$ & $Low_{d-1}(p) > Low_{d-1}(z))$,

which rule (xviii) merely restates.

The proofs of rules (xix), (xx), (xxiii), and (xxiv) are similar but simpler, and are left to the reader. The proof of rule (xxi) is almost the same as that of (xvii), merely involving a change from p [^] p > $Low_{d-1}(z)$ to p [^] p >= $Low_{d-1}(z)$. The proof of rule (xxii) is like that of (xviii), merely involving the change of p [^] p > $Low_{d-1}(z)$ and p [^] p > $Low_d(z)$ to p [^] p >= $Low_{d-1}(z)$ and p [^] p >= $Low_d(z)$ respectively.

These observations complete our proofs of all the rules (i-xxvii) stated above.

Let LO be the language of quantified formulae whose variables designate ordinals and whose only allowed operation is that which forms the maximum of two ordinals x and y, which for convenience we will write as x @ y. We say that a subexpression

```
(1)   (EXISTS x | P(x))
```

of a formula of LO is *of level k* if it contains level (k-1) subexpressions, but none of any higher level; quantifiers not containing any quantified subexpression will be said to be of level 0. Using this notion, we will show that the satisfiability problem for the language LO is decidable. The following result implies this, and gives a convenient form to the necessary decision procedure.

Theorem: Let S be a statement, in the language LO, containing no free variables, and suppose that L is the maximum level, in the sense defined above, of any quantified subexpression of S. Then the truth value of S, quantified over the collection of all ordinals, is the same as the truth value obtained if all the quantifiers in S are restricted to range over polynomial ordinals of degree at most L.

Since every polynomial ordinal of degree at most L is described by a set of L + 1 integer coefficients, and comparisons between two such ordinals and the maximum of two such ordinals can be written as expressions involving only integer comparisons and sums, it follows from this theorem that the satisfiability problem for the language LO reduces to a special decision problem for Presburger's language of additive arithmetic, and so, by the result presented in the previous section, is decidable.

As an example illustrating the use of the theorem just stated, we consider the formula

```
(6) (EXISTS x | (FORALL x' | (x' < x) *imp (EXISTS x* | x* > x' & x* < x))         & (EXISTS y | y < x)
```

The existential clause in the first line states that x is a limit ordinal, and the following clause states that y is less than x. Thus the smallest possible y and x satisfying the condition displayed are 0 and Z respectively. This example makes it plain that the predicate Is_limit(x) stating that x is a limit ordinal can be defined in the language LO. Therefore so can the predicates

```
Is_limit_2(x) :*eq     (EXISTS x | (FORALL x' | (x' < x) *imp (EXISTS x* | x* > x' & x* < x)          & Is_l
```

and so forth. From this, it is easy to see that one can write formulae in LO whose smallest solutions are the ordinals Z [**] 2, Z [**] 3,..., and indeed any polynomial ordinal. The theorem stated above tells us that ordinals larger than every polynomial ordinal cannot be described by formulae of LO, and bound the size of the ordinals that can be described by

formulae of any specified quantifier nesting level.

To prove the Theorem stated above, we first note that any quantified formula of LO can be replaced by an equivalent formula of LO containing no occurrences of the binary operator '@' which returns the maximum of its arguments. To see this, we note that every term appearing in F must be a comparison having either the form $t_1 > t_2$ or $t_1 = t_2$, where t_1 and t_2 are either simple variables or literals formed using the '@' operator. But if t_1 has the form $t_1 = x @ t$, where x is some variable chosen for processing, we can rewrite $t_1 > t_2$ as

```
(1)      (x = t & x > t2) or (x > t & x > t2) or (t > x & t > t2),
```

and similarly rewrite $t_1 = t_2$ as

```
(1)  (x = t & x = t2) or (x > t & x = t2) or (t > x & t = t2).
```

Similar remarks apply if t_2 has the form $t_2 = x @ t$. Applying these transformations repeatedly, as often as necessary, we eventually remove all occurrences of '@' from F, replacing it by a formula written only with quantifiers and the comparisons '>' and '='. Note that the transformation we have described leaves the level of each quantifier in F unchanged.

But now, having removed all occurrences of '@', we re-complicate our language LO by introducing the four additional operators [+], [-], [^], and [~] described above, plus the family of auxiliary predicates Cf_j, into it. Note once more that in occurrences t [+] p, t [-] p, t [^] p, and t [~] p of the operators [+], [-], [^], and [~] the second argument p is required to be some polynomial ordinal with coefficients known explicitly. Let LO' designate the language LO, extended in this way, but with occurrences of '@' forbidden.

With this understanding, we process the existentially quantified subexpressions

```
(1)  (EXISTS x | P(x))
```

of our given formula of LO' in bottom-to-top syntax tree order. As processing proceeds, we continually apply rules (i-xvi) and (xxv-xxviii). This reduces all the literals appearing in P(x) to forms like y [+] p, y [-] p, y [^] p, and y [~] p, where y is a simple variable and p an explicitly known polynomial ordinal, and every occurrence of a predicate Cf_j to the form $Cf_j(y) = c$, where y is a simple variable and both j and c are explicitly known integers. Note in this conditions that inequalities like $Cf_j(y) <= c$, where c is some explicit integer constant, can be written as a disjunction of the equalities $Cf_j(y) = e$, over all e <= c, and so do not violate our requirement that all occurrences of Cf_j must be in contexts $Cf_j(y) = c$. Likewise, inequalities $Cf_j(y) > c$ are disjunctions of negated equalities $Cf_j(y) = e$, over all e <= c. p conditions like p [^] p > $Low_{d-1}(z)$), which appear in rules like (xvii) and (xviii), can be rewritten, if we use the fact that the order of polynomial ordinals is the lexical order of their coefficients, in terms of inequalites between the coefficents $Low_j(z)$ and known integer constants, and then also as Boolean combinations of equalites $Cf_j(y) = c$.

As the processing described in the preceding paragraph goes on, we always push conditionals introduced by applications of rules (i-xxviii) using relationships like

```
if C1 then A1 elseif C2 then A2 elseif...else Ck end if [+] p    = if C1 then A1 [+] p elseif C2 then A2 [+]
```

When the predicate level is reached we use rules (xvii-xxviii), plus rules like

```
if C'1 then A'1 elseif C'2 then A'2 elseif...else C'k end if *eq     ((C'1 & A'1) or ((not C'1) & C'2 & A'2)
```

to eliminate any conditional expressions that may have accumulated. The final Boolean combination that results is then reduced to a disjunction of conjunctions. We will prove recursively that this process can be used to reduce any level k existential (in the sense defined above) to an equivalent disjunction of conjunctions, each involving only variables free in the existential, together with expressions of the form y [+] p, y [-] p, y [^] p, and y [~] p, where p is a polynomial ordinal

of degree at most k with explicitly known constant integer coefficients, also the comparators >, >=, and conditions of the form $Cf_j(y) = c$, where c is a known integer constant no greater than k.

To prove this by induction on k, suppose that it is already known for all existentials of level lower than k, and consider an existential (1) of level k involving only the operators listed above. Then P(x) begins (before application of the rules (i-xvi) and (xxv-xxviii)) as an expression involving combinations t [+] p, t [-] p, t [^] p, and t [~] p with p of degree at most k - 1, plus Cf_j with j no larger than k - 1, and comparisons involving the operators '>' and '>='. Application of the rules (i-xvi) and (xxv-xxviii) does not introduce any polynomial ordinals of higher degree, or any Cf_j with j larger than k - 1. Call a subexpression of P(x) *x-free* if it does not involve the bound variable x. When the predicate level is reached, comparisons of the form y > z and y >= z are reduced using rules (xvii-xxiv), unless they are x-free, in which case they are left as they stand. Non x-free comparisons can have either one or two arguments in which x appears. If x appears only in the first of these two arguments, we use rules (xvii-xxiv) to rewrite the comparison as a conjunction of comparisons of the form x > t and x >= t, where t is x-free, but where now polynomial ordinals of degree k can appear in t (e.g. as the polynomial r' seen in rules (xvii-xx)). Conditions of the form Cf_j with j no larger than k - 1 can also appear. Cases in which x appears only in the second of the two arguments of a comparison can be handled by rewriting a > b as (not (b >= a)) and a >= b as (not (b > a)). Cases in which x appears in both arguments of a comparison will have forms like x [+] p > x [-] q and x [^] p >= x [-] q. To handle these, we observe that all such comparisons as boolean combinations of comparisons between known integers and coefficients $Cf_j(x)$ with $j < k$, and so are in accord with the inductive condition we require.

Once the P(x) of (1) has been rewritten in the manner described in the preceding paragraph, it can be further rewritten as a disjunction of conjunctions. Then we can use predicate relationships like

```
(EXISTS x | Q(x) or R(x)) *eq ((EXISTS x | Q(x)) or (EXISTS x | R(x)))
```

to replace existentials of disjunctions by disjunctions of existentials. We can also move all x-free conjuncts out of the existential, at which point it only remains to consider existentially quantified subexpressions of the form (1) in which P(x) is a conjunct W of conditions of the following forms:

```
(a) x > t, where t is x-free, and involves no polynomial ordinal         of degree greater than k; (b) x >
```

If such a conjunction W can be satisfied (i.e. if the existential (1) can have the value 'true'), then for each j it can contain at most one conjunct $Cf_j(x) = c$, since a second conjunct $Cf_j(x) = c'$ with x /= c' would be inconsistent with this. Moreover, if there is such a conjunct, then any other conjunct $Cf_j(x)$ /= c' must either be inconsistent with or implied by this, and hence could be dropped. Also, conjuncts x > t can be written as x >= t [+] 1. Hence we can suppose without loss of generality that we have (a') no conjuncts of the form (a) and no negations of such conjuncts; (b') for each j, at most one conjunct of the form (d), and if so no conjuncts (e); (c') some finite collection of conjuncts of the form (e).

If, for particular values of the free variables which appear in it, such a W is satisfied by some ordinal value of the bound variable x, it is satisfied by a smallest such x, which we shall call x_0. Of all the t that appear in conditions of the for (b), let t_0 be the largest (for the same particular values of the free variables which appear in (1)). Then (by the subtraction principle stated earlier) x_0 can be written as $x_0 = t_0$ [+] u for some ordinal u. Write u = u' [+] p, where u' is a post-polynomial ordinal and p is a polynomial ordinal. Then t_0 [+] p is no larger than t_0 [+] u' [+] p, but satisfies all the conjuncts (b-e) present in W. Hence x_0 must have the form t_0 [+] p, where p is a polynomial ordinal. We can show in much the same way that the degree of p can be no larger than k. If, for a given j, W contains a conjunct of kind (c), it specifies the corresponding coefficient of t_0 [+] p uniquely, and in particular gives us an explicit upper limit for the corresponding coefficient of p. Moreover, if conjuncts (e) occur for a given j, and we let c_0 be the maximum of all the c that occur in these conditions, then if there is a polynomial ordinal p with $Cf_jf(p) > c_0 + 1$ for which t_0 [+] p satisfies all the conjuncts in W, then the same is true for t_0 [+] p', where p' is the same as p except that its coefficient $Cf_jf(p)$ is reduced to $c_0 + 1$. We see in much the same way that if, for a given j, W contains neither a conjunct of form (d) nor of form (e), then the p corresponding to the smallest t_0 [+] p satisfying W must have $Cf_jf(p) = 0$. Overall we see that explicit upper limits are available for each of the $Cf_jf(p)$ coefficients of the polynomial ordinal corresponding to the smallest t_0 [+] p

satisfying W. Hence, if we let $p_1,...,p_n$ be an enumeration of all these polynomial ordinals, let t vary over all the x-free expressions $t_1,...,t_m$ appearing in conjuncts (b) of W, and let x vary over all the corresponding sums t_i [+] p_j (doing this for all the disjuncts into which (1) has been decomposed), then one of these x will satisfy the quantified condition (1) if there exists any x which satisfies it. It follows that (1) is equivalent to a disjunction of finitely many alternatives of the form $P(t_i$ [+] $p_j)$, completing our inductive step and thereby completing our proof of the theorem stated above.

A language of additive infinite cardinal arithmetic. The decision algorithm just described carries over easily to the following quantified language LC. Variables in LC designate infinite cardinal numbers, and the only operation allowed is cardinal addition. To see that the satisfiability problem for LC is decidable, let n be any ordinal, and let Aleph(n) designate the n-th member, in increasing order, of the collection of all infinite cardinals. Since the sum (or product) of any two infinite cardinals is the larger of the two, the function Aleph is an order isomorphism of the collection of all infinite cardinals, taken with the operation of cardinal addition, onto the collection of all ordinals, taken with the operation which forms the maximum of two ordinals. This operation evidently maps the satisfiability problem for LC to the satisfiability problem for the language LO studied above, and so is solved using the algorithm we have just given for determining the satisfiability of statements in LO.

(FILL IN)

By combining the result stated in the last paragraph with the Presburger decision algorithm given earlier, we can obtain an algorithm for deciding the satisfiability of the quantified language LC obtained by letting variables denote cardinals which are allowed to be both finite and infinite. (FILL IN)

Behmann's quantified language of elementary set-theoretic formulae

We now turn our attention to the class of formulae studied by Behmann, namely quantified formulae in which the unquantified expressions and predicates which appear are set-theoretic expressions formed from set-valued variables by use of the elementary set operators a * b, a + b, a - b and the set inclusion operators a incs b and a *incin b (but excluding the membership operator x in a, which if allowed in the quantified setting we consider would at once make our formulae too general to be decidable by any algorithm).

We shall call the class of quantified set-theoretic formulae limited in this way the *Behmann formulae.*

It is easy to see that these formulae are powerful enough to restrict the cardinality of the sets which appear within them. For example, the condition

```
s /= {} & (not(EXISTS x | s incs x & s /= x & x /= {}))
```

is readily seen to express the condition Is_singleton(s) that s should be a singleton. Then, using this formula as a component we can write the formula

```
(EXISTS x,y | x * y = {} & x + y = s & Is_singleton(x) & Is_singleton(y))
```

which is easily seen to express the condition #s = 2. It should be plain that the condition #s = n can be expressed in much the same way for any given integer n. Thus Behmann's class of formulae is strong enough to express theorems like

```
#s = 10 *imp #(s - t) > 4 or #(s * t) > 4,
```

i.e. to express elementary facts about the cardinality of sets. Hence any algorithm able to decide the satisfiability of all Behmann formulae must be strong enough to decide certain elementary arithmetic statements. Behmann gave such an algorithm, which we will now explain. It will be seen that this decision procedure uses the Presburger algorithm described earlier as a subprocedure.

If we begin our examination of Behmann's class of quantified formulae by confining ourselves to the case (6) in which just one quantifier appears, and appears as a prefix, and allow ourselves to write set union as a sum, set intersection as an ordinary product, and the complement of the set x as -x, then any formula (6) can be written as (a disjunction of formulae of the form)

(8) (EXISTS x | & $_{nk=1}$(a$_k$ * x + (b$_k$ - x) = {}) & & $_{mk=1}$ (c$_k$ * x + (d$_k$ - x) /= {}))

To see this, note that the only operators allowed in Behmann's language are union, intersection, and complementation, and the only comparators are a incs b and a *incin b. a incs b can be written as b - a = { }, and similarly for a *incin b. Thus we can drop the 'incs' and '*incin' comparators and use equality with the nullset as our only comparator. Let x be the variable which is quantified in the Behmann formula or subformula (EXISTS x | B) that concerns us. Using the identity

```
(EXISTS x | P or Q) = (EXISTS x | P) or (EXISTS x | Q)
```

as often as necessary, we can suppose without loss of generality that B is a conjunction of comparisons, some negated, and so all having the for t = { } or t/= { }, where the term t that appears is formed using the union, intersection, and complementation operators. Using deMorgan's rules for the complement, the distributivity of union over intersection, and the fact that y * y = y for any set y, t we can rewrite t as the union of three terms t = t_1*x + (t_2 - x) + t_3, where t_1, t_2, and t_3 are all set terms not containing the variable x. Then, making use of the fact that

```
t1*x + (t2 - x) + t3 = {}
```

is equivalent to

```
t1*x + (t2 - x) = {} & t3 = {},
```

we can move the x-independent clause t_3 = { } out from under the quantifier, leaving us with an existentially quantified conjunction of equalitesand inequalites of jut the form seen in (8), as asserted.

In addition, since (a = { } & b = { }) *eq (a + b = { }), we can always assume n = 1 in (8). The detailed treatment of (8) rapidly grows complicated as m increases; its general treatment, due to Behmann, will be reviewed below. However, since this treatment is hyperexponentially inefficient, we first examine the two simplest cases m = 0 and m = 1, in which easy and efficient techniques are available.

In the case m = 0 we must consider

```
(9)  (EXISTS x | (a * x + (b - x)) = {})
```

which is to say (EXISTS x | b *incin x & x *incin comp(a)), where comp(a) designates the complement of the set a. Here a (minimal) solution is x = b, *so (9) is equivalent to a * b = {}.*

Recursive use of this observation allows some multivariable cases resembling (9) to be solved easily, e.g. to solve

```
(10)    (EXISTS x,y | a++ * x * y + ((a+- * x) - y) + ((a-+ - x) * y) + (a-- - x - y) = {})
```

we use (9) to rewrite it as

```
(11)  (EXISTS x | (a++ * x + (a-+ - x)) * (a+- * x + (a-- - x)) = {}.
```

Multiplying out, we see that this is equivalent to

```
    (EXISTS x | (a++ * a+- * x + ((a-+ * a--) - x)) = {},
```

and so to $a_{++} * a_{+-} * a_{-+} * a_{--} = \{\}$. We see in the same way that (11) has the solution $x = a_{-+} * a_{--}$, from which we obtain the solution

```
 y = (a+- * a-+ * a--) + (a-- - a-+ * a--)          = (a+- * a-+ * a--) + (a-- - a-+)
```

for y.

Proceeding to the next level of recursion we can now treat

```
 (EXISTS x,y,z | a+++*x*y*z + (a++-*(x*y)-z) + ((a+-+-(x*z)-y)) + (a+--*x-y-z)      + a-++*y*z-x + (a-+-*y-x-z) +
```

Using our solution of (10) we can rewrite this as

```
   (EXISTS x | (a+++*x + (a-++-x)) * (a++-*x + (a-++-x))          * (a+-+*x + (a--+-x)) * (a+--*x + (a---x)) = {})
```

'Multiplying out' it follows as above that a solution exists if and only if

```
    a+++*a++-*a+-+*a+--*a-++*a-+-*a--+*a--- = {}.
```

The reader will readily infer the condition for solvability of the corresponding k-variable case.

Next let m = 1 and consider

```
(12) (EXISTS x | (a * x + (b - x) = {}) & (c * x + (d - x)) /= {}))      *eq (EXISTS x | (b *incin x) & (x *
```

By adding a point z in c - a to a solution x of (12) we never spoil the solution, and hence if (12) has a solution it has one of the form b + (c - a) + y, where y must be included in comp(a) and comp(c - a). Since the choice of y will only affect the term (d - x) of (12), which we want to be as large as possible to maximize our chance of having (d - x) /= {}, it is best to take y = {}. Thus if (12) has a solution it has the solution

```
  b + (c - a).
```

Therefore a solution will exist if and only if

```
 a * b = {} & (c - a) + (d - b) /= {}.
```

These conditions, like (12), involve one set equality and one inequality, so that inductive treatment of the n variable case corresponding to (12) is possible. For example, we can consider

```
(13)  (EXISTS x,y | (a++*x*y + a+-*x-y+ a-+*y-x + a---x-y={})      & (b++*x*y + b+-*x-y + b-+*y-x + b--x-y) /= {
```

The inner existential of this can be written as the case of (12) in which

```
 a = a++*x + (a-+-x), b = a+-*x + (a--x),      c = b++*x + (b-+-x), d = b+-*x + (b--x)
```

and so has a solution if and only if

```
  (a++*x + (a-+-x)) * (a+-*x + (a--x)) = {} and      ((b++ - a++)*x + ((b-+ - a-+)-x)) * ((b+- - a+-)*x + ((b-- - a-
```

It follows that (13) is equivalent to

(14) (EXISTS x | (a_{++}*a_{+-}*x + (a_{-+}*a_{--} - x) = {} & ((b_{++} - a_{++})*(b_{+-} - a_{+-})*x + ((b_{-+} - a_{-+})*(b_{--} - a_{--})

and hence, applying the solution of (12) once more, has a solution if and only if

a_{++}*a_{+-}*a_{-+}*a_{--} = {} and ((b_{++} - a_{++})*(b_{+-} - a_{+-}) -a_{++}*a_{+-}) * ((b_{-+} - a_{-+})*(b_{--} - a_{--}) - a_{-+}*a_{--}) /= {}.

Moreover, if (13) has a solution at all, it has the solution

(15) x_0 = a_{-+}*a_{--} + (((b_{++} - a_{++})*(b_{+-} - a_{+-})) - (a_{++}*a_{+-}))

if (13) is solvable at all, from which a value for y can be calculated as follows. Substitute x_0 into (13), getting

(13) (a_{++}*x_0*y + a_{+-}*x_0-y + a_{-+}*y-x_0 + a_{--}-x_0-y={}) & (b_{++}*x_0*y + b_{+-}*x_0-y + b_{-+}*y-x_0 + b_{--}-x_0-y) /= {}).

as the condition that y must satisfy. This is a case of (12), and therefore using the solution b + (c - a) of (12) derived above we have

y = (a_{+-}*x_0 + (a_{--}-x_0)) + ((b_{++}*x_0 + (b_{-+}-x_0)) - (a_{++}*x_0 + (a_{-+}-x_0))).

The common theme of these elementary examples is the progressive elimination of quantifiers. This same method will be generalized below to give a procedure for testing the satisfiability of any Behmann formula.

As another interesting elementary case we can consider quantified formulae built around a single set-theoretic equation $e(x_1,...,x_n)$ = {} but involving no set inequalities. Here we can allow arbitrary sequences of existential and universal quantifiers, and do not always insist that $e(x_1,...,x_n)$ only involve Boolean operators, but suppose that existentially quantified variables only appear as arguments of Boolean operators The simplest case is

(16) (EXISTS x | FORALL y | a_y x + (b_y - x = {})) *eq (EXISTS x | ($Un_y(a_y)$ * x + ($Un_y(b_y)$ - x) = {

Where $Un_y(a_y)$ designates the union of all the set values a_y, etc. Hence, by the above discussion of formula(9), (16) is equivalent to (FORALL y,z | $a_y b_z$) = {}), and has the solution $Un_y(b_y)$ if the truth-value of (16) is 'true'. Similar elementary cases involving more complex sequences of existential and universal quantifiers can be treated in much the same way.

The General Behmann Case

Behmann describes an algorithm for calculating the truth value of any formula quantified over sets and involving only Boolean operators, set inclusion and inequality, the set cardinality operator #S, integer constants, cardinal addition, and inequalities. This can be generalized to a decision procedure for formulae quantified over both sets and cardinals involving all of the operators just mentioned. Such formulae will be called PB-formulae. As noted previously, in considering any existentially quantified PB-formula (EXISTS n | P(n)) or (EXISTS x | P(x)) (where, here and below, n designates a cardinal and x a set) we can assume that P is a disjunction of literal terms. If existentially quantified over an cardinal, such a formula can therefore be written as

(18) (EXISTS n | $\&_{Ik=1}$ (a_k*n >= A_k) & $\&_{Jk=1}$ (b_k*n <= B_k)) & Q

where the a_k and b_k are positive integer constants, the A_k and B_k are valid integer-valued PB-terms, and Q is a valid PB-formula. If existentially quantified over a set, a PB-formula can be written as

(19) (EXISTS x | $\&_{Nk=1}$ $SIGMA_{j=1Mk}$ c_{kj}*#(C_j*x + (D_j - x)) >= A_k) & Q,

where A_k and Q are as before, each c_{kj} is an integer constant, while C_k and D_k are valid set-valued PB-terms.

To see that we only need to consider set-theoretic formulae of the form (19), we can argue much as at the beginning of the preceding section. The only operators on sets allowed in the language PB are union, intersection, complementation, and cardinality, and the only set comparators are a incs b and a *incin b. In this language there are no operators which convert objects of type integer into objects of type set. Since a incs b can be written as b - a = { }, and similarly for a *incin b. Thus we can drop the 'incs' and '*incin' comparators and use equality with the nullset as our only comparator. But a = { } can be written as #a = 0. Thus we can drop the equality comparator for sets also. Let x be the set variable which is quantified in the Behmann formula or subformula (EXISTS x | B) that concerns us. Using the identity

```
(EXISTS x | P or Q) = (EXISTS x | P) or (EXISTS x | Q)
```

as often as necessary, we can suppose without loss of generality that B is a conjunction of comparisons, some negated, and so all having the for t = { } or t/= { }, where the term t that appears is formed using the union, intersection, and complementation operators. Using deMorgan's rules for the complement, the distributivity of union over intersection, and the fact that y * y = y for any set y, t we can rewrite t as the union of three terms $t = t_1*x + (t_2 - x) + t_3$, where t_1, t_2, and t_3 are all set terms not containing the variable x. Then, making use of the fact that

```
t1*x + (t2 - x) + t3 = {}
```

is equivalent to

```
t1*x + (t2 - x) = {} & t3 = {},
```

we can move the x-independent clause t_3 = { } outside the quantifier, leaving us with an existentially quantified conjunction of equalitesand inequalites of jut the form seen in (8), as asserted.

The case of formulae quantified over cardinals has been considered above, leaving us to study the case of formulae quantified over variables representing sets. We handle this by forming all possible intersections H_i of the sets C_k, D_k, and their complements. This gives us a collection $H_1,...,H_R$ of sets. Each of the sets C_k, D_k can then be written as a disjoint union of these H_i:

```
(23) Ck = UNIONj in Gk Hj, Dk = UNIONj in Ek Hj, k = 1...n,
```

where G_k and E_k are subsets of {1,...,R}.

Thus we have

```
Ck*x = UNIONj in Gk Hj*x   and   (Dk - x) = UNIONj in Ek (Hj - x)
```

for k = 1...n, from which we see that (19) constrains only the cardinality of the sets H_j * x and (H_j - x) for j in 1,...,R. This observation allows (19) to be rewritten as

```
(24)    (EXISTS n1,...,nR, m1, ..., mR | (&Rk=1 (nk + mk = #Hk)      & &Nk=1 (SIGMAi=1Mk Cki(SIGMAj in Gi nj + SIGM
```

Once having put (19) into the form (24), we can apply the technique described in the preceding section, using this repeatedly to eliminate the cardinal quantifiers

```
(EXISTS n1,...,nR, m1, ..., mR | ....
```

This will ultimately yield a valid PB-formula equivalent to (19) but containing one less quantifier.

Tarski real arithmetic.

Unquantified theory of Boolean terms, sets, maps, domain, and range, with predicates 'singlevalued', 'one-to-one', and with '#' operator, '+', and integer comparison, 'countable'.

Example:

a + b c & singlevalued(f) & a = range(f) & b = domain(f) & #a = n *imp #c £ n + n

Theory of reals and single-valued continuous functions with predicates 'monotone', 'convex', 'concave', real addition and comparison.

In this section we study the decision problem for a fragment of real analysis, which, besides the real operators +, -, *, and /, also provides predicates expressing strict and non-strict monotonicity, concavity, and convexity of continuous real functions over bounded or unbounded intervals, as well as strict and non-strict comparisons '>' and '>=' between real numbers and functions. Decidability of the decision problem for this unquantified language is demonstrated by proving that if a formula in it is satisfiable, then it has a model in which it function-designating variables are mapped into piecewise combinations of parametrized quadratic polynomial and/or exponential functions, where the parameters are constrained only by conditions expressible in the decidable language of real numbers.

The decision problem we consider is that for an unquantified language which we shall call RMC. This provides two types of variables, namely numerical variables, which we will write as x,y, etc., and function variables, denoted by we will write as f,g, etc.

Syntax of RCM. The language RCM has two types of variables, namely *numerical* variables, denoted by x,y,..., and *function* variables, denoted by f,g,... Numerical and function variables are supposed to range, respectively, over the set Re of real numbers and the set of one-parameter continuous real functions over Re. RCM also provides the numerical constants 0 and 1 and the function constants **0** and **1**.

The language also includes two distinguished symbols, -Infinity and +Infinity, which are restricted to occur only as 'range defining' parameters, as explained in the following definitions.

Numerical terms of RCM are defined recursively as follows:

every numerical variable x,y,... or constant 0,1 is a numerical term;

if t_1,t_2 are numerical terms, then so are (t_1+t_2), (t_1-t_2), $(t_1 * t_2)$, and (t_1/t_2);

if t is a numerical term and f is a function variable or constant, then f(t) is a numerical term.

An extended numerical variable (resp. term) is a numerical variable (resp. term) or one of the symbols -Infinity and +Infinity.

function terms of RCM are defined recursively as follows:

every unary function variable f,g,... or constant **0** and **1** is a function term;

if F_1,F_2 are function terms, then so are (F_1+F_2) and (F_1-F_2). An atomic formula of RCM is an expression having one of the following forms:

$t_1 = t_2$	$t_1 > t_2$
$(F_1 = F_2)_{[E_1,E_2]}$	$(F_1 > F_2)_{[E_1,E_2]}$
$Up(F)_{[E_1,E_2]}$	$Strict\_Up(F)_{[E_1,E_2]}$
$Down(F)_{[E_1,E_2]}$	$Strict\_Down(F)_{[E_1,E_2]}$
$Convex(F)_{[E_1,E_2]}$	$Strict\_Convex(F)_{[E_1,E_2]}$
$Concave(F)_{[E_1,E_2]}$	$Strict\_Concave(F)_{[E_1,E_2]}$),

where t_1,t_2 stand for numerical terms, F_1, F_2 stand for function terms, and E_1, E_2 stand for extended numerical terms such that E_1 /= +Infinity and E_2 /= -Infinity.

A *formula* of RCM is any propositional combination of atomic formulae, constructed using the logical connectives and, or, not, *imp, etc.

Semantics of RCM. Next we define the intended semantics of RCM.

A (real) *assignment* M for the language RCM is a map defined over terms and formulae of RCM in the following way:

Definition of M for RCM-terms.

Mx in Re for every numerical variable x.

M0 = 0, M1 = 1, M(+Infinity) = +Infinity, and M(-Infinity) = -Infinity.

For every function variable f, Mf is a continuous real function over Re.

M**0** and M**1** are respectively the zero function and the constant function of value 1, i.e. (M**0**)(r) = 0 and (M**1**)(r) = 1 for every r in Re.

$M(t_1 @ t_2) = Mt_1 @ Mt_2$, for every numerical term $t_1 @ t_2$, where @ is any of +, -, *, and /.

M(f(t)) = (Mf)(Mt), for every function variable f and numerical term t.

$M(F_1 @ F_2)$ is the real function $(MF_1) @ (MF_2)$, where @ is either of the allowed functional operators + and - $(MF_1) @ (MF_2$ is defined by the condition that$(M(F_1 @ F_2))(r) = (MF_1)(r) @ (MF_2)(r)$ for every r in Re).

Definition of M for RCM-formulae. In the following t_1,t_2 will stand for numerical terms, E_1,E_2 for extended numerical terms, and F_1,F_2 for function terms.

$M(t_1 = t_2)$ = true iff $Mt_1 = Mt_2$.

$M(t_1 > t_2)$ = true iff $Mt_1 > Mt_2$.

$M((F_1 > F_2)_{[E_1,E_2]}$ = true iff either $ME_1 > ME_2$, or $ME_1 <= ME_2$ and $(MF_1)(r) > (MF_2)(r)$ for every r in $[ME_1,ME_2]$. (Here and below we use the interval notation [x,y] even if x = -Infinity and/or y = +Infinity.)

$M((F_1 = F_2)_{[E_1,E_2]})$ = true iff either $ME_1 > ME_2$, or $ME_1 <= ME_2$ and $(MF_1)(x) = (MF_2)(x)$ for every x in $[ME_1,ME_2]$.

$M(Up(F)_{[E_1,E_2]})$ = true (resp. $M(Strict\_Up(F)_{[E_1,E_2]})$ = true) iff either $ME_1 >= ME_2$, or $ME_1 < ME_2$ and the function MF_1 is monotone non-decreasing (resp. strictly increasing) in the interval $[ME_1,ME_2]$.

$M(Down(F)_{[E_1,E_2]})$ = true (resp. $M(Strict\_Down(F)_{[E_1,E_2]})$ = true) iff either $ME_1 >= ME_2$, or $ME_1 < ME_2$ and the function MF_1 is monotone non-increasing (resp. strictly decreasing) in the interval $[ME_1,ME_2]$.

$M(Convex(F)_{[E_1,E_2]})$ = true (resp. $M(Strict\_Convex(F)_{[E_1,E_2]})$ = true) iff either $ME_1 >= ME_2$, or $ME_1 < ME_2$ and the function MF_1 is convex (resp. strictly convex) in the interval $[ME_1,ME_2]$.

$M(Concave(F)_{[E_1,E_2]})$ = true (resp. $M(Strict\_Concave(F)_{[E_1,E_2]})$ = true) iff either $ME_1 >= ME_2$, or $ME_1 < ME_2$ and the function MF_1 is concave (resp. strictly concave) in the interval $[ME_1,ME_2]$.

Logical connectives are interpreted in the standard way; thus, for instance, $M(P_1 \& P_2) = MP_1 \& MP_2$.

Let P be an RCM-formula and let M be an assignment for the language RCM. Note once more that we say that M is a *model* for P iff M(P) = true. If P has a model, then it is *satisfiable,* otherwise it is *unsatisfiable.* If P is true in every RCM-assignment, then P is a *theorem* of RCM. As usual, two formulae are *equisatisfiable* if either both of them are unsatisfiable, or both of them are satisfiable, and the *satisfiability problem* for RCM is the problem of finding an algorithm which can determine whether a given RCM-formula is satisfiable or not. Such an algorithm is given below. Here are a few examples of statements which can be proved automatically using this decision algorithm.

A strictly convex curve and a concave curve defined over the same interval can meet in at most two points.

This statement can be formalized in RCM as follows:

```
(Strict_Convex(f)[E1,E2] & Concave(g)[E1,E2] & &i=1..3(f(xi) = g(xi)) & &i=1..3(E1 <= xi & xi <= E2))        *imp
```

A second example is as follows.

Let g be a linear function. Then a function f defined over the same domain as g is strictly convex if and only if f + g is strictly convex.

Introduce a predicate symbol $Linear(f)_{[x_1,x_2]}$ standing for

$Convex(f)_{[x_1,x_2]}$ & $Concave(f)_{[x_1,x_2]}$.

Note that if M is a real assignment for RCM, then $M(Linear(f)_{[x_1,x_2]})$ = true if and only if the function Mf is linear in the interval $[ME_1,ME_2]$.

It is plain that the proposition shown above is equivalent to the following formula:

```
Linear(g)x1,x2 *imp     (Strict_Convex(f)[x1,x2] *eq Strict_Convex(f + g)[x1,x2]).
```

The following is a somewhat more interesting example.

Let f and g be two real functions which take the same values at the endpoints of a closed interval [a,b]. Assume also that f is strictly convex in [a,b] and that g is linear in [a,b]. Then f(c) < g(c) holds at each point c interior to the interval [a,b].

This proposition can be formalized in the following way in the language RCF.

```
(Strict_Convex(f)[x1,x2] & Linear(g)[x1,x2]     & f(x1) = g(x1) & f(x2) = g(x2)     & x2 > x & x > x1) *imp (g(x
```

Preparing a set of RCM statements for satisfiability testing. We shall prove the decidability of formulae of RCM using a series of satisfiability-preserving steps which reduce the satisfiability problem for RCM to a more easily decidable satisfiability problem for an unquantified set of statements involving real numbers only.

We begin by noting that the decidability problem for RCM can be reduced in the usual way to that for statements which are conjunctions of basic literals, where each conjunct must have one of the following forms:

```
x = y + w, x = y * w, x > y, y = f(x),      (f = g + h)[z1,z2], (f > g)[z1,z2],      (-)Up(f)[z1,z2], (-)Strict_Up(
```

Here x,y,w,w_1,w_2 stand for numerical variables or constants, z_1,z_2 for extended numerical variables (where z_1 is not equal to +Infinity nor z_2 to -Infinity), f,g,h for function variables or constants, and the expression (-)A denotes both the un-negated and negated literals A and (not A). Note that reduction of the full set of constructs allowed in RCM to the somewhat more limited set seen above requires application of the following equivalences to eliminate subtraction, division, and various negated cases:

```
(f1 = f2 - f3)[z1,z2] *eq (f2 = f1 + f3)[z1,z2]     (f1 = f2)[z1,z2] *eq (f1 = f2 + 0)[z1,z2]     t1 = t2 - t3 *eq t2 = t
```

It is also easy to eliminate the negated forms of the predicates Up, Down, Convex, and Concave and the negated forms of the strict versions of these predicates. For example, to re-express the assertion (not $Up(f)_{[z_1,z_2]}$), we can simply introduce two new variables x and y representing real numbers, and replace (not $Up(f)_{[z_1,z_2]}$) by

```
x > y & z2 >= x & y >= z1 & f(y) > f(x).
```

We leave it to the reader to verify that something quite similar to this can be done for the negations of all the relevant predicates.

In further preparation for what follows, we define a variable x appearing in one of our formulae to be a *domain variable* if it appears either in a term y = f(x) or as one of the z_1 or z_2 in a term like $(f = g + h)_{[z_1,z_2]}$, $Up(f)_{[z_1,z_2]}$, or $Strict\_Concave(f)_{[z_1,z_2]}$. We can assume without loss of generality that for each such domain variable and for every function variable f there exists a variable y for which a conjunct y = f(x) appears in our collection. (This y simply represents the value of f on the real value of x). Indeed, if there is no such clause for f and x, we can simply introduce a new variable y and add y = f(x) to our collection of conjuncts. It should be obvious to the reader that this addition preserves satisfiability.

Next we make the following observation. Let $x_1,...,x_r$ be the domain variables which appear in our set of conjuncts. If a model M of these conjuncts exists, then $Mx_1,...,Mx_r$ will be real numbers, of which some may be equal, and where the distinct values on this list will appear in some order along the real axis, and so divide it into subintervals. Each possible ordering of $Mx_1,...,Mx_r$ will correspond to some permutation of $x_1,...,x_r$ which puts $Mx_1,...,Mx_r$ into increasing order, and so to some collection of conditions $x_i < x_{i+1}$ or $x_i = x_{i+1}$, which need to be written for all i = 1,..,r - 1. Where conditions $x_i = x_{i+1}$ appear, implying that two or more domain variables are equal, we identify all these variables with the first of them, and then also add statements y = z + 0 for any variables y,z appearing in conjuncts $y = f(x_i)$, $z = f(x_j)$ involving domain variables that have been identified. It is understood that all possible orders of $Mx_1,...,Mx_r$, and all possible choices of inequalities $x_i < x_{i+1}$ or equalities $x_i = x_{i+1}$ must be considered. If any of these alternatives leads to a set of conjuncts which can be satisfied, then our original set of conjuncts can be satisfied, otherwise not. This observation allows us to focus on each of these orderings separately, and so to consider sets of conjuncts supplied with clauses x > y which determine the relative order of all the domain variables that appear.

Note that this last preparatory step can be expensive, so special care must be taken in implementing it. Nevertheless it clearly can be implemented, and after it is applied we are left with a set of conjuncts satisfying the two following conditions. (i) each conjunct in the set must have one of the following forms:

```
(*)   x = y + w, x = y * w, x > y, y = f(x),      (f = g + h)[z1,z2], (f > g)[z1,z2],      Up(f)[z1,z2], Strict_Up(f
```

(ii) the collection $x_1,...,x_r$ of domain variables present in this set of is arranged in a sequence for which a conjunct $x_i < x_{i+1}$ is present for all i = 1,..,r - 1.

Removal of function literals. Having simplified the satisfiability problem for RCM in the manner just described, we will now show how to reduce it to a solvable satisfiability problem involving real numbers only. We use the following idea. If a set of conjuncts of the form (1) has a model M, the domain variables $x_1,...,x_r$ which appear in it will be represented by real numbers $Mx_1,...,Mx_r$ which occur in increasing order. Consider a conjunct like $Up(f)_{[x,y]}$ or $Convex(f)_{[x,y]}$, where for simplicity we first suppose hat neither of x and y is infinite. Then we must have $x = x_j$ and $y = x_k$ for some j and some j > i. For f to be nondecreasing in the range $[x_j,x_k]$, it is necessary and sufficient that it should be nondecreasing in each of the subranges $[x_i,x_{i+1}]$ for each i from j to k - 1. For f to be convex in the range $[x_j,x_k]$, it is necessary and sufficient that it should be convex in the overlapping set of ranges $[x_i,x_{i+2}]$ for each i from j to k - 2 or, if k = j + 1, convex in $[x_j,x_{j+1}]$. (The proof of this elementary fact is left to the reader.) For f to be nondecreasing in $[x_i,x_{i+1}]$ it is necessary that we should have $f(x_i) <= f(x_{i+1})$, and, if f is piecewise linear with corners only at the points x_i this is also sufficient. Hence the necessary and sufficient condition for such a nondecreasing function to exist is

$$f(x_j) <= f(x_{j+1}) \&...\& f(x_{k-1}) <= f(x_k)$$

For f to be convex in $[x_i,x_{i+2}]$ it is necessary that the value $f(x_{i+1})$ should lie below or on the line connecting the points $[x_i,f(x_i)]$ and $[x_{i+2},f(x_{i+2})]$. This condition can be written algebraically as

$$f(x_i)*(x_{i+1} - x_i) + f(x_{i+2})*(x_{i+2} - x_{i+1}) <= f(x_{i+1}).$$

Conjoining all these conditions gives

$$f(x_j)*(x_{j+1} - x_j) + f(x_{j+2})*(x_{j+2} - x_{j+1}) <= f(x_{j+1}) \& ... \& f(x_k)*(x_{k-1} - x_{k-2}) + f(x_k)*(x_k - x_{k-1}) <= f(x$$

If f is piecewise linear with corners only at the points x_i this is also sufficient. Hence the conjunction just shown is necessary and sufficient for such a convex function to exist. Plainly the same remarks carry over to the nonincreasing and convex cases if we simply reverse the inequalities appearing in the last few conditions displayed.

In the strictly increasing case the necessary conditions become

$$f(x_j) < f(x_{j+1}) \&...\& f(x_{k-1}) < f(x_k)$$

A piecewise linear function satisfying these conditions is also strictly increasing, so these conditions are those necessary and sufficient for a function with the given values, and strictly increasing over the range $[x_j,x_k]$, to exist. For there to exist a strictly convex function in this range,the conditions

$$f(x_j)*(x_{j+1} - x_j) + f(x_{j+2})*(x_{j+2} - x_{j+1}) < f(x_{j+1}) \& ... \& f(x_k)*(x_{k-1} - x_{k-2}) + f(x_k)*(x_k - x_{k-1}) < f(x_{k-1}).$$

are necessary. However in this case they are not quite sufficient, since a piecewise linear function satisfying these conditions is not yet strictly convex, since the slope of such a function is constant, rather than increasing, in each of it intervals of linearity. But it is easy to correct this, simply by passing to functions which are piecewise quadratic (still with corners only at the points x_i), rather than linear. Such function are determined by their end values $f(x_i)$ and $f(x_{i+1})$ and by one auxiliary value f(x) at any point x interior to $[x_i,x_{i+1}]$. It is convenient to let x be the midpoint of the interval $[x_i,x_{i+1}]$. Then for f to be convex it is necessary that $f(x_i) + f(x_{i+1}) <= f(x) + f(x)$, and for f to be strictly convex it is necessary that $f(x_i) + f(x_{i+1}) < f(x) + f(x)$. (Here and below, the same remarks apply, with appropriate changes of sign, to the concave and strictly concave cases also.) If the function f is known to be nondecreasing in the interval $[x_i,x_{i+1}]$ (because $[x_i,x_{i+1}]$ is included in some interval $[x_j,x_k]$ for which a statement $Up(f)_{[x_j,x_k]}$ appears among our conjuncts, we must also write the conditions $f(x_i) <= f(x)$ and $f(x_{i+1}) <= f(x)$. Similarly, if f is known to be nonincreasing we must write the conditions $f(x_i) >= f(x)$ and $f(x_{i+1}) >= f(x)$. Note that if f is known to be nondecreasing and strictly convex in an interval $[x_i,x_{i+1}]$, the strict inequality $f(x_i) < f(x_{i+1})$ follows, since this is implied by the three known conditions $f(x_i) <= f(x)$, $f(x) < f(x_{i+1})$, and $f(x_i) +$

$f(x_{i+1}) < f(x) + f(x)$. In all such cases we will therefore replace $f(x_i) <= f(x_{i+1})$ by $f(x_i) < f(x_{i+1})$ in our set of conjuncts, and similarly for intervals in which f is known to be strictly concave and monotone nonincreasing (Likewise in the corresponding cases in which a conjunct $Strict\_concave(f)_{[x_j,x_k]}$ is present for some interval $[x_j,x_k]$ including $[x_i,x_{i+1}]$). After these supplementary replacements, we can be sure that f must be strictly monotone in every interval $[x_i,x_{i+1}]$ in which it needs to be both monotone and strictly convex (or concave).

The necessary conditions introduced in the preceding paragraph force all the required convexity properties to hold in the whole finite range $[x_{1i},x_r]$ for piecewise linear functions having $x_1,...,x_r$ as their only corners, except that these functions will be linear rather than strictly convex or concave in the intervals between these corners,even if strict concavity or convexity is required. To fix this we can simply add a very small quadratic polynomial vanishing at the two endpoints of the interval to the linear function we initially have in each such interval. The small constant c should be chosen to be negative if strict convexity is required, but positive if strict concavity is required. Since this sign will always be the same as that of the difference $v = f(x_i) + f(x_{i+1}) - 2*f(x)$, we can always take $c = c'*v$ where c' is any sufficiently small positive constant. Note that this will never spoil either the monotonicity or strict monotonicity of f in the interval affected, since if c' is small enough strict monotonicity will never be affected, while the adjustments described in the preceding paragraph ensure that strict monotonicity rather than simple monotonicity will be known in every interval in which strict convexity or concavity is also required.

It follows that the simple, purely algebraic inequalities on the points $x_1,...,x_r$, the intermediate midpoints x, and the corresponding function values $f(x_j)$ and f(x) derived in the two preceding paragraphs are both necessary and sufficient for the existence of a continuous function satisfying all the monotonicity and convexity conditions from which they were derived, at least in the finite interval $[x_1,x_r]$. We shall now extend this result to the two infinite end intervals [-Infinity,x_1] and [x_r,+Infinity], thereby deriving a set of purely algebraic conditions fully equivalent to the initially given monotonicity and convexity conditions. It will then follow immediately that replacing the monotonicity and convexity conditions by the algebraic conditions derived from them replaces our initial set of conjuncts by an equisatisfiable set.

Of the two infinite end intervals, first consider [x_r,+Infinity]. Choose the two auxiliary points $x_r + 1$ and $x_r + 2$ in this interval. Then we can write monotonicity and convexity conditions as above for the values $f(x_{r-1})$, $f(x_r)$, $f(x_r + 1)$, and $f(x_r + 2)$. A previously, if f is both monotone, nondecreasing, and strictly concave or convex in [x_r,+Infinity], it follows that $f(x_r) < f(x_r + 1)$ and $f(x_r + 1) < f(x_r + 2)$, so we replace the monotonicity inequalities $f(x_r) <= f(x_r + 1)$ and $f(x_r + 1) <= f(x_r + 2)$ by their strict versions in this case. Then we can take f to be piecewise linear with corners at the points x_r, $x_r + 1$, and $x_r + 2$, extending f to the infinite range [$x_r + 2$,+Infinity] with the same slope that it has on the interval $[x_r + 1,x_r + 2]$. This definition satisfies all the monotonicity and convexity conditions already present, except for that of strict convexity (or concavity) in the intervals $[x_r,x_r + 1]$, $[x_r + 1,x_r + 2]$, and $[x_r + 1,x_r + 2]$ if this is required. But, as in the cases considered above, these strict conditions can be forced (in $[x_r,x_r + 1]$ and $[x_r + 1,x_r + 2]$) by adding a quadratic term $c*x^2 + ...$, where the coefficient c is $c'*(f(x_i) + f(x_{i+1}) - 2*f(x))$ and c' is extremely small and positive. In the interval [$x_r + 2$,+Infinity] we add the decaying exponential c*exp(-x) instead. This has the same convexity properties as $c*x^2 + ...$, and, for c sufficiently small, is also without effect on the monotonicity properties of every strictly monotone linear function.

We leave it to the reader to verify that the same argument applies to the second end-interval [-Infinity,x_1]. It follows that the conditions on the points $x_1,...,x_r$, the intermediate midpoints x, and function values $f(x_j)$ and f(x) that we have stated are necessary and sufficient for the existence of a continuous function having these values at the stated points and all the monotonicity and convexity properties from which these conditions were derived.

Since all the piecewise quadratic and exponential functions f of which we make use are determined linearly by their values y = f(x) at points x which appear explicitly in our algorithm, any condition of the form f(x) = g(x) + h(x) which appears in our initial collection of conjuncts can be replaced by writing the corresponding conditions f(x) = g(x) + h(x) for all of the domain variables appearing in these conjuncts.

The following result summarizes the results obtained in the last few paragraphs, putting them into an obviously programmable form.

Let a collection of conditions of the form (*) be given, and suppose that this satisfies the conditions (i) and (ii) found in the paragraph containing (*). Introduce additional variables x'_i satisfying $x'_i = (x_i + x_{i+1})/2$ for each i between 1 and r - 1. and also x'_r and x'_{r+1}, x'_1 and x'_0 satisfying $x'_r = x_r + 1$, $x'_{r+1} = x_r + 2$, $x'_1 = x_1 - 1$, $x'_0 = x_1 - 2$. For each variable x_j and x'_j in this extended set, and each function symbol f appearing in the set (*) of conjuncts for which there exists no conjunct of the form $y_j^f = f(x_j)$ or $y'_j{}^f = f(x'_j)$, introduce a new variable to play the role of y_j^f or $y'_j{}^f$, along with the missing conjunct. Then replace all the conjuncts appearing in lines 2 thru 6 of (*) in the following ways:

(a) replace each conjunct $(f = g + h)_{[z_1,z_2]}$ by the conditions $y_j^f = y_j^g + y_j^h$ and $y'_j{}^f = y'_j{}^g + y'_j{}^h$, for all x_j and x'_j belonging to the interval $[z_1,z_2]$.

(b) replace each conjunct $(f > g)_{[z_1,z_2]}$ by the conditions $y_j^f > y_j^g$ and $y'_j{}^f > y'_j{}^g$, for all x_j and x'_j belonging to the interval $[z_1,z_2]$.

(c) replace each conjunct $Up(f)_{[z_1,z_2]}$ (resp. $Strict\_Up(f)_{[z_1,z_2]}$) by the conditions $y_j^f <= y'_j{}^f$ and $y'_j{}^f <= y_{j+1}{}^f$ (resp. $y_j^f < y'_j{}^f$ and $y'_j{}^f < y_{j+1}{}^f$), for all subintervals $[x_j, x_{j+1}]$ of the interval $[z_1,z_2]$. (A slight adaptation of this formulation, which we leave to the reader to work out, is needed in the case of the two infinite end-intervals [-Infinity,x_1] and [x_r,+Infinity].)

(d) replace each conjunct $Down(f)_{[z_1,z_2]}$ (resp. $Strict\_Down(f)_{[z_1,z_2]}$) by the conditions $y_j^f >= y'_j{}^f$ and $y'_j{}^f >= y_{j+1}{}^f$ (resp. $y_j^f > y'_j{}^f$ and $y'_j{}^f > y_{j+1}{}^f$), for all subintervals $[x_j, x_{j+1}]$ of the interval $[z_1,z_2]$. (A slight adaptation of this formulation, which we leave to the reader to work out, is needed in the case of the two infinite end-intervals [-Infinity,x_1] and [x_r,+Infinity].)

(e) replace each conjunct $Convex(f)_{[z_1,z_2]}$ (resp. $Strict\_Convex(f)_{[z_1,z_2]}$) by the conditions

```
yi*(xi+1 - xi) + yi+2*(xi+2 - xi+2) <= yi+1
```

and

```
yi*(x'i - xi) + yi+1*(xi+1 - x'i) <= y'i)
```

(resp. the same conditions, but will the inequality signs <= changed to strict inequality signs '<'), the first replacement being made for each subinterval $[x_j, x_{j+2}]$ of the interval $[z_1,z_2]$, and the second for each subinterval $[x_j, x_{j+1}]$ of the interval $[z_1,z_2]$. (This formulation must be adapted in the manner sketched in the previous subsection to the cases of the two infinite end-intervals [-Infinity,x_1] and [x_r,+Infinity]. We leave to the reader to formulate the required details.) Moreover, if a subinterval $[x_j, x_{j+1}]$ of a $[z_1,z_2]$ for which strict convexity is asserted is also one to which the predicate $Up(f)_{[x_j, x_{j+1}]}$ or $Down(f)_{[x_j, x_{j+1}]}$ applies in virtue of a replacement (c) or (d), change the unstrict inequalities replacing these latter predicates to strict inequalities.

(f) replace each conjunct $Concave(f)_{[z_1,z_2]}$ (resp. $Strict\_Concave(f)_{[z_1,z_2]}$) by the conditions

```
yi*(xi+1 - xi) + yi+2*(xi+2 - xi+2) >= yi+1
```

and

```
yi*(x'i - xi) + yi+1*(xi+1 - x'i) >= y'i)
```

(resp. the same conditions, but will the inequality signs >= changed to strict inequality signs '>'), the first replacement being made for each subinterval $[x_j, x_{j+2}]$ of the interval $[z_1,z_2]$, and the second for each subinterval $[x_j, x_{j+1}]$ of the interval $[z_1,z_2]$. (This formulation must be adapted in the manner sketched in

the previous subsection to the cases of the two infinite end-intervals [-Infinity,x_1] and [x_r,+Infinity]. We leave to the reader to formulate the required details.) Moreover, if a subinterval [x_j, x_{j+1}] of a [z_1,z_2] for which strict convexity is asserted is also one to which the predicate $Up(f)_{[x_j, x_{j+1}]}$ or $Down(f)_{[x_j, x_{j+1}]}$ applies in virtue of a replacement (c) or (d), change the unstrict inequalities replacing these latter predicates to strict inequalities.

These replacements convert our original set (*) of conjuncts into an equisatisfiable set of purely algebraic conditions.

To conclude our work we need an algorithm capable of determining whether the set of algebraic conditions (all of which are either linear or quadratic) to which the foregoing algorithm reduces our original set of conjuncts is satisfiable or unsatisfiable. Since this problem is a special case of the decision algorithm for Tarski's quantified algebraic language of real numbers, such an algorithm certainly exists. This observation completes our proof of that the language RCM has a decidable satisfiability problem.

A final example. To make the foregoing considerations somewhat more vivid, consider the way in which the proof of the third sample proposition listed above results from our algorithm, which can just as easily be used to prove it in the following generalized form.

If f and g are two real functions which take the same values at the endpoints of a closed interval [a,b]. Assume also that f is strictly convex in [a,b] and that g is concave in [a,b]. Then f(c) < g(c) holds at each point c interior to the interval [a,b].

This can be formalized as follows:

```
(Strict_Convex(f)[x1,x2] & Concave(g)[x1,x2]      & f(x1) = g(x1) & f(x2) = g(x2)      & x2 > x & x > x1) *imp (g(x
```

In this case the domain variables are x_1,x_2, and x, and it is clear that the only order in which they need to be considered is x_1,x,x_2. The negation of our theorem is then the conjunction of $Strict\_Convex(f)_{[x_1,x_2]}$, $Concave(g)_{[x_1,x_2]}$, $f(x_1) = g(x_1)$, $f(x_2) = g(x_2)$, and $f(x) > g(x)$. The rules stated above replace the first two conjuncts by the algebraic conditions

```
f(x1)*(x - x1) + f(x2)*(x2 - x) < f(x)
```

and

```
g(xi)*(xi+1 - xi) + g(xi+2)*(xi+2 - xi+1) >= f(xi+1).
```

The other algebraic conditions generated are not needed; these two conditions, together with the facts $f(x_1) = g(x_1)$ and $f(x_2) = g(x_2)$ plainly imply that $f(x) > g(x)$, which is inconsistent with $g(x) >= f(x)$, an inconsistency which the Tarski algorithm alluded to above will detect.

Various parts of elementary point-set topology.

Implications among multiparameter polynomial equations.

Systematic use of calculus can often decide the solvability of systems of polynomial or elementary-function inequalities.

Others.

Still more decidable quantified languages

Theory of algebraically closed, real-closed, p-adic, and finite fields

Theory of commutative groups

Theory of purely multiplicative integer arithmetic

Integers and sets of integers with successor

Integers and finite sets of integers with successor

Countable totally ordered sets and their subsets

Theory of well-ordered sets

Decidable fragments of arithmetic

Decidable fragments of arithmetic, for example statements of the form

$$(\text{EXISTS } x_1 \text{ in } Z,\ x_2 \text{ in } Z,\ldots,x_n \text{ in } Z \mid D(x_1,x_2,\ldots,x_n)) = 0$$

where D is a Diophantine polynomial of degree 2

Various forms of Boolean algebra

Semi-decidable sublanguages of set theory

The Tableau Method

The Davis-Putnam method for testing propositional satisfiability attains efficiency by making all possible 'deterministic' inferences (using clauses containing just one propositional symbol) before making any 'nondeterministic' inference (by exploring both possible truth values of some propositional symbol, when no more clauses containing just one propositional symbol remain. The *tableau method* to be described in this section generalize this approach, first to statements in the unquantified language MLSS discussed earlier, and then to various extensions of MLSS.

Given an initial set of clauses, the tableau method finds their consequences transitively. The strategy used resembles that which we have already seen in the Davis-Putnam case. The deduction rules used for this are segregated into two classes: those which act 'deterministically' (like the use of a singleton clause in the Davis-Putnam algorithm), and those which act 'nondeterministically' (like the choice of a singleton to be given an arbitrary truth-value when there exists no singleton clause in the Davis-Putnam algorithm). This implicitly assumes that completion of a set of clauses using only the first class of rules will, in polynomial time, generate a relatively small clause set, so that exponentially growing costs will result only from nondeterministic application of the second, smaller, nondeterministic class of rules. This makes it reasonable to apply the deterministic rules as long as possible, checking for contradictions which might terminate many paths of expansion before more than a few nondeterministic rules need to be applied. In this strategy, we only apply a nondeterministic rule when no deterministic rule remains applicable. This strategy is also basic to the Davis-Putnam algorithm.

In the case of MLSS, which for convenience we now consider in a version allowing the operators '+', '*', '-', {x}, and the relators 'in', 'incs', and '=', we work with two sets of propositions, one of which collects all currently available propositions of the forms

```
a = b, a incs b, a in b,   not(a = b), not(a incs b), not(a in b),
```

and the other of which collects all propositions of the forms

```
a = b + c, a = b * c, a = b - c, a = {b}.
```

Initially these two collections contain propositions representing the set of statements to be tested for satisfiability. A statement 'b in a' is added for each statement 'a = {b}' initially present.

The initial collections of statements defined in this way are progressively modified as deductions are made. The deduction process will sometimes proceed deterministically, but sometimes branch nondeterministically, i.e. open a path of exploration which may need to be abandoned if it ends in a contradiction. Only statements of the form 'a in b', 'not(a in b)', and 'a = b' are added in the course of deduction. However, the variables appearing in some of the other statements may change as equalities are deduced. Exploration of a branch fails immediately whenever two directly opposed statements 'a in b' and 'not (a in b)' are detected.

The working of the algorithm can be clarified by considering the way in which it will build a model of the set of statements with which it is working if one exists. This is done by examining the collection of all membership relationships 'a in b' deduced, first making sure that this contains no cycles (which are impossible if a model exists). If this check is passed we assign distinct sets of sufficiently large cardinality to all the variables which do not appear on the right of any deduced relationship 'a in b', and then process all the 'other variables in topologically sorted order of the membership relation 'a in b', modeling each b as the collection of all M(a) for which a statement relationship 'a in b' has been deduced.

Equality is handled in a special way, which ensures that all statements a = b are modeled properly, and that all the operations b + c, b * c, b - c are defined uniquely by their arguments. Specifically, whenever a = b has been deduced we choose one of a and b as a representative of the other, all of whose occurrences are then replaced by occurrences of the representative. This process may identify the right hand sides of some statements of the form a = b + c, a = b * c, a = b - c, a = {b}; whenever this happens we immediately deduce that the left-hand sides are also equal. If a model is subsequently found we give each variable replaced in this way the same value as its representative. The rules stated below will sometimes introduce new variables. These variables can only appear in statements of the form 'x in b' and 'not(x in b)', and only on the left of such statements. It will follow that whenever an equality 'a = b' is deduced, one of a and b must be a variable initially present; in choosing representatives we always choose such a variable.

For the model-building procedure described above to work, we must be sure that every statement 'a incs b', 'not (a incs b)', 'not (a = b)', 'a = b + c', 'a = b * c', 'a = b - c', and 'a = {b}' is properly modeled. To this end, we make the following deductions:

> 'x in a' is deduced whenever 'x in b' and 'a incs b' are present.
>
> A new variable x and statements 'x in b', 'x notin a' are set up whenever 'not (a incs b)' is present.
>
> 'x in a' is deduced whenever 'x in b' and 'a = b + c' are present.
> 'x in a' is deduced whenever 'x in c' and 'a = b + c' are present. These two rules ensure that in the model eventually constructed, M(a) is no smaller than M(b) + M(c).
>
> 'x in b' and 'x in c' are deduced whenever 'x in a' and 'a = b * c' are present. This ensures that in the model eventually constructed, M(a) is no larger than M(b) * M(c).
>
> whenever the statement 'x in s' has been deduced, and a statement 's = {t}' is present, the statement 'x = t' is deduced. This ensures that the model of s can contain at most one element.
>
> 'x in b and x notin c' is deduced whenever 'x in a' and 'a = b - c' are present. This ensures that in the model eventually constructed, M(a) is no larger than M(b) - M(c).

The set of rules stated above are all deterministic, but a few nondeterministic rules are required also. These are as follows.

> If 'x in a' and 'not(y in a)' have both been deduced, we deduce an inequality 'x /= y', setting this up as an alternation (x nincs y) or (y nincs x). This ensures that x and y will have different models, implying that all statements 'not (y in a)' are correctly modeled. It is only necessary to do this when both x and y belong to the collection of variables initially present, since, as previously explained, variables not in this collection will always be assigned distinct sets as models.
>
> An alternation 'x in b or x in c', both of whose branches may need to be explored, is set up whenever 'x in a' and 'a = b + c' are present. This ensures that in the model eventually constructed, M(a) is no larger than M(b) + M(c).
>
> Similarly, an alternation 'x in a or x notin c' is set up whenever 'x in b' and 'a = b * c' are present. Likewise an alternation 'x in a or x notin b' is set up whenever 'x in c' and 'a = b * c' are present. This ensures that in the model eventually constructed, M(a) is no smaller than M(b) * M(c).
>
> Similarly, an alternation 'x in a or x in c' is set up whenever 'x in b' and 'a = b - c' are present. This ensures that in the model eventually constructed, M(a) is no smaller than M(b) - M(c).

These rules are sufficient, but to accelerate discovery of contradictions (which can cut off a branch of exploration before multiple alternations need to be resolved, an exponentially expensive matter when necessary) all possible deterministic deductions are made. These are:

> 'x notin b' is deduced whenever 'x notin a' and 'a incs b' are present.
>
> 'x notin b' is deduced whenever 'x notin a' and 'a = b + c' are present.
>
> 'x notin c' is deduced whenever 'x notin a' and 'a = b + c' are present.
>
> 'x notin a' is deduced whenever 'x notin b' and 'a = b * c' are present.
>
> 'x notin a' is deduced whenever 'x notin c' and 'a = b * c' are present.
>
> 'x notin a' is deduced whenever 'x notin b' and 'a = b - c' are present.
>
> 'x notin a' is deduced whenever 'x in c' and 'a = b - c' are present.

To further clarify the style of proof discussed above, we consider its application to the example

```
not(({c} = c + d) *imp (c = {} & d = {c}))
```

which, decomposed propositionally and then initialized in the manner described above, breaks down into the two cases

```
e = {c}, c in e, e = c + d, not(c = {})
```

and

```
e = {c}, c in e, e = c + d, not(d = {c}).
```

In the first of these two cases we progressively deduce f in c, f in e, f = c, c in c, leading to a contradiction. The second case splits nondeterministically into the two cases

```
not (d incs e)  and  not (e incs d).
```

In the first of these cases we deduce f in e, not(f in d), f = c, c in c, leading to a contradiction as before. In the second case we deduce not(f in e), f in d, f in e, leading again to a contradiction and so eliminating the last possible case.

The preceding discussion assumes that the collection of statements with which we deal has been resolved at the propositional level before the analysis described begins. However, it may often be better to integrate the propositional and the set-theoretic levels of exploration, so as to allow the impossibility of a set-theoretic exploration to rule out a whole family of propositional branches which otherwise might need to be explored individually before their (predictable) failure became apparent. This can be done as follows. By introducing additional intermediate variables we can suppose that all the atomic subformulae of our formulae have simple forms like a incs b, a = b, a in b (and their negatives), along with statements like a = b + c, a = b * c, a = b - c, a = {b}. Propositional calculus rules can be used in the standard way to write all the top-level propositions in our set as disjunctions like

```
(*) a incs b or a = b or a in b or ...
```

in which some of the atoms present may be negated. We now arrange all the propositions (*) in order of increasing number of their atomic parts and work through them in the following way. Starting with the first proposition F, we select its atomic parts A in order for processing. Each such A is, when selected, added to our collection AP of atomic propositions, where it will remain unless/until the branch of exploration opened by this addition fails. If such a branch of exploration fails, the atomic formula A that opened the branch is removed and its negative (which will now remain permanently) is added to AP. At the same time the next atomic formula A' after A is selected and added to AP. If there is no such A', then the branch of exploration opened by the selection of A fails; if A belongs to the first formula F, then all possibilites have failed and the given set of propositions is unsatisfiable.

Once a branch of exploration is opened we make all possible deterministic and nondeterministic deductions from it, in the manner described above. Eventually either the branch will fail, or run out of deductions to make. In the latter case we examine all the formulae (*) following the F containing the A that opened to current branch of exploration. Formulae containing atoms B present in or our deduced collection of atoms are bypassed (since they must be satisfied already, and so tell us nothing new). The negatives of all such B are removed from the formulae still to be processed (since these propositions are known to be false; note that this duplicates a deterministic deduction step of the Davis-Putnam algorithm). If any one of these formulae is thereby made null, the branch of exploration opened by A fails. Otherwise the formulae following F are rearranged in order of increasing number of their remaining atomic parts, and we move on to select an atomic subformula of the next formula F' following F.

We illustrate this integrated style of proof, again using the example

```
not(({c} = c + d) *imp (c = {} & d = {c}))
```

whose negative is now expressed as the following set of three clauses

```
e = {c}, c in e, e = c + d, (not(c = {}) or not(d = e).
```

A branch of exploration is opened by adding not(c = { }) to the first three clauses, giving the deductions

```
e = {c}, c in e, e = c + d, not(c = {}), f in c, f in e,          f = c, c in c
```

which fails. The alternate path then begins with

```
e = {c}, c in e, e = c + d, c = {},      (not(d incs e)) or (not(e incs d)),
```

from which we deduce

```
e = {c}, c in e, e = c + d, c = {},     (not(d incs e)) or (not(e incs d)), e = d,
```

and so

```
e = {c}, c in e, c = {}, not(e incs e),f in e, not(f in e),
```

which fails, confirming the validity of our original formula.

Tableau-based proof approaches have the interesting property that if they are sound, and even if they are not complete (so that there can exist contradictory sets of clauses which they are not able to extend to an obvious contradiction), any family of statements found to be contradictory because all branches of exploration fail really is unsatisfiable. This is because the tableau method implicitly makes and then discharges a sequence of suppositions, every one of which has led to a contradiction. So systems of tableau rules can be used even if they are incomplete as long as they converge, and, as a matter of fact, can be used in any individual case whose exploration does terminate, even if the system does not terminate for every possible input. All that is necessary is that such systems should be sound. Therefore if we use a fixed, table-driven tableau code, we can be certain of the rigor of its deductions as long as we know that all rules entered into each driving table are sound. This will necessarily be the case if all such rules are instances of universally quantified, previously proved theorems. For example, once cons, car, and cdr have been given their set-theoretic definitions and it has been proved that

```
(FORALL x,y,u,v | (car([x,y]) = x & cdr([x,y]) = y     & (([x,y] = [u,v]) *imp (x = u & y = v)))
```

we can be sure that the tableau rules derived from this statement are sound, and so we can add them to the table driving a generic tableau code.

A tableau-based proof approach which is sound but not complete can be regarded as a mechanism for searching, not all, but only certain possible lines of argument, namely those defined by its set of saturation and fulfilling rules. If we believe that a proof can result along these lines, this is a good way of searching for it.

Algebraic deduction

Once the sequence of set-theoretic proofs with which we will be concerned in the main part of this book has moved along to the point at which the integers, rationals, and reals have been defined and their main properties established, the normal apparatus of algebraic proof becomes important. One relies on this to establish useful elementary identities on algebraic expressions, and also to show that algebraic combinations of elements belonging to particular sets (e.g. integers, reals, real functions and sequences, etc.) belong to these same sets. Inferences of this latter sort follow readily by syntactic transitivity arguments of the kind discussed already. Algebraic identities follow readily by expansion of multivariate polynomials to normal form, or by systematic or randomized testing of the values of polynomials and rational functions. Expansion to normal form can be used even for non-commutative multiplication operators.

To enable 'proof by algebra' for particular addition, subtraction, and multiplication operators, one issues a verifier command of a form like

```
ENABLE_ALGEBRA(s; plus_op; times_op)
```

or

```
ENABLE_ALGEBRA(s; plus_op(zero_constant);             minus_op; times_op)
```

or

```
ENABLE_ALGEBRA(s; plus_op(zero_constant);             minus_op; times_op(unit_constant))
```

etc. An example is

```
ENABLE_ALGEBRA(Z; *PLUS({}); *TIMES({{}}))
```

where Z denotes the set of integers. In these commands 's' should designate the set in which the algebraic operators work and on which they are closed. If a 'zero_constant' is supplied with the plus_op, it should designate the additive identity for the system. Similarly, if a 'unit_constant' is supplied with the times_op, it should designate the multiplicative identity for the system.

The ENABLE_ALGEBRA command scans the list of all currently available theorems for theorems which reference the operators and object s appearing as ENABLE_ALGEBRA parameters, collecting all those which state required algebraic rules like

```
(FORALL x in s, y in s | (x plus_op y) in s        & (x plus_op zero_constant) = x)
```

and similar commutative, associative and distributive rules. Automatic algebraic reasoning is turned on if proofs of all the basic axioms of polynomial arithmetic are found. To suspend the use of algebraic reasoning for a given collection of operators one writes a command like

```
DISABLE_ALGEBRA(plus_op)
```

where plus_op designates the addition operator that must be present in the group of operators whose automated treatment is being disabled.

Proof by closure

Proof by closure is an important special case of the more general 'proof by structure' technique explained in the next section. It works in those common cases in which certain small theorems of the general form

```
(P_1(x) & P_2(y) & ... & P_k(y)) -> Q(f(x,y))
```

will be applied repeatedly. The three statements

```
(x in Z & y in Z) *imp (x *PLUS y) in Z

(x in Si & y in Si & Is_nonneg(x) & Is_nonneg(y)) *imp           Is_nonneg(x *S_PLUS y)

(x in Si & y in Si & Is_nonzero(x) & Is_nonzero(y)) *imp           Is_nonzero(x *S_TIMES y)
```

where Z denotes the set of integers and Si the set of all signed integers are examples.

Common arguments involving obvious uses of such results can be handled by examining the syntax tree of functional expressions e mentioned in the course of a proof, and marking each with all of the monadic attributes the verifier has been instructed to track. All the nodes in the syntax tree of such e are then marked with the attributes which visibly apply, by a 'workpile' algorithm which works by transitive closure, examining each parent node one of whose children has just acquired a new attribute, until no additional attributes result. The propositions generated by this technique are then made available in the current proof context without explicit mention, for use in other proof steps.

To enable this kind of automatic treatment of particular predicates, one issues a verifier command of forms like

```
WATCH(x:x in Si; x:is_nonneg(x); x:is_nonzero(x))
```

The verifier then scans the list of all currently available theorems for theorems whose hypotheses are all conjunctions of statements involving the currently enabled predicates with a single variable as argument, and whose conclusions are clauses asserting that some combination of these variables also has a property defined by a predicate being watched. To drop one or more predicates from watched status, one issues a verifier command of a form like

```
DONT_WATCH(x:x in Si; x:is_nonneg(); x:is_nonzero()).
```

The conclusions produced by the WATCH mechanism automatically become available to the verifier's other proof mechanisms, but can also be captured explicitly by an inference introduced by the special keyword THUS, which also has access to the conclusions produced by the algebraic inference mechanisms described above. This makes accelerated inferences like the following possible. Suppose that a statement 'x in Si' has been established. Then the inference

```
   THUS ==> ((x *S_TIMES x) *S_PLUS ((x *S_TIMES x)             *S_TIMES (x *S_TIMES x))) in Si &    Is_nor
```

is immediate.

3.3. The resolution method for pure predicate calculus proving

Since all the set-theoretic concepts which we use can be expressed within the predicate calculus by adding predicate symbols and axioms, without any new rules of inference being needed, all the proofs in which we are interested can in principle be given without leaving this calculus. This observation has focused attention on techniques for automatic discovery of predicate proofs. A very extensive literature concerning this built up over the past four decades. This section will explain some of the principal techniques used for this, even though (for reasons that will be set forth at the end of the section, the authors believe that the size of the collections of formulae which such techniques need to explore prevents them from contributing more than marginally to a verifier of the kind in which we are interested).

The standard predicate-calulus proof-search technique begins by putting all of the formulae of a collection C of predicate statements to be tested for satisfiability first into prenex, and then into Skolem, normal form. All of the formulae in C then have the form

```
(FORALL x1,x2,...,xn | P),
```

where P contains no quantifiers. Propositional calculus rules can then be used to rewrite the 'matrix' P of this formula as a conjunction of disjunctions, each disjunction containing only atomic formulae, some of them possibly negated. We can then use the predicate rule

```
(FORALL x1,x2,...,xn | P & Q) *eq     ((FORALL x1,x2,...,xn | P) & (FORALL x1,x2,...,xn | Q))
```

to break up the conjunctions, thereby reducing C to an equisatisfiable set consisting only of formulae of the form

```
(FORALL x1,x2,...,xn | A1 or ... or Ak),
```

where each A_j is an atomic formula built from the predicate and function symbols (including constants) which appear in C, or possibly the negatives of such atomic formula. It is this standardized *disjunctive normal form* input on which predicate-proof searches then concentrate.

Herbrand's theorem tells us that such a collection C is unsatisfiable if and only if a propositional contradiction can be derived by substituting elements e of the Herbrand universe H for the variables of the resulting formulae in all possible ways. These elements are all the terms that can be formed using the constants and function symbols which appear in the formulae of C (one initial constant being added if no such constant is initially present in C). But if one tries to base a search technique directly on this observation, the problem of the exponential growth of the Herbrand universe with the

length of the terms allowed arises immediately. For example, even if C contains only one constant D and two monadic function symbols f and g, the collection of possible Herbrand terms includes all the combinations

```
f(f(g(f(g(g(...(D)))))))),
```

whose number clearly grows exponentially with their allowed length.

Some more efficient way of searching the Herbrand universe is therefore vital. The input formulae themselves must somehow be made to guide the search. A general technique for accomplishing this, the so-called *resolution method*, was introduced by J. Alan Robinson in 1965 (see J.A.Robinson, A machine-oriented logic based on the resolution principle, Journal of the ACM, Vol 12, No. 1, Jan 1965, pp. 23-49). We can best explain how this works by stepping back for a moment from the predicate to the simpler propositional calculus.

Resolution in the propositional calculus

Suppose then that we are given a collection C of formulae F of the propositional calculus, each such F being a disjunction of propositional symbols, some possibly negated. The resolution algorithm works on such sets by repeatedly finding pairs of formulae F_1,F_2 which have not yet been examined and which both contain some common atom A, but with opposite sign, and so have forms like

```
A or G1    and    (not A) or G2
```

where G_1 and G_2 are subdisjunctions, and deducing the formula

```
G1 or G2
```

from them (this is an instance of the tautology ((A *imp B) & (not A *imp D)) *imp (B or D)).

If an empty proposition can be deduced in this way, then the original collection C of propositions is clearly unsatisfiable, since the last resolution step must involve two directly opposed propositions A, '(not A)'. We will show that, conversely, if the original collection C of propositions is unsatisfiable, then an empty proposition can be deduced by resolution. Thus the ability to deduce an empty proposition via some sequence of resolution steps is necessary and sufficient for our original collection C of propositions to be unsatisfiable.

To establish this claim, we proceed by induction on the total length, in characters, of all the propositions in C. So suppose that C is unsatisfiable and that no empty proposition can be deduced from C by resolution, but that for every unsatisfiable collection C' of propositions of smaller total length there must exist a sequence of resolution steps which produces an empty proposition from C'.

Choose some propositional variable A that occurs in C. Clearly C has no model in which A has the truth value 'true', so if we drop all the statements of C in which A occurs non-negated (since these are already satisfied by the choice of 'true' for the truth-value of A), and use the tautology '((not true) or B) *eq B' to remove A from all the remaining statements of C, we get a collection C' of statements, clearly of smaller total length than C, which is unsatisfiable. Hence, by inductive assumption, there must exist some sequence of resolution steps which, applied to C', yield the empty proposition. But then the very same sequence s_1 of resolutions, applied to the statements of C' but before occurrences of '(not A)' are removed, will succeed in deducing '(not A)' by resolution.

In just the same way we can form a collection C'' of statements by dropping all the statements of C in which A occurs negated and drop A from the remaining statements. Since C'' must also be unsatisfiable, we can argue just as in the preceding paragraph to show that there must exist a deduction-by-resolution sequence s_2 from C which produces the single-atom conclusion A. Putting s_1 and s_2 one after another followed by a resolution step involving the formulae '(not

A)' and A, clearly gives a deductions-by-resolution from C which produces the empty proposition from C, verifying our claim.

Suppose that we write the result of a resolution step acting on two formulae F_1 and F_2 and involving the propositional symbol A as $F_1[A]F_2$. Then our overall sequence of resolution steps can be written as

```
...(F1[A]F2)[B](F3[D]F4)...,
```

the final result being an empty formula. Since each initial formula F of C occurs in this display only some finite number of times, we can give our sequence of resolutions the following form:

(i) Each of the formulae of C is copied some number of times.

(ii) The resulting formulae, and the results produced from them by resolution steps, are used only once as inputs to further resolution steps.

(iii) An empty proposition results.

Resolution and syntactic unification in the predicate calculus

In the predicate case, handled in the manner characterized by Herbrand's theorem, each of the resolution steps described above will involve an atomic formula A and its negative '(not A)'. Both of these will be obtained by substituting elements of the Herbrand universe H for variables appearing in atomic formulae A_1 and A_2 that are parts of formulae

```
F1 = A1 or B1 or ...
```

and

```
F2 = (not A2) or B2 or ...
```

of C.The sustitutions applied must clearly make A_1 and A_2 identical. Robinson's predicate resolution method results from a close inspection of conditions necessary for there to exist a substitution

```
x1-->t1,...,xn-->tn
```

of Herbrand t_j terms for the variables $x_1,...,x_n$ appearing in A_1 and A_2 which does this, i.e. makes the two substituted forms identical.

To see what is involved, note that since such substitutions can never change the predicate symbols P_1 and P_2 with which the atomic formulae A_1 and A_2 begin, identity can never be produced if these two predicate symbols differ. More generally, if we walk the syntax trees of A_1 and A_2 in parallel down from their roots, identity can never result by substitution if we ever encounter a pair of corresponding nodes at which different function symbols or constants f_1 and f_2 appear. In this case we say that our parallel tree-walks *reveal a conflict*. If this never happens, then, when we reach an end-branch in one or another of these trees, we must find either

(a) a variable x of the first tree matched to a compound term t of the second tree (momentarily, in this section, we call 'compound' any term which is not a variable, even if it is just a constant);

(b) a variable y of the second tree matched to a compound term t' of the first tree;

(c) a variable x of the first tree matched to a variable y of the second tree.

Only in these cases can there exist a substitution for the variables of A_1 and A_2 which makes the two substituted forms identical. It also follows that (a), (b), and (c) together give us an explicit representation of the most general substitution S (called the *Most General Unifier* of A_1 and A_2 and written $Mgu(A_1,A_2)$) for the variables of A_1 and A_2 which makes the two substituted forms identical. This is obtained simply by collecting all the substitutions

```
(*)  x-->t, ... ,y-->t', ... ,x-->y, ...
```

which appear in (a), (b), and (c) respectively, and whose role is to convert each of the pairs [x,t] into an identity x = t after the indicated substitutions have been performed for all variables.

As shown by the pair of formulae

```
P(x,x)      and      P(f(y),g(y)),
```

it is entirely possible that the collection (*) should contain multiple substitutions $x\text{-->}t_1$, $x\text{-->}t_2$ with the same left-hand sides. In this case, we must find further substitutions which make t_1 and t_2 identical. This is done by walking the syntax trees of t_1 and t_2 in parallel, and applying the collection process just described, following which we can drop $x\text{-->}t_2$ from our collection since the additional substitutions collected make it equivalent to $x\text{-->}t_1$. Since his process replaces substitutions $x\text{-->}t_2$ with substitutions having smaller right-hand sides it can be continued to completion, eventually either revealing a conflict or giving us a collection (*) of substitutions in which each left-hand variable x appears in just one substitution.

However, as the following example shows, one more condition must be satisfied for the presumptive substitution (*) to be legal, i.e. to define a pattern of substitutions which allows all the substitutions (*) into equalities. Consider the two formulae

```
P(x,f(x))      and      P(f(y),y).
```

Applying the procedure just described to these two formulae yields the substitutions

```
x-->f(y), y-->f(x).
```

The problem here is that there exists a cycle of variables x,y,x such that each appears in the term to be substituted for the previous variable, i.e. y appears in the term to be substituted for x and x in the term to be substituted for y. Any such substitution of compound terms x' and y' for x and y respectively would give rise to identities

```
x' = f(y') and y' = f(x'),
```

and hence to x' = f(f(x')), which is impossible.

The same argument applies in any case in which the collected substitutions (*) allow any cycle of variables such that each appears in the term to be substituted for the previous variable. On the other hand, if there is no such cycle of variables, then we can arrange the collection of all variables appearing in (*) in an order such that each variable on the left comes later in order than all the variables appearing on the right, and then progressive application of all these substitutions to the variables appearing on the right clearly reduces all of them to identities. In this case we say that a most general unifier $Mgu(A_1,A_2)$ exists for the two atomic formulae A_1,A_2; otherwise we say that *unification fails*, either *by conflict* or *by a cycle.*

We can just as easily find the most general substitution which reduces multiple pairs A_1,A_2, B_1,B_2 to equality simultaneously. An easy way to do this is to introduce an otherwise unused artificial symbol Y, and then apply the unification technique just described to the pair of formulae

```
Y(A1,B1,...)    and   Y(A2,B2,...) .
```

Clearly a substitution makes these two formulae identical if and only if it reduces all the pairs A_1,A_2, B_1,B_2 to equality simultaneously.

For use in the next section we will need a somewhat more precise statement concerning the relationship between the most general unifier of two sets of atoms or compound terms, and the other substitutions which unify these same atoms/terms. In deriving this statement it will be convenient to write

```
(+)  Mgu([t1,...,tn],[t1',...,tn'])
```

for the most general simultaneous unifier of all the atoms/terms t_j with the corresponding t_j', and

```
(++) All_u([t1,...,tn],[t1',...,tn'])
```

for the collection of all substitutions which unify all the atoms/terms t_j simultaneously with the corresponding t_j'. Using these notations, take any t_j, t_j' in the sequences shown. If these are atomic formulae or terms and have distinct initial symbols, unification is impossible. Otherwise if they are atoms/terms and have identical initial symbols, they will unify if and only if their arguments unify; hence we can replace t_j and t_j' by their argument sequences in (+) without changing its value. The same argument gives the same conclusion for (++).

If no further replacements of the kind just described are possible, then for each pair t_j, t_j' either t_j and t_j' must be identical constants, or at least one of t_j, t_j' must be a variable. We collect all pairs in which both are variables, which the substitutions in which we are interested must convert to identical terms, choose a representative for each of the groups of equivalent variables thereby defined, and, in all other terms/atoms, replace all occurrences of variables having such representative by their representative. Again it is obvious that this transformation of the t_j and t_j' changes neither (+) nor (++). Once this standardization of variables has been accomplished, we collect all cases in which a given variable v appears as a t_j or t_j' and is mapped to a non-trivial t_j' or t_j. All but one of these pairs are removed from the argument sequences of (+) and (++), and replaced with other pairs implying that each of the remaining terms must be equal to the term retained. Again this is a transformation that changes neither (+) nor (++).

The step just described may allow the whole sequence of steps that we have described to restart, so we keep iterating till none of the steps we have described are possible. At this point each t_j in (+) will be matched either to an identical constant t_j', or one of t_j and t_j' will be a variable that appears only once, while the other is a variable or term. Neither (+) nor (++) will have changed.

Whenever we have a corresponding pair t_j, t_j' in which one member is a variable, we say that the term *expands* the variable. We shall call variables x which appear somewhere in t_1,...,t_n,t_1',...,t_n', but do not have representatives and are not matched to non-trivial terms in pairs t_j, t_j' *base variables.* We complete our calculation of Mgu by repeatedly replacing all variables that expand into nontrivial terms t by these terms t. Again this transformation changes neither (+) nor (++). Since we have seen that unification is only possible if there is no cycle of expansions, this process must converge, at which point every remaining variable will either be a base variable, have a base variable as its representative, or be expanded into a term in which only base variables appear. Now let S be a member of the set (++) of substitutions, i.e. a substitution which makes each t_j equivalent to its corresponding t_j'. If t_j and t_j' are both variables then it is clear that S must substitute the same term for both of them. If one of them, say t_j, is a variable and the other t_j' is a term, then it is clear that the term which S substitutes for t_j must be the same as that which results by first substituting t_j' for t_j, and then substituting Sx for each base variable x remaining in t_j'. Thus, if we let S_0 designate the restriction of S to the base variables, it follows that the substitution S (regarded as a mapping of variables into terms) factors as the product M o S_0, where M is the most general unifier (+). We state this observation as a lemma.

Lemma: If two sequences $t_1,...,t_n$ and $t_1',...,t_n'$ consisting of atomic formulae an/or terms can be unifies by a substitution S which makes each t_j identical to its corresponding t_j', then each substitution S having this effect can be written as a product S = M o S_0, where M is the most general unifier

```
Mgu([t1,...,tn],[t1',...,tn']),
```

and the substitution S_0 replaces some of the base variables of M by other variables or nontrivial terms. Conversely, by applying any substitution S of the form M o S_0 to all of the t_j and t_j' we make each t_j identical to its corresponding t_j'.

The preceding discussion of resolution and unification gives us the following general way of handling the problem of finding a Herbrand contradiction which will show that a collection C of predicate formulae given in our normal form

```
(*)   (FORALL x1,x2,...,xn | A1 or ... or Ak)
```

is unsatisfiable.

> (Res-i) Guess, or search for, the pattern in which resolution steps can (or will) occur in a sequence of such steps (for substituted instances of our collection C of formulae) leading to a propositional contradiction.
>
> (Res-ii) The guess (or search) (i) implies that designated atomic formulae A occurring in C, perhaps in multiple copies of formulae like (*) (but with the quantifiers in (*) removed), Must unify in the pattern determined by the sequence of resolution steps. Check that this unification is actually possible. If so, the substitutions forced by the required unifications identify a collection of elements in the Herbrand universe which allow the pattern of resolutions found in step (i) to be executed, and thereby show that the set C of predicate statements is unsatisfiable.

We can use the example

```
(EXISTS x | (FORALL y | P(x,y))) *imp              (FORALL y | (EXISTS x | P(x,y)))
```

as a particularly simple example of the proof method just described. The negative of this implication, rewritten as a pair of clauses in Skolem normal form, is

```
(+)  (FORALL y | P(D1,y))),              (FORALL x | (not P(x,D2)))).
```

The substitutions x->D_1 and y->D_2 unify P(D_1,y) with (not P(x,D_2)), giving P(D_1,D_2) and (not P(D_1,D_2)), a clear contradiction which proves the unsatisfiability of (+), and so the universal validity of our original formula. Note that if we started with the reverse implication

```
(FORALL y | (EXISTS x | P(x,y))) *imp              (EXISTS x | (FORALL y | P(x,y)))
```

whose Skolemized inverse is

```
(FORALL y | P(f1(y),y))),              (FORALL x | (not P(x,f2(x)))),
```

we wold neeed to unify P(f_1(y),y)) and P(x,f_2(x))), which leads to the (cyclic) impossibility

```
x->f1(y), y->f2(x).
```

This shows that the reverse implication is not universally valid.

A great variety of methods which aim to reduce the cost of the combinatorial search implicit in (Res-i) and (Res-ii) above have been published. Some are deterministic pruning schemes, which aim to eliminate whole subtrees of the search tree by showing that none of their descendant searches can succeed. Others are standardization techniques, which eliminate redundant work by performing the necessary searches in an order and manner allowing many redundancies to be eliminated, perhaps by detecting and bypassing them. Still others are heuristics guided by guesses concerning favorable unifications and sets of statements. These may involve some implicit or explicit notion of the distance separating an intermediate set of resolution steps from the full set needed to demonstrate unsatisfiability.

A short summary of some of these methods will be given below. The commonly encountered Horn case, in which each quantifier-stripped formula of the input contains at most one non-negated predicate atom, serves to illustrate some of the issues involved. Since every substituted instance of a Horn formula is also Horn, we can use the observation made in our earlier discussion of Horn sets in the propositional case to establish that only resolutions involving at least one positive unit formula need be considered, and that if the null clause can be deduced it can be deduced using just one of the negative unit formulae, and that only once.

We will use the set of (quantifier-stripped) formulae seen below as an example. Their unsatisfiability expresses the following theorem of elementary group theory: in a group with left inverse and a left identity, each element also has a right inverse. In these formulae, the normal group-theoretic operation x * y is recast in pure predicate form by introducing a predicate P(x,y,z) representing the relationship z = x * y. Inspection of the formulae displayed below shows that only this predicate is needed. The first two statements respectively express the hypotheses 'there is a left inverse' and 'there is a left identity'. The next two statements allow reassociation of products to the left and to the right. The final statement is the negative of the desired conclusion: 'there is an element "a" with no right inverse'.

```
P(I(x),x,e)  P(e,x,x)  (not P(x,y,u)) or (not P(y,z,v)) or (not P(u,z,w)) or P(x,v,w)  (not P(x,y,u)) or
```

Since the set C of formulae shown is evidently Horn, we can (in accordance with our earlier discussion of Horn sets) regard the two first formulae as 'inputs', the last formula as a 'goal', and the two remaining formulae as 'multiplication rules' which allow triples of inputs to be combined (if the simultaneous unifications required for this are possible) to produce new unit-formula inputs. We must then aim to find a sequence of such multiplications which reaches the negative of our 'goal' formula. This is a path-finding problem resembling others studied in the artificial intelligence literature. It is easily organized for efficiency in the following way. At any given moment a collection Uc of positive unit formulae will be available. We form all triples of these formulae which can be combined using the two available 'multiplication rules' and generate new positive unit formulae. This step is repeated until either our goal formula is reached or the resulting computation becomes infeasible.

This way of looking at things reveals a (deep) pitfall that can affect resolution searches, even in particularly favorable Horn cases like the one under consideration. Since each of our two 'multiplication rules' allows the available inputs to be combined in up to three possible ways, each cycle of 'multiplication' can in the worst conceivable case increase the number n of available atomic formulae to as much as $2n^3 + n$. Even starting from n = 2 this iteration increases very rapidly: 2; 18; 11682; 3,188,464,624,818;... Unless this exponential increase in the size of our search space is strongly limited by the failure of most of the unifications required by the 'multiplication' operations considered, we could hardly expect to search more than 4 levels deep without using some other idea to prune our search very drastically.

Deduction succeeds in the example shown above, in part because a quite 'shallow' proof is possible. This is a proof involving only two successive multiplications. Even without additional search optimizations, the proof is found after 75 unification attempts, of which 7 successfully generate new atomic formulae. 2, rather than 16, formulae are added to the list of available atoms at the end of the first cycle of multiplication, so the branching factor is not nearly as bad as is indicated by the worst-case estimate given above, making it reasonable to estimate that proofs a much as 6 levels deep may be within reach of the resolution method in the pure Horn-clause case. The proof found is

```
P(I(I(X)),E,X) from: [P(I(X),X,E), P(I(X),X,E), P(E,X,X)]     using: [P(X,Y,U), P(Y,Z,V), P(U,Z,W), P(X,V
```

The following formulae are generated but not used in the proof found:

```
P(E,E,E), P(I(E),X,X), P(I(E),E,E), P(I(I(I(X))),X,E),     P(I(I(I(E))),E,E), P(I(I(E)),X,X), P(I(I(E)),E
```

Note that a few of these formulae are special cases of others or of input formulae, and so could be omitted. For example, P(E,E,E) is a special case of P(E,X,X), and P(I(E),E,E) is a special case of P(I(E),X,X).

Examination of the above list of useless atomic formulae reveals that some of them are *subsumed* by, i.e. are special cases, of others, and hence visibly unnecessary. For example, P(E,E,E) is a special case of P(E,X,X), and P(I(I(E)),E,E) is a special case of P(I(I(E)),X,X). The unification procedure can be used to test for and eliminate these redundancies. If this is done, the number of unifications attempted in the preceding example falls to 87, and only the following 4 unneeded atomic formulae are generated:

```
P(I(E),X,X), P(I(I(I(X))),X,E), P(I(I(E)),X,X),    P(I(I(I(I(X)))),E,X), P(I(I(I(E))),X,X)
```

The following is a second Horn example (taken, like the example above, from Chin-Liang Chang and Richard Char-Tung Lee, *Symbolic Logic and Mechanical Theorem Proving,* Academic Press 1973, p. 160).

```
D(x,x)  L(m,a)  (not P(x)) or D(g(x),x)  (not P(x)) or L(m,g(x))  (not P(x)) or L(g(x),x)  (not D(x,a)) o
```

Here we have two inputs, seven multiplication rules (of these, four involve just one input, two involve two inputs each, and one involves three inputs), and one target, which in this case is a disjunction of three atoms rather than a single atom.

Deduction succeeds in this case after 5 levels of multiplication, involving 123 unification attempts of which 17 generate new atomic formulae, 8 being used in the proof found, which is

```
P(A) from: [D(X,X)] using: [D(X,A), P(X)]  D(G(A),A) from: [P(A)] using: [P(X), D(G(X),X)]  L(M,G(A)) fro
```

No subsumption cases occur during the processing of this example. Here the branching factor is seen to be quite small. The following atomic formulae are generated but not used in the final proof.

```
P(G(A)), D(F(G(A)),G(A)), D(G(G(A)),G(A)), L(M,G(G(A))),   L(G(G(A)),G(A)), D(G(G(A)),A), D(G(F(G(A))),F(
```

In this case the search efficiency can be improved by using a simple heuristic, which attempts to find 'easy' proofs (those involving relatively short formulae) before trying harder ones. As new atomic formulae are generated, we prefer the shorter of the new formulae over the longer by sorting the newly generated formulae into order of increasing string length an adding just one new formula, the shortest, to the collection of inputs used during each cycle of multiplication. With this improvement we find the same proof after 48 unification attempts of which 12 generate new atomic formulae. Another small group-theoretic example from Chang and Lee shows some of the difficulties that slow or block resolution proofs in more general cases. This states the axioms of group theory in the same ternary form as above, but also introduces a predicate S(x) which asserts that x is an element of a particular subgroup of the group implicit in the axioms. An axiom states that this subgroup is closed under the operation x * I(y), and we are simply required to prove that the inverse of an element b of the subgroup belongs to the subgroup. The input axioms are

```
P(I(x),x,e)  P(x,I(x),e)  P(e,x,x)  P(x,e,x)  S(b)  not S(I(b))  (not P(x,y,u)) or (not P(y,z,v)) or (not
```

The proof found involves just two steps:

```
S(E) from: [S(B), S(B), P(X,I(X),E)]     using: [S(X), S(Y), P(X,I(Y),Z), S(Z)]  S(I(B)) from: [S(E), S(E
```

However the search required makes many unification attempts and generates many useless formulae having forms like

```
P(E,X,I(I(I(I(I(X)))))))  P(I(X),X,I(I(I(E))))  P(I(I(I(I(E)))),E,I(E)) etc.
```

In this case the 'easy proofs' heuristic considered above greatly improves search efficiency, finding a proof after 139 unification attempts and the generation of 12 atomic formulae.

Next we present a technique that realizes the ideas of (Res-i) and (Res-ii) very directly in non-Horn cases. Before giving the details of this scheme, we need to take notice of a technical point overlooked in the preceding discussion. For resolution to work as claimed, even at the propositional level, duplicate occurrences of propositional symbols must be eliminated. For example, the two statements

```
(++)  A or A, (not A) or (not A)
```

are clearly contradictory and a null proposition follows immediately by resolution if these are simplified to A, (not A). But if we resolve without eliminating duplicates resolution leads only to 'A or (not A)' and thence back to the original statements (++), and so we can never reach an empty proposition. Both in the purely propositional and the predicate cases, we must remember to eliminate duplicate atomic formulae whenever resolution produces them.

Here is one way in which the steps (Res-i) and (Res-ii) above can be organized.

> (a) We begin by guessing the number of times each of our input formulae (*) need to be used to generate distinct substituted instances in the refutation-by-resolution for which we are searching. This creates an initial collection C of formulae F which we strip of quantifiers. Distinct variables are used in each of these formulae, and the set Initial_atoms of all atomic formulae A which they contain is formed. Each such A is associated with the F in C in which it appears, and with the sign (negated or non-negated) with which it appears. The F in C are given some order, which is then extended to a compatible ordering of all the atomic formulae A in these F.
>
> (b) A preliminary survey is made of all the pairs A_1,A_2 in Initial_atoms, to determine the cases in which A_1 and A_2 can be unified. These are collected into two maps: can_rev(A_1) holds all the A_2 of sign opposite to A_1 with which A_1 can unify, and can_same(A_1) holds all the A_2 of the same sign as A_1 with which A_1 can unify.
>
> (c) One these maps have been collected we search for a combinatorial pattern representing a successful refutation-by-resolution. These must have the following properties:
>
> > (c.i) Each atomic formula A_1 must be mapped into an element either of can_rev(A_1) or can_same(A_1) by a single-valued mapping match(A_1).
> >
> > (c.ii) If A_2 = match(A_1) belongs to can_rev(A_1), then we must have A_1 = match(A_2).
> >
> > (c.iii) It must be possible to unify all the atomic formulae A_1 with the corresponding match(A_1) simultaneously.
> >
> > (c.iv) No two formulae F_1, F_2 in C containing atomic formulae A_1, match(A_1) of opposite sign can lbe connected by a prior chain of links between matching atomic formulae of opposite signs.
> >
> > (c.v) The collection of propositions generated from the F in C by identifying A_1 and A_2 whenever A_2 = match(A_1) is unsatisfiable.

Review of the conditions (c.i-c.v) shows them to be equivalent to the condition that corresponding substitutions into the formulae of C define a group of resolution steps leading to an empty statement. The matches for which match(A_1) and A_1 have opposite signs correspond to resolution steps involving the atomic formulae A_1 and match(A_1); the matches for which match(A_1) and A_1 are of identical sign correspond to eliminations of duplicate atomic formulae. Condition (c.ii) states that the pairs of atomic formulae A_1, match(A_1) entering into resolution steps are symmetrically related. Condition (c.i) states that all the necessary unifications must be individually possible; (c.iii) states that all must be simultaneously possible. Condition (c.iv) excludes tautologous intermediate formulae containing two identical atoms of opposite sign. Condition (c.v) ensures that the pattern of resolutions chosen can lead to a null formula.

The unifiability check required in step (c.ii) above can be organized in the following way.

> (c.iii.i) All the formulae F in C are parsed, and each node in the resulting syntax trees is marked with its associated predicate symbol, function symbol, or variable, and with all its descendant variables.
>
> (c.iii.ii) When two groups of atomic formulae $A_1,A_2,...,A_n$ and $B_1,B_2,...,B_n$ are to be checked for simultaneous unifiability, we collect all the top-level terms $t_1,...,t_m$ and $t_1',...,t_m'$ from them in order, form two corresponding atomic formulae $Z(t_1,...,t_m)$ and $Z(t_1',...,t_m')$ using an auxiliary predicate symbol Z, and test these two formulae for unifiability. All of the necessary operations can be managed efficiently using lists and sets of pointers to syntax tree nodes. Topological sorting can be used to check that a purported collection of substitutions leads to no cycles among variables.

To eliminate the repeated examination of failed unification patterns

Some of the optimization heuristics that have be used in the many other resolution approaches described in the literature can be worked into the scheme presented above.

3.4. Universally quantified predicate formulae involving function symbols of one argument only

We shall now use some of the ideas developed in the preceding section to derive an algorithm for determining the satisfiability of sets of pure predicate formulae of the restricted form

```
(*)  (FORALL x | P),
```

whose 'matrix' P is a Boolean combination of atomic formuale $A(t_1,t_2,...,t_k)$, where the argument terms t_j must be built from constants using monadic function symbols only. A(x,f(x),...,f(g(f(h(f(f(x))))))) is an example of such an atomic formula. Note that Skolemization of formulae

```
(**)  (FORALL y | (EXISTS x1,x2,...,xn | Q),
```

where the matrix Q is subject to the same restriction, always leads to formulae of this kind, so that the algorithm we present will also decide the validity of formulae (**),

We begin our analysis by transforming P propositionally into a conjunction of disjunctions of atomic formulae, each of which is either negated or non-negated. Since the predicate identity

```
  (FORALL y | R & R') *eq ((FORALL y | R) & (FORALL y | R'))
```

can be used to decompose the conjunctions, we can suppose that each of the matrices P in (*) is a disjunction of negated and non-negated atomic formulae.

Herbrand's theorem assures us that (*) is satisfiable if and only if no propositional contradiction arises among any of the instances of (*) formed by substituting elements of the Herbrand universe H for the x in (*). The appearance of such a contradiction will reflect the pattern in which substituted instances of atomic formulae A_1 and A_2 appearing in the matrices P of such formulae become equal. For two such A_1 and A_2 to be made equal by any substitution they must unify. The discussion of unification developed in the preceding section tells us that two such atoms A_1 and A_2 (initially transformed to have different variables x and y) will only unify in one of the following cases:

(i) they are made equal by replacing one of x an y by the other.

(ii) they are made equal by replacing the variable x by some constant term t and the variable y by some other constant term t'.

(iii) they are made equal by replacing the variable x by some term formed from the variable y using the available monadic function symbols, for example replacing x by f(g(f(h(f(f(y)))))).

(iv) they are made equal by replacing the variable y by some term formed from the variable x using the available function symbols.

In case (i) the two atomic formulae are equal if written using the same variable x. In case (ii) we have A(t) = B(t') for the two constant terms t and t'. In case (iii) we have the identity A(t(y)) = B(y) for some term t(y) formed using the available function symbols, and similarly in case (iv) we have A(x) = B(t(x)). In the first of these two cases we say that B is *expressible in terms of A*. in the second that A is expressible in terms of B. Note that expressibility in this sense is transitive. Moreover, if any B is expressible in terms of two distinct A_1 and A_2, then there is clearly a substitution which makes A_1 and A_2 identical, so one of A_1 and A_2 must be expressible in terms of the other. Thus in each group of atomic formulae related by a chain of expressibility relationships there must be one, which we shall call A, in terms of which all the others are expressible, and which is such that A is not expressible in terms of any other atomic formula B. We call such A *basic* atomic formulae. Note that given two different basic atomic formulae A and B, there can be no substitutions for the variables x and y they contain which makes A and B identical.

We now take the matrices P of all the formulae of our collection, and introduce a new monadic predicate symbol Q(x) for each basic atomic formula A which appears in them. All the other atomic formulae B can then be expressed uniquely in terms of these Q, as Q(t), where t is a term formed by applying the available function symbols to the variable x appearing in B, or possibly t is a constant term formed by applying these function symbols to a constant. Let P' by the matrix formed by replacing each of the atomic formulae in P by its corresponding Q, or, if A is not basic, by the appropriate Q(t). Since each basic atom A(x) in P has a unique corresponding Q, it is clear that if the set of formulae (*) has a model, so does the set of formulae

```
(+)    (FORALL x | P')
```

derived from it in the manner just explained. Suppose conversely that (+) has a model M, whose universe we may, by Herbrand's theorem, take to be the Herbrand universe H. For each predicate symbol Q appearing in one of the formulae (+), let Q^M be the Boolean function corresponding to it in the model M. If Q(x) has been used to represent a basic atomic formula $A(t_1(x),..,t_k(x))$, define $A^M(x_1,..,x_k)$ to be Q(x) for all tuples $x_1,..,x_k$ of arguments in the Herbrand universe H which have the form $[t_1(x),..,t_k(x)]$ for some x in H, but to be false for all other argument tuples. Since the tuples $[t_1(x),..,t_k(x)]$ which appear as arguments of different basic atomic formulae in (*) must always be different if the predicate symbols A appearing in these formulae are the same (since otherwise some substitution would unify the distinct basic atomic formulae, which is impossible) it follows that this definition of Boolean values is unique. Since no other argument tuples appear in (*), it follows that this assignment of Boolean mappings to the predicate symbols appearing in (*) gives a model of (*). Thus the sets (*) and (+) of formulae are equisatisfiable. But all of the predicate symbols which appear In (+) are monadic, so the satisfiability of (+) can be decided by a procedure described earlier. It follows at once that the reduction which we have just described, used together with this procedure, decides the satisfiability of sets (*) of

formulae.

3.5. Accelerated instantiation of quantifiers and setformers

Steps which simply generate instances of quantified formulae and implicitly quantified formulae involving setformers are common in the proofs in which we will be interested. For example, we may need to deduce the contradiction

```
not (e(c) = e(c) & c in s)
```

from the set-theoretic statement

```
e(c) notin {e(x): x in s},
```

or to deduce

```
((a in s & c = e(a) & p(a)) &         (not (a in s & c = ep(a) & pp(a)))) or   ((not (b in s & c = e(b)
```

where a and b are two newly generated symbols, from a previously proved set-theoretic statement

```
(c in {e(x): x in s | p(x)} &         (not (c in {ep(x): x in s | pp(x)})) or   s((not(c in {e(x): x in
```

Our verifier handles steps of this common kind in the following friendly way. All variable names which are not explicitly bound by quantifiers (or bound in setformers) are temporarily regarded as 'constants', i.e. substitutions for them are temporarily forbidden. We first search for quantifiers, which may be nested several levels deep within propositional structures like

```
((FORALL x | P(x)) and .. (EXISTS y | Q(y))...)     or.. implies ... not (FORALL z | R(z)) ...
```

but not nested within other quantifiers. We generate unique bound variables for each of these quantifiers, also making sure that they are distinct from any free variables that appear.

Each quantifier in such a propositional structure has an implicit 'sign', determined by the following simple rules:

(i) Un-nested universals are positive, while un-nested existentials are negative.

(ii) The nesting of a quantifier within a construction 'a and b' or "a or b' does not affect its sign.

(iii) Each level of nesting of a quantifier within a negation 'not a' reverses its sign, e.g. 'not .. not ... not ...(EXISTS y | Q(y))' is positive.

(iv) 'a implies b' is simply '(not b) or a'.

In nested propositional constructs like those shown above, we could if we wanted move any quantifier out to the front by using the standard rules

(FORALL x | (P(x) and A)) *eq (FORALL x | P(x)) and A)

(FORALL x | (P(x) or A)) *eq (FORALL x | P(x)) or A)

(EXISTS x | (P(x) and A)) *eq (EXISTS x | P(x)) and A)

(EXISTS x | (P(x) or A)) *eq (EXISTS x | P(x)) or A),

where we assume that the variable x is not free in A. Once moved out these quantifiers could be instantiated, subject to the normal rules:

(a) Only a previously unused constant symbol can be substituted for a negatively (existentially) quantified bound variable.

(b) Any constant expression at all can be substituted for a positively (universally) quantified bound variable.

It is clear that the instantiations allowed by these rules can be performed without the preliminary step of moving the quantifier to the front. Any number of negative quantifiers can be replaced, one after another, by hitherto unused constant symbols. Then any number of positive quantifiers can be replaced by any desired expressions. It is clear that these rules apply also to nested quantifiers, which can be subjected to sequences of instantiations moving inward. Note however that an existential nested within a universal can never be instantiated unless the universal is first instantiated in accordance with the rule we have stated.

Given a formula F and a second F', it is easy to determine, by comparing their syntax trees, whether F' arises from F by such a set of instantiations. If it does, then F' is valid deduction from F. Our verifier allows steps of this kind to be indicated simply by writing the keyword INSTANCE.

Here are a few examples showing that this works well for various familiar pure-predicate cases:

To prove

```
(FORALL x | P(x)) *imp (EXISTS x | P(x))
```

we form its negative

```
(FORALL x | P(x)) and not (EXISTS x | P(x))
```

and then the instance

```
P(c) and not P(c)
```

which is clearly impossible, proving the validity of our first statement. Similarly we can prove the validity of

```
(EXISTS y | (FORALL x | P(x,y))) *imp          (FORALL x | (EXISTS y | P(x,y))
```

by forming its negative, which is

```
 (EXISTS y | (FORALL x | P(x,y)) &          (not (FORALL x | (EXISTS y | P(x,y))),
```

and then instantiating this to the impossible P(d,c) and not P(d,c).

Statements involving membership in setformers can be treated in much the same way, since

```
a in {e(x): x in s | P(x)}
```

is a synonym for

```
(EXISTS x | a = e(x) and x in s and P(x)).
```

Thus every membership (resp. nonmembership) statement counts initially as negative (resp. positive) and instantiates to 'a = e(c) and c in s and P(c)' (resp. 'a /= e(c) or c notin s or not P(c)'), where c must be a new constant if the context of the setformer makes it negative, but can be any expression if the context of the setformer makes it positive. This observation makes it possible to recognize the deduction shown at the very start of this section as an instantiation, which can be written as

```
INSTANCE(Stat1) ==> not (e(c) = e(c) & c in s)
```

in our verifier. Similarly, the deduction in the second example at the start of this section is a combination of positive and negative instantiations which can be written as

```
 INSTANCE(Stat1) ==>   ((a in s & c = e(a) & p(a)) & (not (a in s &         c = ep(a) & pp(a)))) or
```

Note finally that the Boolean equivalence operator '*eq' must be decomposed into its two parts 'a & b or (not b) & (not a)', since these give different signs to quantifiers or setformers nested within them.

3.6. The Knuth-Bendix equational method

(A) Overview of the method

The equational method introduced by Knuth and Bendix in their well-known paper *Computational Problems in abstract algebras* (J. Leed, Editor, Pergamon Press, pp. 263-297, 1970) offers a general and systematic treatment of the algebraic process of 'simplification'.

It assumes that all the hypotheses to be dealt with are universally quantified equations of the form

```
(FORALL x1,x2,...,xn | t = t'),
```

and determines whether these entail another such identity $t_0 = t_0'$.

Given a set C of identities t = t' whose implications are to be analyzed, one begins by arranging them in a 'downhill' direction t ~> t', with the 'simpler' side of each identity on the right. The identities will always be used in this direction. One then determines whether these simplifications always lead to a unique ultimate reduction of every term t.

For this approach to be possible, some systematic notion of 'expression complexity' is required. Knuth and Bendix define such a complexity measure by adding up the total number of symbols in each expression, possibly with auxiliary assigned 'weights'. Expressions having the same total weight are ordered in a suitable lexicographic way. For this easy notion of complexity to be stable in the presence of substitution for variables, in a manner which guarantees that if exp is 'simpler' than exp' then every substituted form of exp is 'simpler' than the corresponding substituted form of exp', we also require that the number of occurrences of every variable in t be at least as large as the corresponding number in t'. (Thus, systems including identities like f(x,y,y) = f(x,x,y) are out of reach of the Knuth-Bendix method).

When a clear direction of simplification can be defined in the way explained, we can reduce any expression exp to (a possibly non-unique) 'canonical' or 'irreducible' form by repeatedly (and nondeterministically) finding some subexpression exp' of exp which is identical with a substituted version of the left-hand side of some simplification t ~> t', and then replacing exp' within exp by the corresponding substituted version of the right-hand side t' of this same simplification. If the irreducible form of each expression turns out to be unique, we will have an easy test for determining whether the equality of two expressions exp, exp' is entailed by a collection of identities: reduce both exp and exp' to their irreducible forms, and see if these are equal. Thus the essential point is to be able to determine when the irreducible form of every

expression exp is unique.

Given an expression e which contains another expression e' as a subexpression, we can write e as a...e...b, where a... (resp. ...b) is the part of e that precedes (resp. follows) its subexpression e'. If σ denotes a substitution x_j-->e_j which replaces each of a collection of variables by some expression and e denotes an expression in which these variables appear, we will write (eσ) for the result of replacing all occurrences of each of the variables x_j by the corresponding e_j. We temporarily reserve the letter σ (possibly subscripted) for substitutions of this kind.

We will see that the irreducible form of an expression e, i.e. a simplification of e which cannot be simplified further, can only be non-unique when e contains some subexpression se, which in turn has a sub-sub-expression sse, having the following property:

> (i) se must have the form ($t\sigma_1$), where t is the left-hand side of some simplification t~>t_1 and σ_1 is a substitution (as above).
>
> (ii) sse must have the form ($T\sigma_2$), where T is the left-hand side of some simplification T~>T_1 and σ_2 is also a substitution (as above).
>
> (iii) In this situation we can write e in either of two ways, namely either as a...($t\sigma_1$)...b or as a...a'...($T\sigma_2$)...b'...b , and accordingly can simplify it in either of two ways, namely either to a...($t'\sigma_1$)...b or to a...a'...($T'\sigma_2$)...b'...b . These can only fail to have the same irreducible form if ($t'\sigma_1$) and a'...($T'\sigma_2$)...b' can have different irreducible forms.

Now we can note that ($t\sigma_1$) (resp. ($T\sigma_2$)) is a substituted form of the left-hand side of the entire left-hand side of the entire left-hand side of the simplification t~>t' (resp. $T\sigma_2$). Hence there can exist an expression e having two different ultimate simplifications only if there are a pair of simplifications t~>t' and T~>T' such that

> (a) Some subexpression s of t can be 'unified' with T by a pair σ_1, σ_2 of substitutions σ which make Sσ and Tσ syntactically identical.
>
> (b) The two simplifications ($t'\sigma_1$) and $a'\sigma_2$...($T'\sigma_2$)...$b'\sigma_2$ of t thereby generated (where t == a...s...b) have distinct irreducible forms r_1 and r_2.

In this case the identity r_1=r_2 is plainly a consequence of our initial set of identities since it is obtained by simplifying ($t'\sigma_1$) in two different ways. It may be possible to arrange this 'new' identity as a simplification r_1~>r_2 and add it to our initial set of simplifications, thereby getting an expanded set E of simplifications, in which plainly r_1 and r_2 have the same simplified form r_2. If we are lucky, this expanded set of simplifications will give every expression a unique irreducible form, in which case we say that E has '*attained completion*'. If this is not the case, we can repeat the procedure just described to find a further expansion of E, and hope that E attains completion after some finite number of expansion steps.

As this process goes along we must always arrange the equalities t = t' with which we are working as reductions t ~> t', which means that some appropriate way of ordering the terms e appearing in these clauses must always be kept available. In many cases the new equations a = b generated will fit immediately into the ordering of terms used previously. In such situations the term-ordering used need not be changed. If this is not the case, a new ordering can be adopted at any time (since the role of the ordering is merely subsidiary, i.e. serves only to define the direction of reduction). But when a new ordering is adopted one may well want to examine the existing equations to see if any can be dropped.

(B) Details

(i) Ordering of ground terms

We suppose that a collection C of (implicitly universal) identities t = t' is given, and form the Herbrand universe of all terms that can be built from the constants of C using the function symbols which appear in C. (As usual, we add one 'priming' constant if none is available in C). Assume that a non-negative integer weight w(f) is associated with each constant c and function symbol f, and that all the constants and function symbols have been arranged in some order, so that we can write f > g if f comes later than g in this order. Function symbols of more than one argument can have 0 weight, but we assume that at most one monadic function symbol f can have 0 weight and that all constants have positive weight.

Given this assumption we can define the weight w(t) of a *ground* term (i.e. a term containing no variables, only constants and function symbols) to be the sum of the weights of all its constants and function symbols.

Note that each argument of a term t of the form $f(a_1,...,a_n)$ must have weight smaller than w(f) unless f is the unique monadic operator L with weight 0, in which case a and L(a) have the same weight.

Using these weights we order the Herbrand universe H of ground terms t as follows:

> If $w(t_1) > w(t_2)$ then $t_1 > t_2$ (i.e., lighter terms come first)
>
> elseif $w(t_1) = w(t_2)$, $t_1 == f(a_1,...,a_n)$, $t_2 == g(b_1,...,b_m)$ and f > g then $t_1 > t_2$ (i.e. terms of the same weight are ordered by their principal operator; note that either n or m can be zero, i.e. either f or g can be a constant rather than a function symbol);
>
> elseif $w(t_1) = w(t_2)$, $t_1 == f(a_1,...,a_n)$ and $t_2 == f(b_1,...,b_n)$, then $t_1 > t_2$ if $(a_1,...,a_n) > (b_1,...,b_n)$ in lexicographic order (i.e. terms of the same weight and principal operator are given the lexicographic order of their argument strings).

This recursive definition assigns a position in order to all ground terms. It is legitimate since in its recursive third case each of the arguments of a term like $t_1==f(a_1,...,a_n)$ is either of smaller weight than t_1, or shorter than t_1.

Lemma: The ordering of ground terms just defined is a well-ordering, i.e. there can exist no infinite descending sequence $t_1 > t_2 > t_3 > ...$ of ground terms.

Proof: Suppose that such an infinite descending chain did exist. Then the weights $w(t_j)$ are also non-increasing, and so would necessarily reach their lower limit at some point. Hence we can assume without loss of generality that all the t_j have the same weight and can assume inductively that this is the smallest weight for which an infinite descending sequence $t_1 > t_2$ &gt ... can exist. Consider the sequence f_j of leading operators of the terms t_j. These must be non-increasing (in our assumed ordering of all function symbols of ground terms), and so must also reach their lower limit, so we can assume without loss of generality that all the f_j are identical. Then the f_j cannot be constants (i.e. parameterless function symbols) since if they were we would have $f_j = t_j$, contradicting $t_1 > t_2 > ...$. If they are all function symbols of positive weight, then their argument sequences $(a_1,a_2,...,a_n)$ are sequences of elements of smaller weight descending in lexicographic order, and so by our inductive assumption there cannot be infinitely many of them. It remains to consider the case in which all the f_j are monadic operators f of weight 0, in which case f must be the last operator in the assigned order of operators. In this case, we can write the t_j as $f^{nj}(b_j)$, where n_j designates the number of successive occurrences of f as the principal operator of t_j. Since all the b_j are of equal weight, and f is the last operator in the assigned order of operators, $f^{nj}(b_j) > f^{nk}(b_k)$ if $n_j > n_k$. Hence the integers n_j must form a non-increasing sequence, which will therefore reach its lower limit n at some point. In this case, we can assume without loss of generality that all the n_j are equal, so that our sequence of terms has the form $f^n(b_j)$ for some fixed n, where all b_j have lead operator different from f. Since the b_j must form a decreasing sequence of terms, it follows by what we have already shown that the sequence b_j cannot be infinite, so that we have a contradiction in all cases. **QED.**

(ii) A substitution-invariant partial ordering of non-ground terms

To extend the ordering described above to *non-ground* terms, we give each variable the smallest weight of any constant, and order non-ground terms as follows:

> If $w(t_1) > w(t_2)$ and each variable occurs at least as often in t_1 as it does in t_2, then $t_1 > t_2$;
>
> elseif $w(t_1) = w(t_2)$ and each variable occurs at least as often in t_1 as it does in t_2, while $t_1 ==$ f(...) and $t_2 ==$ g(...) with f > g (in the ordering of operators), then $t_1 > t_2$
>
> elseif $t_1 == f(a_1,...,a_n)$ and $t_2 == f(b_1,...,b_n)$, then we order t_1 and t_2 in the lexicographic order of their argument strings.
>
> Otherwise $t_2 > t_1$ (symmetrically), or t_1 and t_2 are unrelated (which we shall write as $t_1 ? t_2$).

g(x,y,y) and f(x,y,y) give us an example of unrelated terms.

Plainly, if $t_1 > t_2$ and we make a common substitution S for the variables that they contain, writing the substituted results as t_1 o S and t_2 o S then t_1 o S > t_2 o S.

Corollary: There can be no infinite descending sequence $t_1 > t_2 >$ ... of non-ground terms.

Proof: Let S replace all variables by some constant c of smallest possible weight. Then t_1 o S > t_2 o S > ... will be an infinite descending chain of ground terms, which is impossible. **QED.**

Lemma: If , $t_1 > t_2$ and t' is obtained from a term t by replacing one occurrence of t_1 by an occurrence of t_2, then t' > t.

Proof: Plainly w(t') >= w(t), and every variable occurs at least as often in t' as in t. Also, at every level in its syntax tree, t' has function arguments which are at least as large as those of t. **QED.**

(iii) Sets of reductions

A set of identities t = t' is called a *set of reductions* (relative to an ordering of all ground and non-ground terms) if, for each of its members, we have either t > t' or t' > t. In this case we order the identities so that the left side is larger, and write the identity t = t' as t ~> t'.

We can use the elementary identities of group theory as an example of this notion. These involve just two operators, multiplication and inversion, which for ease of reading we write in their usual infix and postfix forms as `x * y` and `x⁻` respectively. The standard elementary identities can be written as simplifications, and then

```
e * x ~> x; x- * x ~> e; (x * y) * z ~> x * (y * z).
```

To order terms formed using these operators, we can use weights w(e) = 1, w(-) = 0, w(*) = 0, and let '-' be the last operator. Note that in this ordering of terms (x * y) * z > x * (y * z) since the leading operators are the same, but x * y > x. That is, 'right-associations are smaller'.

Given a general set of reductions, any term t can be fully reduced (in a non-unique way) by the following procedure;

> Repeatedly find a subterm of t having the form l o S, where S is a substitution and l ~> r is some reduction, and replace this subterm by r o S.

This process must terminate, since it steadily reduces t, in the ordering of terms we have defined.

Definition: If every t reduces to a *unique* final form t*, the set of reductions is said to be *complete.*

We write t => t' if t has a subterm of the form l * S where l~>t is some member of our set of reductions, and t' is obtained by replacing this subterm by r * S.

We write t =>* t' if some such sequence of subterm reductions leads from t to t'.

Lemma: (The 'PPW' -- 'Permanent parting of the ways' Lemma) A set of reductions is complete iff, given any t and two reductions t ~> t' and t ~> t'' of it, there exists some t* such that t' =>* t*, t'' =>* t*.

Proof: If a set of reductions is complete, and t* is the unique full reduction of t, where t ~> t' and t~> t'', then clearly t' =>* t* and t'' =>* t*. Conversely, suppose that t can be fully reduced to two different irreducibles t* and t**, so t =>* t* and t =>* t**. Let t be a minimal element for which this can happen. Then the first steps of these two different reductions must be different. Hence we must have t ~> t_1 =>* t* and t ~> t_2 =>* t**, where t_1 and t_2 are different. By assumption, t_1 and t_2 can be reduced to a common element t_3, which can then be reduced fully to some t***. Thus we have t_1 ~> t_3 =>* t*** and t_2 ~> t_3 =>* t***. One of t* and t** must be different from t***; suppose by symmetry that this is t*. Then t_1 =>* t***, but also t_1 =>* t*. That is, t_1 can be reduced to two different irreducible elements. Since t_1 is less than t, this must be impossible. **QED.**

Definition: We write t ~ t' if there is a chain of subterm substitutions t == t_1 <~> t_2 <~> t_3 <~>...<~> t_n == t', where each t_{j+1} is obtained from the preceding t_j by replacing some subterm of t_j having the form l o S, where S is a substitution and l ~> r is some reduction, by the corresponding r o S, or possibly t_j is obtained from t_{j+1} in this way.

Lemma: A set of reductions is complete iff any two t ~ t' have the same full reduction t*.

Proof of Lemma: If t has two different full reductions t*, t**, then plainly t* ~ t ~ t**, while both t* and t** are their own full reductions. Hence if any two equivalent irreducibles are identical, the set R of reductions is complete. Conversely, let R be complete. Suppose that n is the smallest integer for which there exists a chain t_1 <~> t_2 <~> t_3 <~>...<~> t_n for which t_1 and t_n have different final reductions. Then irrespective of whether t_1 ~> t_2 or t_2 ~> t_1 both have the same final reductions. Hence the two ends t_2 and t_n of the smaller chain t_2 <~> t_3 <~>...<~> t_n would also have different final reductions, a contradiction which proves our lemma. **QED.**

The first lemma stated above implies that if a set of productions is not complete, there exists a 'parting of the ways' t ~> t_1 =>* t_1* and t ~> t_2 =>* t_2* where t_1* and t_2* are fully reduced and different, and where t_1 and t_2 have no common reduction. We call this a '*permanent parting of the ways*'. In this case the reduction t ~> t_1 replaces a subterm w_1 == l_1 o S_1 of t by r_1 o S_1. The reduction t ~> t_2 replaces a subterm w_2 == l_2 o S_2 of t by r_2 o S_1. The two subterms w_1 and w_2 cannot be disjoint (or the paths of reduction would be rejoinable). Hence one replaced part, say w_2 == l_2 o S_2, must be a subterm of the other, i.e. of w_1 == l_1 o S_1.

Let l_1' o S_1 be the subterm of l_1 o S_1 that is actually matched by w_2, i.e. l_1' o S_1 == l_2 o S_2. We can of course write the two identities l_1 ~> r_1 and l_2 ~> r_2 that we are using with disjoint sets of variables. If this is done, then the substitution S_1 on the variables of l_1' and the substitution S_2 on the variables of l_2 can be seen as a common substitution S on all the variables together, and we have l_1' o S == l_2 o S. That is, S is a *unification* of l_1' and l_2. Hence, by the analysis of unification given in the previous section, there is a most general unifier M such that l_1' o M == l_2 o M, and we can write S as the product S == M o T of M and some other substitution T.

Let l_1'' be the result of replacing the subterm l_1' o M of l_1 o M by l_2 o M. Then r_1 o M and l_1'' are two direct reductions of l_1 o M, i.e. l_1 o M ~> r_1 o M and l_1 o M ~> l_1''. These two reductions must themselves be a 'permanent parting of the ways', since if there were further reductions

```
l₁ o M ~> r₁ o M =>* s and l₁ o M ~> l₁'' =>* s,
```

we would also have

```
l₁ o M o T ~> r₁ o M o T =>* s o T     and l₁ o M o T ~> l₁'' o T =>* s o T,
```

implying the existence of reductions t ~> t_1 =>* (some t*) and t ~> t_2 =>* (the same t*), and so the two reductions t ~> t_1 and t ~> t_2 would not be a 'permanent parting of the ways', contrary to assumption. Therefore, if a set R of reductions is not complete, we can find a 'parting of the ways' by unifying the left-hand side of one reduction l_2 ~> r_2 with a subword of the left-hand side of some other reduction l_1 ~> r_1, and then converting the resulting identity to an identity of the form t = t', where both t and t' are irreducible. If the new identity t = t' is not simply t = t, we call it a *superposition* of the two reductions l_1 ~> r_1 and l_2 ~> r_2. As we shall now see, this gives us the key to the Knuth-Bendix procedure.

Testing completeness by superposition of reductions: the Knuth-Bendix completion process.

We saw in the preceding discussion that if a set of reductions is not complete, there exists a pair of reductions l_1 ~> r_1, l_2 ~> r_2, such that we can unify the left-hand side of the second reduction with a subterm of the left-hand side of the first, i.e. find substituted versions of both which allows the left-hand side of the first to be reduced either as a whole or by replacement of a subterm. This yields a pair of versions t, t' of the substituted left-hand side known to be equal. We now reduce both t and t' to their irreducible forms t*, t**. If these are identical, then nothing new results. But if t* and t** are not identical (in spite of the fact that their equality is entailed by the other equalities in our set of reductions) and we can arrange them as a reduction t* ~> t**, then we can extend our set of reductions by adding t* ~> t** to it. Adding t* ~> t** to our original set of reductions clearly refines our notion of reduction, i.e. a term t which was previously irreducible may now admit of further reductions. The Knuth-Bendix method consists in repeatedly adding all non-trivial superpositions of an existing set R of reductions to R, in the hope of eventually reaching a complete set, for which all terms then have a unique canonical form. As we have seen, this would allow us to test two terms to determine whether or not their identity is entailed by our set of reductions just by reducing both of them to the irreducible form and checking these irreducible forms for identity.

More details

When no reorderings of terms become necessary during its operation, the Knuth-Bendix completion process just described is a 'semi-decision algorithm' for determining whether the identity of two terms is entailed by a set of identities. That is, it searches for a completion of the given set of identities, either continuing to search indefinitely and endlessly finding new reductions, or eventually attaining completion. The overall procedure is that implied by the preceding discussion. In more detail, it is as follows.

Suppose that a set l_i ~> r_i of reductions is given.

> Repeatedly resolve the left-hand sides of these reductions with subterms of the right-hand sides of these reductions in all possible ways, generating new pairs of irreducible terms t*, t** known to be equal, in the manner described in the preceding section. Arrange t* and t** as a reduction l* ~> r*. Add these reductions l* ~> r* to the set of reductions, using the same weights and operator ordering if possible.
>
> Change the weights and operator ordering if necessary. (These play only an auxiliary role).
>
> Each time a new reduction l* ~> r* is added, retest every other to see if it is now *subsumed*, and if so drop it from the set of reductions.

Definition: A reduction l ~> r belonging to a set C of reductions is *subsumed* by the other members of C iff l and r are both reducible to a common element by the set C - {l ~> r} obtained from C by dropping the reduction l ~> r.

We continue adding new reductions until the resulting set becomes complete, or until whatever conclusion t = t' we wish to test reduces to t = t. If this process runs unduly long we stop it.

(B) Examples of the Knuth-Bendix procedure

(1) Simple associativity. First we consider what is almost the simplest possible system, that involving just a single dyadic operation (which we will write in infix form), and just one identity, namely the associative law

```
(x * y) * z = x * (y * z)
```

All weights, including w(*), are taken to be 1. The Knuth-Bendix ordering rule then gives

```
(x * y) * z > x * (y * z)
```

since (x * y) > x, and so our system is seen to consist of the one reduction

```
(x * y) * z ~> x * (y * z).
```

This makes it clear that, in this system, term reduction consists in using the associative law to move parentheses to the right, so that the irreducible form of a term is its fully right-parenthesized form. This makes it clear that irreducible forms are unique in this simple system, so that our single reduction R is already complete. To verify this using the formal Knuth-Bendix criterion, note that the only way of unifying the left side of R with a subterm of the left side of R (first rewritten using different variables) is to unite (x * y) * z with the subword (u * v) of (u * v) * w. The unifying substitution converts this second term to ((x * y) * z) * w, which as we have seen in our general discussion can be reduced in two ways to produce (x * (y * z)) * w and (x * y) * (z * w). But in this case nothing new results since both of these terms have the same right-parenthesized form.

(2) Minimal axioms for the theory of free groups. We now examine a more elaborate and interesting example, that of the elementary identities of group theory touched on earlier. These involve just two operators, multiplication and inversion, which for ease of reading we write in their usual infix and postfix forms as x * y and x^- respectively. As before, we use weights w(e) = 1, w(-) = 0, w(*) = 0, and let '-' be the last operator. We begin with identities which state the existence of a left identity and left inverses, along with associativity. These are

```
[P1] e * x ~> x;  [P2] x- * x ~> e;   [P3] (x * y) * z ~> x * (y * z).
```

Knuth-Bendix analysis of these identities will show that these initial identities imply that the left identity is also a right identity and that the left inverse is also a right inverse. (The reader may want to improve his/her appreciation of the Knuth-Bendix procedure by working out direct proofs of these facts). We begin as follows: Superpose [P2] on [P3], getting:

```
[P4] x- * (x * z) ~> z.
```

Now superpose [P1] on [P4], getting:

```
[P5] e- * z ~> z.
```

Superpose [P2] on [P4], getting $x^{--} * (x^- * x) \sim> x$, or

```
[P6] x- - * e ~> x.
```

Superpose [P6] on [P3], getting $(x^{--} * e) * z \sim> x^{--} * (e * z)$, or

```
[P7] x- - * z ~> x * z.
```

[P6] is now replaced by

```
[P8] x * e ~> x.
```

Thus the left identity is a right identity, and [P6] reduces to

```
[P9] x- - ~> x.
```

Now [P8], [P5] superpose to give

```
[P10] e- ~> e.
```

Now [P2] and [P9] superpose to `x ⁻ ⁻ * x⁻ ~> e,` or

```
[P11] x * x-  ~> e.
```

Thus the left inverse is also a right inverse. Two more derived identities complete the set:

```
e * x ~> x; x * e ~> x;  x- * x ~> e; x * x-  ~> e;  e- ~> e; x- - ~> x; (x * y) * z ~> x * (y * z);  x- *
```

The normal form of any term in this theory is obtained by expanding it out using `(x * y)⁻ ~> y⁻ * x⁻` as often as possible, associating to the right, performing as many cancellations `x * x⁻ ~> e, x⁻ * x ~> e, x⁻ ⁻ ~> x` as possible, and removing e from all products. This is of course a standard normal form for the elements of free groups.

3.7. A decision algorithm for the theory of totally ordered sets

The (unquantified) theory of totally ordered sets allows variables designating elements of such a set, and un-negated or negated comparisons '>' and '=' between such elements. The comparison operator is assumed to satisfy all the assumptions standard for such comparators, i.e.

```
(FORALL x,y,z | (x > y & y > z) *imp (x > z))     (FORALL x,y | (x > y) *imp (not(y > x or y = x))
```

Since for the elements of such a set not(x > y) is equivalent to (y > x or y = x) and x /= y is equivalent to (x > y or y > x), we can eliminate all the negated comparisons and thus have only to decide the satisfiability of a conjunction of comparisons, some of the form x > y and others of the form x = y. By identifying all pairs of variables x,y for which a conjunct x = y is present, we can eliminate all occurrences of the'=' operator, and so have only to consider conjunctions of inequalities x > y. Such a conjunct is satisfiable if and only if it contains no cycle of relationships x > y. Indeed, if there is such a cycle it is clear that the given set of statements admits of no model by the elements of a totally ordered set. Conversely, if there is no such cycle, our variables can be topologically sorted into an order in which x comes later than y wherever x > y, and this very ordering gives us the desired model.

A related and equally easy decision problem is that for the (unquantified) **elementary theory of subsets of totally ordered sets.** This is the language whose variables s,t designate subsets of some totally ordered set U, whose operators are the elementary set union, intersection, and difference operators +, *, and -, whose comparators are 'incs' and '=', but where we also allow the comparator s > t (and also s >= t) which states that every element of s is greater (in the given ordering of U) than every element of t. We want this language to describe subsets of some universe of totally ordered sets, so we define models of any collection S' of statements in the language to be a mapping of the variables which appear in S' into subsets of some totally ordered set U with ordering '>', such that s > t and s >= t are respectively equivalent to

```
(FORALL x in s, y in t | x > y) and (FORALL x in s, y in t | x >= y).
```

To handle this language, it is convenient to make use of the notion of 'place' introduced in our earlier discussion of decision algorithms for the language of elementary set operators (Section XXX), and of the properties of that notion defined in Section XXX. As usual, we reduce the satisfiability problem that confronts us to the satisfiability problem for a collection of conjuncts, each having one of the following forms:

```
(*) s = t + u   s = t - u   s = t * u   s = 0   s /= 0       s > t   s >= t   not(s > t)   not(s >= t).
```

Let S' be the set of all conjuncts listed above, and let S be the subset consisting of all those conjuncts listed in the first line of (*). We saw in section XXX that, given any model N of S, and any point p in the universe U of such a model, the function f_p(s) *eq (p in Ms) defines a *place* for S, i.e. a Boolean-valued mapping of the variables and elementary expressions appearing in S, such that

```
  fp(s + t) = fp(s) or fp(t), fp(s * t) = fp(s) and fp(t),     fp(s - t) = fp(s) and (not fp(t)), fp(0) = fals
```

We also saw in Section XXX that the set of all points p in U defined an *ample* set of places, in the sense that for any conjunct of the form s /= 0 there must exist a place f_p such that f_p(s) = true. Conversely, given any ample set P of places, the formula Ms = {f in P | f(s) = true} defines a model of the set S of conjuncts.

For our present purposes we need a slight reformulation of this result which allows individual places f to be used more than once in a model. In this reformulation we use not simple a set P of places, but a finite sequence P' of places. We call such a sequence of places ample if the set of places f_i that occur in it is ample. In this case, it is easily seen that the modified formula

```
(**)  Ms = {i | fi(s) = true}
```

also defines a model of the subset S of conjuncts. Suppose now that the full set S of conjuncts has a model with some universe U, where as said above U must be ordered and its ordering '>' must model the operator s > t of our language in the manner indicated above. For every conjunct not(s > t) (resp. not(s >= t)) in S' choose a pair of points p,q in U such that p in Ms, q in Mt, and q >= p (resp. q >= p). To these points, add a point p in Ms for every conjunct s /= 0 in the set S' of conjuncts. It is then clear that if we restrict our universe to this collection U' of points, i.e. take M's = Ms * U' for every variable s of S', we still have a model of the full set S' of conjuncts. If these points p_j are be arranged in their '<' order, we will have $p_j > p_k$ if j > k. Now consider the sequence of places f_j corresponding to these points, i.e. $f_j = f_{p_j}$. These have the property that if f_j(s) = true (equivalent to 'p_j in s'), and also f_k(t) = true, then the presence in S' of a conjunct s > t (resp. s >= t) implies j > k (resp. j >= k). Moreover, the presence in S' of a conjunct not(s > t) (resp. not(s >= t)) implies the existence of indices j,k such that k >= j (resp. k > j) such that f_j(s) = true and f_k(t) = true. Hence if we take the M defined by formula (**), whose universe U is simply the set of integer indices of the finite sequence P' of places, and give these points their ordinary integer ordering, M is a model of our full set S' of conjuncts. This establishes the following conclusion, which clearly implies that the language presently under consideration has a solvable satisfiability problem:

A a collection S' of conjuncts of the form (*) is satisfiable if and only if it admits an ample sequence f_j of places, in which no place occurs more than n + 1 times, where n is the total number of conjuncts having either the form not(s > t), or the form not(s >= t).

3.8. A decision algorithm for ordered Abelian groups

Ordered Abelian groups G are characterized by the presence of an associative-commutative addition operator '+', with identity '0' and inverse '-', and also a comparison operator x > y satisfying

```
  (FORALL x in G | (not (x > x)) &         (FORALL y in G | x > y or x = y or x < y))    (FORALL x in G, y
```

The last axiom plainly implies that

```
(FORALL x in G, y in G, z in G | (x > y) *imp (x + z) > (y + z))
```

We will show in this section that the satisfiability of any finite collection C of unquantified statements in this theory, i.e. unquantified statements written using operators '+' and '>' subject to the above axioms, is decidable. To this end, note first that if such a conjunction C is satisfiable, i.e. has some model which is an ordered Abelian group G', it can plainly be modeled in the subgroup G of G' generated by the elements of G' which correspond to the symbols which appear in the statements of C. Hence C has a model which is an ordered Abelian group with finitely many generators. Conversely, if there exists such a model, then C is satisfiable. Thus we can base our analysis on an understanding of the structure of finitely generated ordered Abelian groups G.

The additive group of reals contains many such ordered subgroups with finitely many generators, as does the additive group of real vectors of dimension d for any d if we order these vectors lexicographically. We will see in what follows that these examples are generic, in the sense that any finitely generated ordered Abelian group can be embedded into one of these two groups by an order-preserving isomorphism (we will call such isomorphisms 'order-isomorphisms' in this section.)

This can be done as follows: By a well-known result, Abelian groups with finitely many generators are decomposable, in an essentially unique way, as direct sums of finitely many copies of the integers and of finitely many finite cyclic groups. The order axiom plainly rules out any finite cyclic components, so G must be the direct sum of finitely many copies of the integers. We denote by 'rank(G)' the number of these copies that appear in the direct sum representing G. A standard result, whose proof we will repeat below, tells us that this number depends only on G, not on the way in which G is represented.

To see how the order in G must be represented, we will consider two cases separately: that in which G has 'infinitesimals', and that in which it does not. To this end, we define the subgroup Inf(G) of infinitesimals of G as follows:

```
Inf(G) = {x in G | there exists a y in G such that mx < y            holds for all signed integers m},
```

where for m > 0, mx designates the sum of m copies of x; mx is the zero element of G if m = 0, and mx = -(-m)x if m < 0.

It is easy to show that Inf(G) is indeed a subgroup of G, and we leave this to the reader.

First suppose that G contains no 'infinitesimals', i.e. that for each x > 0 and y > 0 there exists a positive integer n such that nx > y. In this case we can show that the group must be order-isomorphic to an ordered subgroup of the additive group of reals. In the easy case in which there is just one generator, G is plainly isomorphic to the ordered group of integers. More generally, choose some y > 0, and then, for each x, consider the set S(x) of all rationals m/n with positive denominator n such that nx > my. This is defined independently of the way that m/n is represented by a fraction, since the order axioms imply that nx > my *imp knx > kmy for each positive k, and conversely if knx > kmy then nx <= my is impossible. Also, for every x, there is a positive integer n such that not(n in (S(x)) and (-n in S(x)), so S(x) is neither empty or all the rationals. Moreover S(x) is bounded above, because if not(m/n in S(x)) (i.e., my >= nx) and m'/n' > m/n (i.e., nm' > mn'), then not(m'/n' in S(x)) (i.e., m'y >= n'x). Finally, if m/n in S(x) then there are m', n' such that m'/n' in S(x) and m'/n' > m/n. Together these facts imply that S(x) is a cut in the set of rationals, i.e. that there is a unique smallest real r(x) such that S(x) = {a: a < r(x)}.

The mapping r maps G to the reals in an order-preserving manner. Indeed, if x' > x, and the rational number m/n (with positive denominator) belongs to S(x), then nx > my, and so nx' > my also, proving that m/n belongs to S(x'). That is, x' > x implies that S(x') incs S(x), and thus plainly implies that r(x') >= r(x). Suppose now that m/n < r(x) and that m'/n' < r(x'), both denominators n and n' being positive. Then nx > my and n'x > m'y, so nn'x > mn'y and nn'x > m'ny, and therefore

```
nn'(x + x') > (mn' + m'n)y,
```

from which it follows that m/n + m'/n' belongs to S(x + x'). This proves that r(x + x') >= r(x) + r(x'). Now suppose that r(x + x') > r(x) + r(x'), and let m/n and m'/n' respectively be rationals which approximate r(x) (resp. r(x')) well enough from above so that we have m/n + m'/n' < S(x + x'), while m/n > r(x) and m'/n' > r(x'). This implies that nx <= my, n'x <= m'y, and nn'(x + x') > (mn' + m'n)y. This is impossible since our first two inequalities imply that nn'(x + x') <= (mn' + m'n)y. It follows that r(x + x') > r(x) + r(x') is impossible, so r(x + x') = r(x) + r(x'), i.e. r is a homomorphism of G into the ordered group of reals. Finally, suppose that r(x) = 0. Then we cannot have x > 0, since if we did then nx > y would be true for some positive n, so 1/n would be a member of S(x), implying that r(x) >= 1/n, which is impossible. Similarly if x < 0 it would follow that r(-x) >= 1/n for some positive n, also impossible. Since r has been seen to be additive r(-x) = -r(x), and it follows that x must be 0, proving that r is an order-isomorphism of G into the reals. This completes our treatment of the case in which G has no infinitesimals.

Next we will show that in the presence of infinitesimals, namely when Inf(G) is nontrivial, G can be embedded into the lexicographically ordered additive group of real vectors of dimension not more than rank(G). To handle this case, we need to use a few more standard results about finitely generated Abelian groups, which we pause to derive. The first of these is the fact that rank(G) is independent of the way in which we represent G as the sum of a finite collection of cyclic groups, i.e. as an additive group N_k of integer vectors of length k. To see this, suppose that two such groups N_k and $N_{k'}$ are isomorphic, and that k > k'. Let f be an isomorphism of N_k onto $N_{k'}$. If we embed N_k and $N_{k'}$ into the corresponding spaces N^*_k and $N^*_{k'}$ of vectors with rational coefficients, and extend f to a linear mapping of N^*_k into $N^*_{k'}$, then, since the dimension k of N^*_k exceeds that of $N^*_{k'}$, there exists a nonzero rational vector, and hence a nonzero integer vector in N^*_k which f maps to zero. This contradicts the fact that f is an isomorphism, and so proves our assertion concerning rank(G).

Next we will show that any subgroup S of a finitely generated ordered group G is also finitely generated, and has rank no greater than the rank of S, again a standard result. By what has been proved above, we can suppose without loss of generality that G is the additive group of integer vectors of dimension d. If there is no vector v in S whose first component is nonzero, then S is a subgroup of the group of integer vectors of dimension d - 1, and so (by our inductive hypothesis) there is nothing to prove. Otherwise let v be such a vector with smallest possible first component c_1. Then any other V' in S must have a first component c'_1 which is divisible by c_1, since otherwise the greatest common divisor of c'_1 and c_1, which is the first component of some vector of the form k*v + k'*v', where k and k' are integers, would be positive and smaller. It follows that every v in S can be written in the form k*v + u, where u is a vector in S whose first component is 0. Therefore, if we let S' be the subgroup of S consisting of all vectors whose first component is 0, S' is a subgroup of the additive group of integer vectors of dimension d - 1. By inductive hypothesis, S' is a finitely generated group with at most d - 1 generators. If we add v to this set of generators, we clearly have a set of generators for S, proving our assertion.

In what follows we will also need to use the following facts.

Lemma. Let (G,<) be a finitely generated ordered Abelian group, and let B be a subgroup of G such that

```
(*) x < y for each x in B and each positive y in G - B.
```

Then

```
(1) if given the ordering '<' defined by        (g + B) < (g' + B) iff g < g' and (g + B) /= (g' + B).
```

Proof: To prove (1) , note first of all that the relationship (g + B) < (g' + B), i.e. g < g', is independent of the elements g and g' chosen to represent (g + B) and (g' + B). For if other g and g' were chosen, the difference g' - g will change to g' - g + b, where b is some element of B. But since g' - g is positive, we must have -b < g' - g by assumption (*), so g' - g + b is positive also. Knowing this, we see at once that the relationship (g + B) < (g' + B) is transitive, and that if (g + B) < (g' + B) and (h + B) < (h' + B), then ((g + h) + B) < ((g' + h') + B), proving (1).

(2) Follows immediately from (1), since the Cartesian product of any two groups, lexicographically ordered, is always an ordered group. To prove (3), note that

(a) B is a subgroup of a finitely generated Abelian group, and so (as proved above) is finitely generated.

(b) if g_1, ..., g_n is a system of generators of G, then $(g_1 + B)$, ..., $(g_n + B)$ is a system of generators of G/B (not necessarily a minimal set of generators), so that G/B is finitely generated.

Let $\{h_1 + B,...,h_p + B\}$ be a minimal set of generators of G/B. Let T be the map from G onto G/B defined as follows:

For each g in G there exist unique integers $k_1,...,k_p$ such that

$$g + B = k_1(h_1 + B) + ... + k_p(h_p + B).$$

Using these k_j, put

$$T(g) = [g + B, g - (k_1h_1 + ... + k_ph_p)].$$

It is not difficult to verify that for any two g, g' in G we have T(g - g') = T(g) - T(g'). Moreover, if T(g) = 0, we must have g + B = 0, so $k_1,...,k_p$ must all be zero, and therefore g = 0. This shows that T is an isomorphism of G onto the Cartesian product group C of G/B and B. Since the rank of a finite group is independent of its representation, we also have rank(G) = rank(G/B) + rank(B). To show that T is also an order-isomorphism from (G,<) onto the lexicographically ordered Cartesian product C of G/B and B, suppose that g' >g, and write g'+ B as

$$g' + B = k'_1(h_1 + B) + ... + k'_p(h_p + B).$$

Then if g' + B /= g + B we have g' + B > g + B by (1) above, so

$$[g + B, g - (k_1h_1 + ... + k_ph_p)] > [g' + B, g' - (k'_1h_1 + ... + k'_ph_p)].$$

On the other hand, if g' + B = g + B we have $k_j = k'_j$ for all j, and so

$$[g + B, g - (k_1h_1 + ... + k_ph_p)] > [g' + B, g' - (k_1h_1 + ... + k_ph_p)]$$

in this case also. Hence this inequality holds in any case, i.e. T is both an isomorphism and an order-isomorphism. **QED**

Assume as above that Inf(G) is nontrivial. Then the condition (*) of the previous Lemma holds for the proper subgroup Inf(G) of G. Indeed, if x is infinitesimal and y is positive and not infinitesimal, and x < y is false, then y < x. Since x is infinitesimal there exists some positive z such that mx < z for all integers m. Then plainly my < z for all positive integers m, and since y is positive this holds for all negative integers m also. It follows that y is infinitesimal, a contradiction proving our assertion.

It follows from (2) and (3) above that (G,<) is isomorphic to the lexicographically ordered Cartesian product C of G/Inf(G) and Inf(G), and that rank(G) = rank(G/Inf(G)) + rank(Inf(G)). Moreover

$$\text{(i) } G/Inf(G) \text{ is nontrivial, i.e. } rank(G/Inf(G)) > 0; \quad \text{(ii) } G/Inf(G) \text{ has no infinitesimals but 0.}$$

To prove (i), note that if G/Inf(G) were trivial, i.e. Inf(G) = G, all elements, and in particular all generators, of G would be infinitesimals. Thus for each generator g_j there would exist a positive y_j such that $m_jg_j < y_j$ for every integer m_j. Let y be the sum of all these y_j. Then plainly $m_jg_j < y$ for every integer m_j. But y is itself a sum

$$y = k_1g_1 + ... + k_pg_p.$$

It follows that

$(k_1g_1 + ... + k_pg_p) < y_1 + ... + y_p = y = (k_1g_1 + ... + k_pg_p),$

a contradiction which shows that G/Inf(G) is nontrivial.

To prove (ii) we argue as follows. Suppose that g + Inf(G) is infinitesimal in G/Inf(G), i.e. that there exists a positive y + Inf(G) in G/Inf(G) such that m(g + Inf(G)) < y + Inf(G) for all integers m. This gives mg < y for all integer m, and then g is plainly infinitesimal in G, so it must belong to Inf(G), i.e. g + Inf(G) must be the zero element of G/Inf(G), proving (ii).

Since G/Inf(G) is nontrivial by (i), we must have rank(Inf(G)) < rank(G). Now applying (3) inductively, it follows that G is isomorphic to the lexicographically ordered Cartesian product of the sequence

$G/Inf(G),\ Inf(G)/Inf_2(G),...,\ Inf_{k-1}(G)/Inf_k(G),\ Inf_k(G),$

of groups for each k < rank(G), where by definition $Inf^i(G) = Inf(Inf^{i-1}(G))$. By (ii), each group in this sequence is a finitely generated Abelian group with no nontrivial infinitesimals. Since, as was shown above, each such group can be embedded into the additive group of reals, it follows that G can be embedded into the additive group of real vectors of dimension rank(G), ordered lexicographically. This is the key conclusion at which the preceding arguments aimed.

It follows from what has now been established that, given any quantifier-free conjunction F of statements in the theory of ordered Abelian groups which contains n distinct variables, F is satisfiable in some ordered Abelian group if and only if it is satisfied in the additive group of real vectors of dimension n, ordered lexicographically. But it is easy to reduce the satisfiability problem for the lexicographically ordered additive group of real vectors of dimension n to the satisfiability problem for the additive group of reals. Indeed, a real vector of dimension n is just a collection of n real numbers $x_1,...,x_n$, addition of two such vectors is just addition of their individual components, and the condition x < y for two vectors x and y can be written as the disjunction

$x_1 < y_1 \text{ or } (x_1 = y_1 \ \&\ x_2 < y_2) \text{ or...or } (x_1 = y_1 \ \&...\&\ x_{n-1} = y_{n-1} \ \&\ x_n < y_n).$

This observation shows that the satisfiability problem for any collection of statements in the theory of ordered Abelian groups reduces without difficulty to the problem of satisfying a corresponding collection of real linear equations and inequalities. This is the standard problem of linear programming, which can be tested for solvability using any convenient linear programming algorithm.

3.9. 'Blobbing' more general formulae down to a specified decidable or semi-decidable sublanguage of set theory

'Blobbing' captures Aristotle's insight that statements true in logic are true because their form can be matched to some template known to generate true statements only. The basic syntactic technique which it involves can be explained as follows: Suppose that we are given a language L for which a full or partial decision algorithm is available. Then any formula F can be 'reduced' or 'blobbed down' to a formula in the language L, in the following way. Work top-down through the syntax tree of F, until some operator not belonging to the language L is encountered. Replace the whole subformula G below this level by a 'blob', i.e. by a freshly generated term or element of the form $g(x_1,x_2,...,x_k)$, where $x_1,x_2,...,x_k$ is the set of all variables free in G.

'Blobs' generated in this way from separate subformulae of F should be made identical wherever possible; this can be done whenever they are structurally identical up to renaming of bound variables, or where part of the structure belongs to an equational theory for which a decision or semidecision algorithm is available. The 'blobbed' variant of F is the formula that results from this replacement. If the blobbed version of F is a consequence of the blobbed versions of the union of all our previous assumptions and conclusions, then F follows from these assumptions and conclusions, and can therefore be added to the set of available conclusions. 'Default' or 'ELEM' deduction is the special case of this general observation which results when we blob down to an extended multi-level syllogistic, of the kind described above.

The following is an example of 'Blobbing'. Given the input formula

```
({m(x): x in s | x *incin t} *incin         {m(x): x in s | x in t}       & {m(x): x in s | x in t} *inc
```

its blobbed version (blobbed down to the elementary theory of sets and inclusion relationships) is:

1_ *incin 2_ & 2_ *incin 3_ *imp 1_ *incin 3_

which makes the truth of our original, rather enigmatic formula obvious.

3.10. Computation with hereditarily finite sets

Set theory, as we will work with it later in this book, is a family of sentences concerning objects, some of which lie far beyond the finite realm within which conventional computational mechanisms can operate. But the hereditarily finite sets considered earlier are accessible to computation and can model all standard computational processes in quite a satisfactory way. These are the sets which can be constructed starting from the null set {} by repeatedly forming sets of the form $\{s_1,s_2,...,s_n\}$ using elements $s_1,s_2,...,s_n$ previously defined. Four examples are

{}, {{}}, {{},{{}}}, {{{{},{{}}}}}, etc.

We can readily define the standardized representation rep(s) of each such set s recursively as

$$rep(s) = \{rep(s_1),rep(s_2),...,rep(s_n)\},$$

where we assume that (after duplicates have been removed) the elements on the right are sorted, first by the length of the strings used to write them, strings of the same length then being arranged in lexicographic order. Any suitable computer encoding of the system of strings defined in this way can be used as the basis of a system for programmed computation with (entirely general) hereditarily finite sets.

Hereditarily finite sets satisfy several general induction principles. If there exists an hereditarily finite set (resp. finite) t satisfying P(t), where P is any predicate, then there also exists an hereditarily finite (resp. finite) set t' satisfying

P(t') & (FORALL t *incin t' | (t /= t') *imp (not P(t))).

A second induction principle that is sometimes easier to apply is as follows. If there exists an hereditarily finite set t satisfying P(t), where P is any predicate, then there also exists an hereditarily finite set t' satisfying

P(t') & (FORALL t in t' | not P(t)).

This second induction principle remains valid for infinite sets.

An adequate computational system based on hereditarily finite sets needs only the following modest collection of primitives.

(i) Given a set s, we can form the singleton set {s}. (This primitive function maps sets to sets).

(ii) Any two sets can be tested for equality (they are equal if and only if their standard representations are the same). (The primitive $s_1 = s_2$ maps pairs of sets to Boolean values).

(iii) Any set s can be tested for membership in any other. s_1 is a member of s_2 if and only if s_1 is equal to one of the items in the list of members defining s_2. (The primitive s_1 in s_2 maps pairs of sets to Boolean values).

(iv) We can find an element arb(s) of any set s other than the null set. (This primitive function maps sets to sets). It is convenient to let x = arb(s) be the first element of s in the order of elements described above. This ensures that arb(s) and s have no element in common, since if there were any such element y, then clearly y would have a string representation shorter than that of x, and so y would come before x in the standard order of elements of x, contradicting our assumption that x is the first of these elements. (To complete the definition of the function 'arb', it is convenient to put arb({}) = {}).

(v) Given two sets s_1, s_2 we can form the set 's_1 with s_2' obtained by adding s_2 to the list of elements of s_1. If s_2 is already on this list, then 's_1 with s_2' is just s_1. (This primitive maps sets to sets).

(vi) Given two sets s_1, s_2 we can form the set 's_1 less s_2' obtained by removing s_2 from the list of elements of s_1. If s_2 is not on this list, then 's_1 less s_2' is just s_1. (This primitive also maps sets to sets).

(vii) New set-valued and new Boolean-valued functions of hereditarily finite sets can be introduced by writing (direct or recursive) definitions

```
function name(s1,s2,...,sn);    return if cond1 then expn1      elseif cond2 then expn2      ...      elsei
```

Here all the $cond_1$,...,$cond_m$ must be nested, Boolean-valued expressions built using primitive or previously-defined function names, plus the elementary Boolean operations &, or, not, etc. and the variables s_1,s_2,...,s_n. Either all the $expn_1$,$expn_2$,...,$expn_{m+1}$ must be set-valued, in which case the defined function 'name' is also set-valued, or all the $expn_1$,$expn_2$,...,$expn_{m+1}$ must be Boolean-valued, in which case the defined function 'name' is also Boolean-valued.

Function definitions of this type can be used to program many other basic and advanced set-theoretic functions. For example, we can write

```
function union(s1,s2);     return if s1 = {} then s2       else union(s1 less arb(s1),s2) with arb(s1) end i
```

Note that the hereditarily finite sets s for which is_integer(s) is true are precisely those of the recursive form

{{},{{}},{{},{{}}},...,prev(s)}

which represent integers in their von Neumann encoding. For such integers n we have last(n) = n - 1, last(n - 1) = n - 2 etc.

From here we can easily go on to define the cardinality operator and all the standard arithmetic operations, e.g.

```
function #s; return if s = {} then {}         else next(#(s less arb(s))) end if;   end #;  function sum(
```

Our next group of procedures lets us work with maps:

```
function ordered_pair(s1,s2);     return {{s1},{{s1},{s2,{s2}}}};   end ordered_pair;  function car(s); ret
```

Another useful notion is 's_1 is a sequence of elements of s_2':

```
function is_sequence(s);     return is_map(s) & is_single_valued(s)       & is_integer(domain(s));   end
```

```
function is_sequence_of(s1,s2);     return is_sequence(s1)       & difference(range(s1),s2) = {};   end is_
```

Function definitions like those seen above are said to be 'mirrored in logic' if for each function definition

```
function name(s1,s2,...,sn);    return if cond1 then expn1      elseif cond2 then expn2      ...      elsei
```

we have defined a corresponding logical symbol 'Name' for which the statement

```
(FORALL s1 in HF,s2 in HF,...,sn in HF |     Name(s1,s2,...,sn) = if cond1 then expn1      elseif cond2 then
```

is available as a theorem. (In case 'name' is Boolean-valued, '*eq' must supersede '=' in the above statement). It will now be shown that every one of the function definitions given above and all others like them can be mirrored in logic. This lets us use the following general lemma as one of our mechanisms of deduction.

Mirroring lemma

Mirroring lemma: Let name(s_1,s_2,...,s_n) be a set-valued function appearing in a sequence of functions defined in the manner described above, and let c_1,c_2,...,c_{n+1} be hereditarily finite sets represented by logical terms e_1,e_2,...,e_{n+1} as described above. Suppose that the calculated value of name(c_1,c_2,...,c_n) is c_{n+1}. Suppose that 'Name' is the logical function symbol which mirrors 'name'. Then the formula

```
Name(e1,e2,...,en) = en + 1
```

is a theorem. Similarly, if 'name' is a Boolean-valued function, then

```
Name(e1,e2,...,en) *eq en + 1
```

is a theorem.

Proof: Consider the number of steps h involved in an evaluation of a function having a definition like

```
function name(s1,s2,...,sn);    return if cond1(s1,s2,...,sn) then expn1(s1,s2,...,sn)      elseif cond2(s1,s
```

and suppose that our assertion is true for all such evaluations having fewer steps than h. The final step in an evaluation of this recursive function will end at some branch, say the k-th branch, of the conditional expression following the keyword 'return', and will be preceded by evaluation of all the conditions

```
cond1(s1,s2,...,sn),cond2(s1,s2,...,sn),...,         condk(s1,s2,...,sn)
```

which appear before this branch, and of the expression

```
expnk(s1,s2,...,sn)
```

occurring in the k-th branch. This last expression will return some value 'val', which then becomes the value returned by the function 'name'. In the situation considered, the first k - 1 Boolean conditions must have the value 'false' and the k-th must have the value 'true'. Since all of these subevaluations must involve fewer steps than h, there must exist proofs of the theorems

```
not Cond1(s1,s2,...,sn), not Cond2(s1,s2,...,sn),...,    not Condk - 1(s1,s2,...,sn), Condk(s1,s2,...,sn),
```

and there must also exist a proof of the statement

```
Expnk(s1,s2,...,sn) = val
```

It follows from these results that we can prove

```
if Cond1(s1,s2,...,sn) then Expn1(s1,s2,...,sn)      elseif Cond2(s1,s2,...,sn) then Expn2(s1,s2,...,sn)
```

Since

```
(FORALL s1,s2,...,sn | Name(s1,s2,...,sn) =   if cond1 then expn1      elseif cond2 then expn2      ...
```

this proves that

```
Name(s1,s2,...,sn) = val.
```

QED

Note that the mirroring lemma only provides us with a way of proving statements giving particular constant values of recursively defined functions and predicates, but not a way of proving any universally quantified statement. For example,

```
Domain(With({Ordered_pair(0,0)},Ordered_pair(1,1))) =             With({0},1)
```

follows by mirroring, but no universally quantified statement like

```
(FORALL x | (Is_integer(x) *imp (x = {} or     (x = Next(Prev(x)) & Is_integer(Prev(x))))))
```

can be proved simply by mirroring. In fact, even a statement like

```
Domain(With({Ordered_pair(0,x)},Ordered_pair(1,y))) =             With({0},1)
```

lies beyond the reach of the mirroring lemma, since it involves the symbolic variables x and y.

Deduction by Semi-symbolic Computation

A useful and much more general 'Deduction by semi-symbolic computation' primitive is included in our verifier. This operation lets us use recursive relationships as means of easy computational deduction within the completely controlled environment in which a verifier must operate. The idea is to evaluate certain elementary operations on hereditarily finite sets explicitly, while leaving all other expressions unchanged. Recursive relationships are used as long as they apply, allowing complex identities and logical equivalences to be derived by single deduction steps. We shall see that this makes a wide variety of 'Mathematica'-like conclusions directly available within the verifier.

A prototypical example is furnished by the recursive identity

```
(is_seq(B) & m in domain(B) & range(B) *incin Z) *imp           SIG(B | m) =         if m = 0 then 0 else
```

(where Z denotes the set of integers, and the predicate is_seq(t) is true if t is a sequence, i.e. a mapping whose domain is either a finite integer or the set Z of all integers). As we shall see later, this simple general theorem is easily proved (in a slightly different notation) using the definition of the summation operator SIG. Deduction by semi-symbolic computation lets us apply this in the case m = 100 to get the theorem

(is_seq(B) & 100 in domain(B) & range(B) *incin Z) *imp
SIG(B | 100) = 0 + B[0] + B[1] + ... B[100],

as an immediate conclusion. Clearly derivations of statements like this would otherwise require tediously lengthy and repetitive sequences of steps. Another common case is the evaluation of expressions like

$$\text{Value\_at}(j,\{\text{Ordered\_pair}(0,x_0),\text{Ordered\_pair}(1,x_1),...,\text{Ordered\_pair}(k,x_k)\}),$$

which it would otherwise be tedious to deal with but which are easily handled by semi-symbolic computation.

But in fact the method of deduction by semi-symbolic computation is much more general. To explain this assertion, we must first define an appropriate relationship between the language of computation with hereditarily finite sets and the purely set-theoretic language of the verifier. This can be done as follows. We first define the relator x *inin s (x is an

eventual member of s) by the following formula:

(x *inin s) *eq (x in s or (EXISTS y in s | x *inin y))

Operations on hereditarily finite sets which can be defined and evaluated recursively include all the propositional connectives, $s_1 + s_2$, $s_1 * s_2$, $s_1 - s_2$, all elementary set comparisons, all quantifiers over hereditarily finite ranges, all set formers over hereditarily finite ranges, #s, all elementary arithmetic operations, the operations car(s) and cdr(s) for pairs (note however that the choice operator arb(s) cannot be calculated in this way), the pair-former [x,y], the set operators range(s), domain(s), Un(s), pow(s), the predicate is_single_valued(s), set and tuple formers by enumeration, the operator f{s}, the operator which evaluates the range of the restriction of a hereditarily finite map f to an hereditarily finite set s, the Cartesian product operator s_1 *PROD s_2, the inverse-map operator inv_map(s), the map-restriction operator f|s, the concatenation operator s_1 cat s_2, if-expressions, case-expressions, and various others. By the mirroring lemma proved above, the value produced when one evaluates such an expression is always logically equal to the original expression, provided of course that the operator signs appearing in the expression have their standard meanings.

Next suppose that an implication of the form

P(B) & s in HF *imp F(s,B) = if C_1(s) then e_1(s,B)
elseif C_2(s) then e_2(s,B) ... elseif C_n(s) then e_n(s,B) else F(s,B) end

has been shown to hold for all hereditarily finite sets s. We assume here that the conditions C_j(s) involve only the elementary operations listed above. However, the expressions e_j(s,B) on the other hand can be more general, and involve: (i) subexpressions $e^{(1)}_j$(s) in which only elementary operations appear; (ii) recursive appearances F($e^{(2)}_j$ (s),B) of the function F, in which the parts $e^{(2)}_j$ (s) contain only elementary operations; (iii) other subexpressions; (iv) other recursive appearances F($e^{(3)}_j$ (s,B),$e^{(4)}_j$ (s,B)) of F. This implication can be used as a recursive procedure in the following way: suppose that P(B) is true and that s is an hereditarily finite set given explicitly. Calculate all the conditions C_j (s) one after another, until one evaluating to 'true' is found. (If none such is found, stop the computation; F(s,B) simply evaluates to itself). If some first C_j (s) evaluates to 'true', calculate the corresponding e_j (s,B) recursively. This is done by going through the syntax tree of e_j(s,B) in bottom-to-top order, evaluating all elementary subexpressions of the form exp(s) directly, expanding each recursive occurrence of F having the form F(exp(s),B), where exp(s) is elementary, recursively, and leaving all other subexpressions untouched. If this process fails to terminate it can simply be stopped after a while, but if it terminates it will yield an identity F(s,B) = expn(s,B). Deduction by semi-symbolic computation makes this identity available as a theorem, in the form

P(B) & s in HF *imp F(s,B) = expn(s,B).

A typical, relatively elaborate application of this general form of deduction by semi-symbolic computation makes it possible to obtain 'Mathematica'-like conclusions by syntactic means in a very direct way. Many of the 'formula-driven' parts of elementary and intermediate-level mathematics are covered by this technique.

Derivative manipulations in calculus furnish a characteristic example. Suppose, e.g., that one has defined the derivative 'Deriv' as a map from smooth functions of a real variable to their derivative functions, and that the specific real functions 'sin', 'cos', 'exp', along with the basic rules for differentiation and the derivatives of these specific functions, have also been defined. This basic information can then be built up in the following way into general symbolic-manipulation mechanisms allowing direct derivation of composite relationships like

Deriv({x.cos(cos(x)): x in R}) = {x.sin(cos(x)) _rprod sin(x): x in R},

where R designates the set of real numbers. For this, we first define an appropriate class of (hereditarily finite) syntax trees as follows:

wf(t) *eq is_tuple(t) & #t = 3 & t[0] in {"+","*","-"} & wf(t[1]) & wf(t[2]) or is_tuple(t) & #t = 2 & t[0] in {"sin","cos","exp"} & wf(t[1]) or is_string(t) & t = "x" or is_integer(t)

Next we write a definition for the intended semantic meaning of a well-formed tree, which for the example at hand is an elementary function of reals to reals:

```
(*)  tree_value(t) :=    if is_tuple(t) & #t = 3 & t[0] = "+" then         tree_value(t[1]) +' tree_value(
```

Here the predicate is_tuple(t) states that t is a finite sequence (i.e. a mapping whose domain is an integer). The operator +' designates the pointwise sum of two functions, namely

f +' g = {[x,f~[x] + g~[x]]: x in domain(f) sect domain(g)},

and similarly for *' and -'; note that "+" symbolizes real rather than integer summation here, and similarly for "*" and "-". Also f@g will designate the composition of the two functions f and g. 'float' is the function which embeds the integers into the reals.

The next step is to define the operation on trees which builds their formal derivatives. This is

```
formal_deriv(t) =     if wf(t) & #t = 3 & t[0] = "+" then         ["+",formal_deriv(t[1]),formal_deriv(t[
```

Given these definitions, it is not hard to prove the following recursive relationship.

(t in HF & wf(t)) *imp Deriv(tree_value(t)) = tree_value(formal_deriv(t))

In sketch, the proof is as follows: suppose not, i.e. suppose that there exists a t such that

```
t in HF & wf(t) & Deriv(tree_value(t)) /=     tree_value(formal_deriv(t)) & wf(t[1]) & wf(t[2]).
```

Choose a smallest such t, in the ordering defined by the relationship *inin. For this, we must have

```
(**)  (j in  Z & 0 < j & j < #t) *imp     (Deriv(tree_value(t[j])) = tree_value(formal_deriv(t[j]))).
```

From this, we can derive a series of conclusions which collectively contradict our supposition. For example, if

```
wf(t) & #t = 3 & t[0] = "+" & wf(t[1]) & wf(t[2]),
```

then we have

```
Deriv(tree_value(t[1])) = tree_value(formal_deriv(t[1])) &     Deriv(tree_value(t[2])) = tree_value(forma
```

since t[1] *inin t & t[2] *inin t. Thus if

```
wf(t) & #t = 3 & t[0] = "+" & wf(t[1]) & wf(t[2]),
```

so that tree_value(t) = tree_value(t[1]) +' tree_value(t[2]) by (*), we must also have

```
Deriv(tree_value(t)) = Deriv(tree_value(t[1])) +'     Deriv(tree_value(t[2]))
```

by the standard theorem on the derivative of the sum of two real functions, which we assume to have been proved separately, along with the corresponding elementary results for products, quotients, sin and cos, exp, etc. Hence in this case the conjunct (**) seen above cannot hold.

To apply this in the most convenient manner, we will sometimes require one more inductive relationship for use as a computational rule, namely

```
tree_value(t)[x] =      if wf(t) & #t = 3 & t[0] = "+" then         tree_value(t[1])[x] + tree_value(t[2]
```

We also need to show that

```
t in HF & wf(t) *imp single_valued(tree_value(t)),
```

which follows readily by an induction like that sketched above.

Putting all this together, it follows that we can:

(i) Deduce a general theorem, like that outlined above, which relates the syntax trees of a class of formulae of interest to the semantic (set theoretic) values of these trees;

(ii) Supply the well-formed tree t of the formula we want;

(iii) Then Deriv(tree_value(t)) and tree_value(formal_deriv(t)) can be evaluated automatically, and the identity Deriv(tree_value(t)) = tree_value(formal_deriv(t)) can be made available as a theorem directly.

This allows derivative calculations like

```
Deriv({[x,cos(cos(x))]: x in R}) =     {[x,sin(cos(x)) _rprod sin(x)]: x in R}
```

to become theorems without further proof, if we simply supply an appropriate syntax tree to a deduction by semi-symbolic computation.

Since the tree t required for this little procedure is available directly from the formula of interest (e.g. cos(cos(x))), we can even package very useful theorem- generators of this form as Mathematica-like computational tools, i.e. introduce auxiliary system commands having forms like

```
DIFFERENTIATE: cos(cos(x));
```

This command can simply parse its input formula to obtain the tree of interest, generate the additional boilerplate seen in

```
Deriv({x.cos(cos(x)): x in R}) =     {x.sin(cos(x)) _rprod sin(x): x in R},
```

and make this available as a theorem.

Simple system-extension tools for doing just this are described below.

In addition to the specific use just sketched, deduction by semi-symbolic computation is applicable in a wide variety of other circumstances. These include:

- computations with partial derivatives;
- symbolic integration and differentiation;
- manipulation of series and of combinatorial coefficients;
- other Mathematica-like symbolic computations;
- elementary arguments concerning continuity and smoothness;
- elementary reasoning concerning object types;
- polynomial computations in one and several variables;

- use of trigonometric identities;
- matrix computations and linear algebra;
- some asymptotic estimates;
- some numerical computation, e.g. approximate evaluation of integrals;
- Boolean computations;
- computations with finite groups, sets of permutations, and computations in modular arithmetic.

Deduction by semi-symbolic computation is also available for Boolean equivalences of the form

$$P(B) \;\&\; HF(s) \;*imp\; Q(s,B) \;*eq\; \text{if } C_1(s) \text{ then } e_1(s,B) \text{ elseif } C_2(s) \text{ then } e_2(s,B) \ldots \text{ elseif } C_n(s) \text{ then } e_n(s,B) \text{ else } F(s,B) \text{ end},$$

and yields implications of the form

P(B) & s in HF *imp Q(s,B) *eq expn(s,B).

Deduction by semi-symbolic computation can also be given nondeterministic and/or interactive form. To make it nondeterministic, we can supply a set of identities

$$P(B) \;\&\; HF(s) \;*imp\; F(s,B) = \text{if } C_1\,j(s) \text{ then } e_1\,j(s,B) \text{ elseif } C_2\,j(s) \text{ then } e_2\,j(s,B) \ldots \text{ elseif } C_n\,j(s) \text{ then } e_n\,j(s,B) \text{ else } F(s,B) \text{ end},$$

j = 1..k, rather than a single such identity. In the presence of such a set SI of initial identities and of the assumption P(B) we can supply a target identity, and then explore the set of substitutions generated by SI nondeterministically in all possible patterns, until either the target identity is generated or all possible substitutions have been examined.

3.11. Details of verifier command syntax

3.12. A closer examination of the sequence of definitions and theorems presented in this book

As already said, this text will culminate in the sequence of definitions and theorems found in Chapters XX-YY. The present section prepares for these chapters by expanding the broad survey of these definitions and theorems presented ibelow to a more detailed examination of individual definitions and theorems, but for the moment still without proofs. These proofs are given in Chapters XX-YY.

Our first step is to give the following definition of the notion of ordered pair. Its details are unimportant; all that matters is that the first and second components of an ordered pair can be reconstructed uniquely from the pair itself. The five theorems which follow Definition 1 assure us that this is the case, and give explicit (but subsequently irrelevant) formulae for extracting the first and second components of an ordered pair.

```
Def 1: Ordered pair: [x,y] := {{x},{{x},{{y},y}}}

Theorem 1: arb({X}) = X   Theorem 1a: arb({{X},X}) = X   Theorem 2: arb([X,Y]) = {X}   Theorem 3: arb(arb
```

The two following definitions simply capture the two formulae which extract the first and second components of an ordered pair.

```
Def 2: car(x) := arb(arb(x))  Def 3: cdr(x) := arb(arb(arb(x - {arb(x)}) - {arb(x)}))
```

All our subsequent work with ordered pairs uses only the properties stated in Theorems 5, 6, and 7, which now follow immediately. These are the properties which are built into our verifier's ELEM deduction mechanism.

```
Theorem 5: car([X,Y]) = X   Theorem 6: cdr([X,Y]) = Y   Theorem 7: [X,Y] = [car([X,Y]),cdr([X,Y])]
```

Next we give a few small theories which make elementary properties of setformers available in a convenient form. These are

```
THEORY setformer(e,ep1,s,p,pp1)            (FORALL x in s | e(x) = ep1(x))         & (FORALL x in s | p(x
```

The first of the above theories simply allows equals-by-equals replacement in setformers involving single and double iterations respectively. The second assures us that setformers involving non-empty iterations must define non-empty sets. All the required proofs involve tedious elementary detail which the availability of these theories allows us to elide subsequently.

We go on to define the basic notions of mapping (a mapping is simply a set all of whose elements are ordered pairs), the domain and range of a mapping, and the notions of single-valued and one-to-one mappings. This done by the five following definitions.

```
Def 4: is_map(f) :*eq f = {[car(x),cdr(x)]: x in f}  Def 5: domain(f) := {car(x): x in f}  Def 6: range(f
```

Next we state, and subsequently prove, a general principle of transfinite induction. For ease of use, this is captured as a theory called 'transfinite_induction'. It states that, given any predicate P set which is true for some set, there must exist a set s for which P is true, but for all whose members P is false. This principle, which follows very directly from our strong form of the axiom of choice, is encapsulated in the following Theory.

```
THEORY transfinite_induction(n,P)    P(n)  ==>(m)    P(m) & (FORALL k in m | not P(k))  END transfinite_i
```

Our next strategic aim is to define the notion of 'ordinal number', and to prove the basic properties of ordinals. We follow von Neumann in defining an ordinal as a set properly ordered by membership, and for which members of members are also members. This ties the ordinal concept very directly to the most basic concepts of set theory, allowing the properties of ordinals to be established by using only elementary properties of sets and set formers, with occasional use of transfinite induction. The key results proved are: (a) the collection of all ordinals is itself properly ordered by membership, and members of ordinals are ordinals, but (b) this collection is not a set. (c) any set can be put into 1-1 correspondence with an ordinal.

The formal statement of the property 's in an ordinal' is as follows.

```
Def 10: Ord(s) :*eq (FORALL x in s | x *incin s)        & (FORALL x in s | (FORALL y in s |             (
```

Since we have defined ordinals in a directly set-theoretic way, the notion of 'successor ordinal' (the next ordinal after a given ordinal) also has an elementary set-theoretic definiton: the set obtained from s by adding s itself as a (necessarily new) member. Formally, this is:

```
Def 11: next(s) := s + {s}
```

Next we prove the basic properties of ordinals. Theorem 8, which serves as an auxiliary lemma, states that each proper sub-ordinal T of an ordinal S is the smallest element of the complement (S - T), We then prove that the intersection of any two ordinals is an ordinal, and that, given any two ordinals, one is a subset of the other (so that their intersection is simply the smaller of the two and their union is the larger). Somewhat more precisely, given any two distinct ordinals one is a member of the other (which tells us that comparison between ordinals can be expressed either by inclusion or by membership). Every element of an ordinal is also an ordinal, and if s is an ordinal then next(s) is a larger, and indeed the

next larger, ordinal.

The class of sets cannot be a set (i.e. there can be no set of which all sets are members, since if there were this would have to be a member of itself). Similarly, there can be no ordinal of which all ordinals are members (since if there were, the union of all elements of this set would have to be the largest ordinal, and hence would be a member of itself). These two facts, which tell us that 'all sets' and 'all ordinals' are both too large to be sets, are Theorems 13 and 14 in the following group.

```
Theorem 8: (Ord(S) & Ord(T) & T *incin S) *imp         (T = S or T = arb(S - T))   Theorem 9: (Ord(S) & O
```

Next we prove that every set s can be put into one-to-one correspondence with an ordinal. This is done by defining a correspondence between ordinals and elements of s recursively: the element corresponding to any ordinal o is the first element, if any, not corresponding to any smaller ordinal. Since we have already proved that the collection of all ordinals is too large to be a set, this enumeration must ultimately cover the whole of s. This is the 'enumeration theorem' fundamental to our subsequent work with cardinal numbers.

The following definition formalizes the enumeration technique just described.

```
Def 9: [The enumeration of a set]     enum(X,S) := if S *incin {enum(y,S): y in X} then S         else ar
```

The following six theorems do the work necessary to prove the Enumeration theorem, which is the last of them. Theorem 17 is a lemma for Theorems 18, which states that enum(X,S) is always either a member of S, or, past a certain point, the whole of S. Theorem 20 states that if an ordinal is large enough to enumerate S, so is every larger ordinal.Theorem 20 states that enum(X,S) defines a 1-1 correspondence up to the point at which the whole of S has been enumerated, and Theorem 21 states that the whole of S must eventually be enumerated (since otherwise we would have a 1-1 correspondence of all ordinals with a subset of S, contradicting the fact that there are too many ordinals to constitute a set).

```
Theorem 17: (Ord(X) & S in {enum(y,S): y in X}) *imp     (S *incin {enum(y,S): y in X})  Theorem 18: enum
```

Our next goal is to define the notion of the cardinality of a set s, i.e. the number, finite or infinite, of its elements. As appears in definition 15 below, this is simply the smallest ordinal which can be put into 1-1 correspondence with s. But in preparation for this definition we first define a few more elementary set-theoretic notions and prove a few more elementary properties of maps. The notions defined are: the restriction of a map to a set, the inverse map of a map, the identity map on a set, and map composition. The image of a point x under a map is defined as the unique element (or, if not unique, the element chosen by 'arb') of the range of the restriction of the map to the singleton {x}. The following block of definitions formalize these ideas.

```
Def 12: [Map Restriction]     def(f *ON  a) := {p in f | car(p) in a}  Def 13: [Value of single-valued fu
```

A collection of elementary theorems expressing familiar set-theoretic facts is proved next: the restriction of a map to a set is a submap of the original map; a set is a map if and only if all its elements are ordered pairs; a subset of a map is a map. A subset of a single-valued map is a map. a subset of a one-to-one map is a one-to-one map. Theorems 24 and 25 just express the intersection and difference of two sets as setformers.

```
Theorem 23: F *ON  A *incin F  Theorem 24: S * T = {x in S | x in T}  Theorem 25: S - T = {x in S | x not
```

Continuing this series of elementary set-theoretic propositions, we have the following results: The first and second components of any element of a map belong to the map's domain and range respectively. The union of two maps is a map. The restriction of a map to a union set is the union of the separate restrictions. The restriction of the union of two maps to a set is the union of their separate restrictions. A map is its restriction to its own domain. Map products are associative.

```
Theorem 30: (X in F) *imp (car(X) in domain(F))   Theorem 31: (X in F) *imp (cdr(X) in range(F))   Theore
```

Three additional theorems in this elementary group state that (i) the image under a map of any element of its domain belongs to its range; (ii) A single-valued map can be written as the set of all pairs built from images of its domain elements; (iii) The range of a single-valued map is the collection of all image elements of its domain.

```
Theorem 38: (X in domain(F)) *imp (F~[X] in range(F))  Theorem 39: Svm(F) *eq F = {[x,F~[x]]: x in domain
```

It is convenient to repackage the elementary results just stated as a theory which puts every one-parameter function symbol f onto correspondence with a single-valued map g which sends each element x of the map's domain into f)x) as image element. The theory shown below does this, and also expresses the range of g and the condition that g should be one-to-one in terms of f.

```
THEORY fcn_symbol(f,g,s)          g = {[x,f(x)]: x in s}  ==>      domain(g) = s      (FORALL x in s | g~
```

In working with maps we often need to use elementary properties of ordered pairs. The following theorems are two such: Any ordered pair can be written in standard fashion in terms of its formal first and second component. Any element of a map is an ordered pair. The small utility theory which follows states that every setformer involving only ordered pairs defines a map.

```
Theorem 40: (U = [A,B]) *imp (U = [car(U),cdr(U)])   Theorem 41: (is_map(F) & U in F) *imp (U = [car(U),c

THEORY iz_map(f,a,b,s)       f = {[a(x),b(x)]: x in s}  ==>      is_map(f)  END iz_map;
```

More elementary utility results on maps and their ranges and domains now follow. The domain and range operators are both additive, and if one is null so is the other. A single-valued map sends the first component of any pair in it to the second component of the same pair. The union of two single-valued maps with disjoint domains is a single-valued map. The union of two one-to-one maps with disjoint domains and ranges is a one-to-one map. Any restriction of a map is a map; any restriction of a single-valued map is a single-valued map; any restriction of a one-to-one map is a one-to-one map. The range of any restriction of a map is a subset of the map's range, and the domain of a map's restriction to a set s is the intersection of s and the map's domain. f the range of a map f is included in the domain of a map g, the domain of the composite map f @ g is the domain of g, and its range is the range of the restriction of f to the range of g. Hence if the range of f equals the domain of g, the range of the composite map equals the range of f.

```
Theorem 42: domain(F + G) = domain(F) + domain(G)  Theorem 43: range(F + G) = range(F) + range(G)  Theore
```

Next we have a block of elementary results on map inverses. The inverse of a map f is a map, whose domain is the range of f and vice-versa. The inverse of the inverse of a map is the map itself. If a map is one-to-one, so is its inverse. The inverse f a map f sends the image under f of each element x of the domain of f into x, and vice-versa. The composite of a map and its inverse sends every element x of the map's domain into x, and symmetrically the composite of the inverse of f and f sends each element y of the range of f into y.

```
Theorem 53: is_map(inv(F)) &     range(inv(F)) = domain(F) & domain(inv(F)) = range(F)   Theorem 54: is_m
```

Next we give a few elementary results elementary results on identity maps, i.e. maps which send every element of some set s into itself. Every such map is one-to-one, inverse to itself, and has s as its range and domain. The composite of any single-valued map f with its inverse is the identity map on the range of f, and, if f is one-to-one, the composite in the reverse order is the identity map on the domain of f.

```
Theorem 58: [Elementary Properties of identity maps]  one_1_map(ident(S)) & domain(ident(S)) = S &
```

The final theorems in our collection of elementary focus on composite maps. If two maps are one-to-one and inverse to each other, their composite is the identity map on the domain of one of them, and the composite in the opposite order is the identity map on the corresponding range. The composite of two maps is a map, the composite of two single-valued maps is a single-valued map, and the composite of two one-to-one maps is a one-to-one map. If f and g are two single-valued maps with the range of g included in the domain of f, then their composite sends each x in the domain of f into the g-image of the f-image of x, and both the composite map and its range can be written as setformer expressions. Map composition is distributive over map union.

```
Theorem 61: [An inverse pair of maps     must be 1-1 and must be each others inverses]      (is_map(F) &
```

Now we are ready to go on to the theory of cardinal nmubers, in preparation for which we Skolemize the theorem which states that any set has a standard enumeration, to get the following definition, which gives a name to the ordinal which enumerates each set in the standard way.

```
Def 14c: Ord(enum_Ord(s)) & s = {enum(y,s): y in enum_Ord(s)} &         (FORALL y in enum_Ord(s) | (FORALL
```

We can now define the cardinal number of a set s to be the least ordinal with which it can be put into one-to-one correspondence. Accordingly an ordinal is a cardinal if it can not be put into one-to-one correspondence with any smaller ordinal. The two following definitions capture these ideas.

```
Def 15: [Cardinality]     def(#s) := arb({x: x in next(enum_Ord(s)) | (EXISTS f in OM |       (one_1_map(
```

In working with cardinals (and in particular with products of cardinals) we will need various elementary facts about Cartesian products. The Cartesian product of two sets s and t is simply the set of all pairs whose first component belongs to s and whose second component belongs to t. Formally this is

```
Def 17: [Cartesian Product]     def(s *PROD t) := {[x,y]: x in s, y in t}
```

The two following theorems state associativity and commutativity properties of the Cartesian product: ((A *PROD B) *PROD C) and (A *PROD (B *PROD C))) are always in natural one-to-one correspondence, as are (A *PROD B) and (B *PROD A). These facts will subsequently imply the associativity and commutativity of cardinal multiplication.

```
Theorem 68:     (F = {[[[x,y],z],[x,[y,z]]]: x in A, y in B, z in C}) *imp      (one_1_map(F) & domain(F)
```

Now we go on to the study of cardinals, beginning with a few relevant facts about ordinals, stated in the next block of theorems. Theorem 70 states that the standard enumeration function enum(x,s) = $enum_s(x)$ defined earlier is the identity (in x) if s is an ordinal. Theorem 71 states that every set can be put into one-to-one correspondence with a certain smallest ordinal. Then we show that the enumerating ordinal of a set has the same cardinality as the set, and that if a set s of ordinals includes a set t, then arb(s) is smaller than arb(t). These are both lemmas needed later. Theorem 74 states a related lemma, needed later to prove that the cardinal number of a set s is at least as large as the cardinal number of any of its subsets:

```
Theorem 70: (Ord(S) & X in S) *imp (enum(X,S) = X)  Theorem 71: [Cardinality Lemma] Ord(#S)     & (EXISTS
```

The block of theorems which now follow encapsulate the basic properties of cardinal numbers. We define the notion of 'cardinality' and the related operator #s which gives the (possibly infinite) number of members of a set. The cardinality of a set is defined as the smallest ordinal which can be put into 1-1 correspondence with s, and it is proved that (a) there is only one such cardinal, and (b) this is also the smallest ordinal which can be mapped onto s by a single-valued map. We also define the notions of cardinal sum and product of two sets a and b. These are respectively defined as #(copy_a + copy_b), where copy_a and copy_b are disjoint copies of a and b, and as the cardinality of the Cartesian product a *PROD b of a and b. Using these definitions, it is easy to prove the associative and distributive laws of cardinal arithmetic. We also prove a few basic properties of the #s operator, e.g. its monotonicity.

```
Theorem 78: [Subsets of an ordinal have a cardinality        that is no larger than the ordinal]     (Or
```

All the preceding results are as true for infinite sets, ordinals, and cardinals as for finite objects. We now go on to introduce the important notion of finiteness and to prove its basic properties. The definition is as follows: a set s is finite if it cannnot be mapped into any proper subset of itself by a one-to-one mapping.

```
Def 18: [Finiteness]      Finite(s) :*eq not(EXISTS f in OM | (one_1_map(f) &         domain(f) = s & ran
```

An equivalent property is that s should not be the single-valued image of any proper subset of itself. To begin work with the basic notion of finiteness, we prove that the null set is finite, that any subset of a finite set is finite, and that a set is finite if and only if its cardinality (with which it is in one-to-one correspondence) is finite. It is also proved (Theorems 102, 103, and 104) that two sets in one-to-one correspondence are both finite if either is, and that the image of a finite set under a single-valued map is always finite.

Along the way we prove a utility collection of results on the finiteness and cardinality of maps, and of their ranges and domains. These are as follows. Both the range and domain of a mapping have a cardinality no larger than that of the map itself. If a map is single-valued, it has the same cardinality as its domain. If t is a non-null subset of s, then there exists a single-valued mapping whose domain is s and whose range is t; this map can be one-to-one if and only if s and t have the same cardinality. A set s is a cardinal if and only in it is its own cardinality, i.e. s = #s. The cardinality operator '#' is idempotent, and the membership operation for cardinals has the transitivity properties of a comparison operator. A single-valued map is one-to-one if and only if each domain element of the map is defined uniquely by the corresponding range element.

Theorem 101 is a utility lemma collecting various elementary properties of product maps, some of them proved previously.

We also prove the basic lemmas (Theorems 96 and 97) that we will use to show that cardinal multiplication is associative and commutative once this multiplication operation has been defined, and the lemma (Theorem 100) needed to show that the power operation 2^C is well-defined for cardinal numbers C.

```
Theorem 86: [0 is a finite cardinal]     Ord(0) & Finite(0) & Card(0)   Theorem 87: #domain(F) *incin #F
```

Our next block of theorems works further into the properties of finite sets. We show that any proper subset of a finite set s has a smaller cardinality than s (this condition is equivalent to finiteness), that any member of a finite ordinal is finite (so that any infinite ordinal is larger than any finite ordinal), that the addition of a singleton to a finite set gives a finite set. This implies that any singleton is finite, and that the successor set next(s) of a finite set is always finite. We prove the equivalence of a second possible definition of finiteness: s is finite if and only if it cannot be the single-valued image of any of its proper subsets.

Theorem 110 is a simple utility lemma asserting that any two elements of a set can be interchanged by a one-to-one mapping of the set into itself. Theorem 111 collects various elementary properties of single-valued maps, their domains, and their restrictions. Theorem 112 is an elementary converse of the utility Theorem 101.

```
Theorem 106:     [Proper subsets of a finite set have fewer elements]     (Finite(S) & T *incin S & T /=
```

Our next main goal is to prove that the collection of all finite ordinals is a set (this set, which is also the set of all finite cardinals, is of course the set of integers, and hence the foundation stone of all traditional mathematics). This is done by using the infinite set s_inf whose existence is assumed in the axiom of infinity to prove that there exists an infinite ordinal. The set Z of integers (from the German: 'Zahlen') can then be defined as the least infinite ordinal, which we show is also a cardinal. We also show that a cardinal is finite if and only if it is a member of Z, and define the standard integers 1,2,3, etc. as next(0), next(1), next(2), etc., and prove that these are all distinct.

```
Theorem 115: not(Finite(s_inf))   Theorem 116: [Infinite cardinality theorem]         not(Finite(#s_inf))
```

Def 18a: [The set of integers] Z := arb({x in next(#s_inf) | not Finite(x)}) Theorem 118: Ord(Z) & (not Finite(Z)) & (FORALL x in OM | ((Card(x) & Finite(x)) *eq x in Z))

Def 18b: [Standard definitions of the finite integers] 1 = next(0) & 2 = next(1) & 3 = next(2) & ... Theorem 119: Ord(0) & 0 in Z & 1 in Z & 2 in Z & 3 in Z Theorem 120: [The set of integers is a Cardinal] Card(Z) Theorem 121: 0 in Z & 1 in Z & 2 in Z & 3 in Z & 1 /= 0 & 2 /= 0 & 3 /= 0 & 1 /= 2 & 1 /= 3 & 2 /= 3

Our next block of theorems continues to develop the basic principles of arithmetic, and hence brings us into standard mathematics. The notions of addition, multiplication, (unsigned) subtraction, division, and remainder after division are first defined using simple set theoretic constructions. (The sum of two cardinals n and m is the cardinality of the union of any two disjoint sets in 1-1 correspondence with n and m respectively; the product of n and m is the cardinality of the Cartesian product of the sets m and n; their difference is the cardinality of the difference set m - n). The quotient of m over n is the largest k whose product with n is included in m, and the remainder is m - (m/n) as usual. All this is formalized in the six following definitions, which also include the definition of the notion of powerset.

```
Def 19: [Cardinal sum]       def(n *PLUS m) = #({[x,0]: x in n} + {[x,1]: x in m})  Def 20: [Cardinal pro
```

Next a few necessary lemmas are proved. The sets which appear in the definition of cardinal summation are disjoint and have the same cardinality as the sets from which they are formed; the null set is a one-to-one map with null range and domain; and a single ordered pair defines a one-to-one map. We also prove a simple utility formula for maps constructed out of just two ordered pairs.

```
Theorem 122: {[x,0]: x in N} * {[x,1]: x in M} = 0  Theorem 123: is_map(0) & Svm(0) & one_1_map(0) &
```

In preparation for a closer examination of the rules of cardinal arithmetic, we prove next that the cardinal sum and product of two sets can be calculated either from the sets or from their cardinal numbers. We so show that any proper subset of a finite set has a smaller cardinal number.

```
Theorem 127: N *PLUS M = #N *PLUS #M  Theorem 128: N *PLUS M = N *PLUS #M  Theorem 129: N *TIMES M = #N *
```

Since the following discussion will occasionally use inductive arguments which refer to the subsets of a finite set, it is convenient to make these available in a theory. This states that, given any predicate P(x) which is true for some finite set, there exists a finite set s for which P(s) is true, but P(s') is false for all proper subsets of s.

```
THEORY finite_induction(n,P)    Finite(n) & P(n)  ==>(m)    m *incin n & P(m) &         (FORALL k *incin
```

Now we are ready to prove the main elementary properties of integer arithmetic. We show that the union of two finite sets is finite, and that a cardinal sum of two sets is finite if and only the union of the two sets (i.e. both of the two sets) is finite. The statements '0 times any n equals zero', and 'one times any n equals n' are proved in several convenient equivalent forms. We show that n * m is at least as large as m if n is not zero, and show how to express the cardinal sum as the cardinality of two distinct Cartesian product sets, whose disjointness is then demonstrated. Then the distributive and commutative laws for cardinal (and hence integer) arithmetic are established by relating them to corresponding set-theoretic constructions. Finally we show that the Cartesian product of two finite sets is finite, and that the converse holds as long as neither of the sets is empty.

```
Theorem 132: Finite(N) & Finite(M) *eq Finite(N + M)   Theorem 133: Finite(N *PLUS M) *eq Finite(N + M)
```

Next a few well-known results concerning power sets and their cardinalities are proved. The power set of the null set is the singleton {0}. The power set of a set s is finite if and only if s is finite, but (Cantor's theorem, the historical root of the whole theory of infinite cardinals) always has a larger cardinality than s.

```
Theorem 154: pow(0) = {0}    Theorem 155: Finite(N) *eq Finite(pow(N))    Theorem 156: [Cantor's Theore
```

Next we prove some properties of cardinal subtraction, along with some auxiliary properties of the cardinal sum: n - n is always 0, n - 0 is n. (n - m) + m and m + (n - m) are both n if m is no larger than n. The cardinality of the union set s + t is the cardinal sum of s and t if the two sets are disjoint, and this value depends only on the cardinalities of the sets involved.

```
Theorem 157: N *MINUS N = 0  Theorem 158: N *MINUS 0 = #N   Theorem 159: [Disjoint sum Lemma]     (N * M
```

Because of the set-theoretic way in which we have defined ordinals, the maximum of a set s of ordinals is simply the union of all the ordinals. This fact is captured in our next block of theorems, which begins with the very simple definition of the concet 'union set'.

```
Def 25: [Union Set]: Un(S) := {x: x in y, y in S}
```

Our next two theorems capture the fact stated just above: the union of a set s of ordinals is always an ordinal, and is the least upper bound of s.

```
Theorem 163: [Union set as an upper bound]      (FORALL x in S | x *incin Un(S)) &         ((FORALL x in
```

Now we prove two basic elementary properties of division: n/m is no larger than n unless m is 0, and is an integer if n and m are both integers. We also show that the sum, product, and difference of integers is an integer.

```
Theorem 165: (M /= 0) *imp (N *OVER M *incin N)  Theorem 166: (M /= 0 & N in Z) *imp     (N *OVER M in Z
```

Next several results on the monotonicity of addition, multiplication, and subtraction are given. Once we have extended the notion of 'integer' to that of 'signed integer' these will become the standard monotonicity properties for algebraic combinations of signed integers, and ultimately of rational numbers and of reals. We show that: integer addition is strictly monotone in both of its arguments (several variants of this result are given); integer multiplication is monotone in both of its arguments (but not strictly, unless 0 is excluded as a factor). We also prove that subtraction is strictly monotone in its arguments, and establish the cancellation rule for unsigned addition needed later to justify the definition of signed addition and its relationship to signed subtraction.

```
Theorem 169: [Strict monotonicity of addition]      (M in Z & N in Z & N /= 0) *imp (M in M *PLUS N)  The
```

Our next, rather miscellaneous block of theorems show that subtraction stands in the correct relationship to addition, and prove some related facts on the monotonicity of addition. We also show that the cardinality of any singleton is 1, that only the empty set has cardinality zero, and that if a cardinal product is zero one of its two factors must be zero. This last statement is subsequently used in constructing rational numbers.

```
Theorem 176: (M in Z & N in Z & K in Z &     N incs M & N *MINUS M incs K) *imp       (N incs M *PLUS K &
```

Although our main goal is now to move on to the principal notions and theorems of analysis, we digress to prove that the sum and product of any two infinite cardinals degenerates to their maximum. The two following theories prepare for this. Given any ordinal-valued function on a set s, the first theory constructs the subset 'rng' of s on which f assumes its minimum. The second theory tells us that any well-ordering of a set s defines a one-to-one mapping of some ordinal o onto s which realizes an isomorphism of the natural ordering of o (by the 'in' relator) to the given ordering of s. It also asserts that the mapping sends all ordinals larger than o onto s, and all smaller ordinals onto an initial slice of s, i.e. all elements of s up to some given v in s.

```
THEORY ordval_fcn(s,f);      [Elementary functions of ordinal-valued functions]      s /= 0 & (FORALL x i
```

```
THEORY well_ordered_set(s,Ordrel);    (FORALL x in s | (FORALL y in s |       ((Ordrel(x,y) or Ordrel(y,x
```

The next seven theorems lead up to the Cardinal Square theorem which is the main result of our digression. The main theorems are 194 and 195, which state that the cardinal product of any infinite cardinal n with itself, or with any smaller nonzero cardinal, is simply n, and 192, which states that the sum of two infinite cardinals is simply the larger of the two. The remaining theorems in the block displayed are preparatory. Theorem 189 states that addition of a single new element to an infinite set does not change its cardinality. Theorems 190 and 191 tell us that any infinite set s can be divided into two parts, both of the same cardinality as s. Theorem 193 tells us that any infinite set is in 1-1 correspondence with the Cartesian product of some other set t with itself.

```
Theorem 189: [One-more Lemma]     (not Finite(S)) *imp (#S = #(S + {C}))  Theorem 190: [Division-by-2 Lem
```

Returning to our main line of development, we now introduce the set of signed integers as the set of pairs [x,0] (representing the positive integers) and [0,x] (representing the integers of negative sign). The formal definition is as follows.

```
Def 26: [Signed Integers]     Si := {[x,y]: x in Z, y in Z | x = 0 or y = 0}
```

Any pair of integers can be reduced to a signed integer by subtracting the smaller of its two components from the larger. This operation, introduced by the following definition, appears repeatedly in our subsequent proofs of the properties of signed integers.

```
Def 27: [Signed Integer Reduction to Normal Form]     Red(P) := [car(P) *MINUS (car(P) * cdr(P)),
```

Next we extend the notions of sum, product, and difference to signed integers, and also define three elementary operators, the absolute value, negative, and sign of a signed integer, that have no direct analog for unsigned integers. The sum of two signed integers i and j is simply the reduction of their componentwise sum. The absolute value of i is the maximum of its two components (only one of which will be nonzero). The negative of i is simply i with its components reversed. The difference of two signed integers is then simply the sum of the first with the negative of the second. The product of signed integers is defined as the reduction of an algebraic combination of their components, formed in away that reflects the standard 'law of signs'. A signed integer is positive if its second component is null; otherwise it is negative.

[0,0] is the 'signed integer' 0, and the 1-1 mapping x --> [x,0], whose inverse is simply y --> car(y), embeds Z into the set of signed integers, in a manner allowing easy extension of the addition, subtraction, multiplication, and division operators to signed integers.

The relevant formal definitions are as follows.

```
Def 28: [Signed Sum] def(MM *S_PLUS NN) :=       Red([car(MM) *PLUS car(NN),cdr(MM) *PLUS cdr(NN)])  Def
```

The sequence of about 30 theorems which follows establishes all the main properties of signed integers, deriving these from the properties of unsigned integers established previously. The proofs involved are all elementary, though sometimes a bit tedious. Theorem 196 is a lemma asserting that the reduction of any pair of unsigned integers is a signed integer, and that the set of unsigned integers is closed under multiplication. Theorem 196 is a second lemma which merely restates the defining properties of signed integers. Theorem 197 restates the way in which signed integeres are defined using integers. Theorem 199 begins our main work, by showing that the set of signed integers is closed under addition and multiplication.

```
Theorem 196: (M in Z & N in Z) *imp     (Red([M,N]) in Si & M * N in Z)  Theorem 197: (N in Si) *imp (N =
```

To move toward our goal of establishing all the basic elementary properties of signed integers, we first prove some auxiliary properties of the reduction mapping 'Red' which normalizes pairs of integers by subtraction, sending them into equivalent signed integers. We show that Red([n,m]) remains invariant if a common integer is added to n and m, that Red([n,n]) is always the signed zero element [0,0], and that the signed addition and multiplication operations remain invariant if one of their arguments [n,m] is replaced by Red([n,m]). The proofs are all elementary, but many involve

examination of multiple cases.

```
Theorem 200: (N in Z) *imp (Red([N,N]) = [0,0])  Theorem 201: (J in Z & K in Z & M in Z) *imp     (Red([J
```

Moving on toward proof of the basic properties of signed integers, we first prove commutativity of signed integer addition via two preliminary lemmas which give commutativity for corresponding sums of ordered pairs of integers, and then commutativity, associativity, and distibutivity of signed integer multiplication. Next, after a lemma which states that the reduction of a signed integer is the signed integer itself, we show that the mapping of n into [n,0] sends integers into signed integers in a manner which makes unsigned addition, multiplication, and subtraction correspond to signed addition, multiplication, and subtraction respectively.

```
Theorem 205: [Commutativity Lemma]     (K in Si & N in Z & M in Z) *imp         (K *S_PLUS [N,M] = [N,M]
```

Next we give a few elementary theorems on the operation of sign reversal for signed integers: the law of signs for multiplication, the rule that -(-n) is n, and the fact that n + (-n) is 0.

```
Theorem 214: (N in Z & M in Z) *imp     (S_Rev(Red([M,N])) = Red([N,M]))  Theorem 215: (N in Si & M in Si
```

Our next four theorems lead up to the proof that signed integer multiplication is associative. The first three results state this in special cases. This stepwise approach is needed since a large number of cases need to be examined.

```
Theorem 219: [Associativity Lemma]     (K in Z & N in Z & M in Z) *imp         ([N,0] *S_TIMES ([M,0] *S_
```

The final block of theorems in this 'signed integer' group show that n + (-m) is n - m, that -(n + m) is -n - m, that [1,0] is the multiplicative identity for signed integer multiplication, and that [0,0] is the additive identity. All the proofs are elementary.

```
Theorem 223: (N in Si & M in Si) *imp     (N *S_MINUS M = N *S_PLUS S_Rev(M))  Theorem 224: (N in Si & M
```

Next, in direct preparation for the introduction of the set of rational numbers, we prove that the set of signed integers is an 'integral domain' in which multiplication has the standard algebraic cancellation property. This is done in Theorems 232 and 234. We also show that multiplication is distributive over subtraction, and that the negative of a signed integer can be expressed as its product with the signed integer -1, ie. [0,1].

```
Theorem 232: [Si is an Integral Domain]   (FORALL n in Si | (FORALL m in Si |      ((m *S_TIMES  n = [0,0
```

This completes our work on the basic properties of signed integers.

To prepare for what will come later we give various results stating principles of induction. Many of these are cast as theories, for convenience of use. We also prove various auxiliary results on the set of 'ultimate members' of a set s, i.e. all those t which can be connected to s via a finite chain of membership relations. These are used in some of the work with the principles of induction in which we are interested. The main result is simply that the collection of ultimate members of a set is also a set.

The first theory developed simply tells us that any predicate P of an ordinal which is not always false admits some ordinal for which it is true, but for which the P(t) is false for all smaller ordinals. This tailored variant of the more general principle of transfinite induction stated earlier is sometimes the most convenient form in which to carry out a transfinite inductive proof.

```
THEORY ordinal_induction(o,P)      Ord(o) & P(o)  ==>(t)      Ord(t) & P(t) & t *incin o & (FORALL x in t
```

Next we define the set Ult_membs(s) of 'ultimate members' of a set s, which plays a role in some of our versions of transfinite induction, and prove its properties. The definition is as follows.

```
Def 35a: Ult_membs(s) :=     s + {y: u in {Ult_membs(x): x in s},y in u}
```

The eight elementary theorems which follow state various basic properties of Ult_membs(s). Theorems 236, 239, and 242 state that Ult_membs(s) always includes s, is increasing in s, but is identical to s if s is an ordinal. Theorem 240 states that Ult_membs({S}) is almost the same as Ult_membs(s), containing the set s as its only additional member; Theorem 241 specializes this result to ordinals. Theorem 237 gives a convenient inductive definition of Ult_membs(s), and Theorem 238 states that Ult_membs(s) contains all members of members of s. Theorem 243 tells us that if y in s, then Ult_membs(y) is a subset of Ult_membs(s).

```
Theorem 236: S *incin Ult_membs(S)  Theorem 237: Ult_membs(S) = S + {y: x in S, y in Ult_membs(x)}  Theor
```

Next we give four variants of the principle of mathematical induction, some based on the preceding work with 'Ult_membs', and others specialized to the set of integers. Two of these are designed to facilitate arguments by 'double induction' on a pair of indices.

```
THEORY transfinite_member_induction(n,P)      P(n)  ==>(m)      m in Ult_membs({n}) & P(m) & (FORALL n ir

THEORY double_transfinite_induction(n,k,R)      R(n,k)  ==>(m,j)      R(m,j) & (FORALL n in m | (not R(n,

THEORY mathematical_induction(P)      n in Z & P(n)  ==>(m)      m in Z & P(m) & (FORALL n in m | (not P(

THEORY double_induction(R)      n in Z & k in Z & R(n,k)  ==>(m,j)      m in Z & j in Z & P(m,j) & (FORAL
```

Next we introduce two important 'theories' mentioned earlier: the theory of equivalence classes and the theory of SIGMA. As previously noted, the theory of SIGMA is a formal substitute for the common but informal mathematical use of 'three dot' summation (and product) notations like

$a_1 + a_2 + \ldots + a_n$ and $a_1 * a_2 * \ldots * a_n$.

The theory of equivalence classes characterizes the dyadic predicates R(x,y) which can be represented in terms of the equality predicate using a monadic function, i.e. as R(x,y) *eq (F(x) = F(y)). These R are the so-called 'equivalence relationships', and for each such R defined for all x and y belonging to a set s, the theory of equivalence classes constructs f (for which arb turns out to be an inverse), and the set into which f maps s. This range is the 'family of equivalence classes' defined by the monadic predicate R. The construction seen here, which traces back to Gauss, is ubiquitous in 20th century mathematics.

'sigma_theory' in the following general formulation allows us to associate an overall sum of range values with any single-valued map having a finite domain and range included in a set for which a commutative and associative addition operator with a zero element is defined. It also tells us that summation is additive for pairs of such maps having disjoint domains.

```
THEORY sigma_theory(s,PLUZ,e)      e in s      (FORALL x | (FORALL y in s | x *PLUZ y in s))      (FOR
```

The theory of equivalence classes is as follows. Note that for each such class s, arb(s) is a convenient standard representative for the class.

```
THEORY equivalence_classes(P)   [Theory of equivalence classes]      (FORALL x in s | (FORALL y in s |
```

Returning again to our main line of development, we prepare for the intended introduction of rational numbers (which

follows bit later) by defining the set of formal fractions of signed integers and establishing the algebraic properties of these fractions. This will allow us to define rational numbers as equivalence classes of fractions under the usual 'equality of cross-products' equivalence relationship. Note that the path followed is that of the standard algebraic construction of a field from an integral domain.

The set of fractions is simply the set of ordered pairs of signed integers, of which the second (the 'denominator') must be nonzero.

```
Def 35: Fr := {[x,y]: x in Si, y in Si | y /= [0,0]}
```

Two fractions are equivalent, i.e. stand in the relationship Same_frac(P,Q), if their cross-products are equal.

```
Def 36: Same_frac(P,Q) :=     car(P) *S_TIMES  cdr(Q) = cdr(P) *S_TIMES car(Q)
```

The 'Same_frac' relationship is an equivalence relationship.

```
Theorem 245: (FORALL x in Fr | (FORALL y in Fr |     ((Same_frac(x,y) *eq Same_frac(y,x)) & Same_frac(x,x
```

At this point, the theory of equivalence classes can be used to introduce the set Ra of rational numbers (i.e. the equivalence classes of fractions), and a map Fr_to_Ra of fractions into rationals such that Same_frac(x,y) is equivalent to Fr_to_Ra(x) = Fr_to_Ra(y).

```
Theorem 247:   (FORALL y in Ra | (arb(y) in Fr & Fr_to_Ra(arb(y))  = y)) &       (FORALL x in Fr | Fr_to_
```

Having now introduced rationals as equivalence classes of fractions, we can define the zero and unit rationals, and the algebraic operations on rationals, from the corresponding notions for fractions. The reciprocal of a rational is obtained by simply inverting any of the fractions which represent it. These familiar ideas are captured by the following sequence of definitions. Note that multiplication of fractions is componentwise, but that to add one must first multiply their denominators to put the two fractions being added over a 'common denominator'. Division of rationals is defined as multiplication by the reciprocal, subtraction as addition of the negative. A rational is non-negative if any (hence all) of its representative fractions have numerator and denominator of the same sign; x is greater than (or equal to) y if x - y is non-negative. These standard notions are formalized by the following group of definitions.

```
Def 37: Ra_0 := Fr_to_Ra([[0,0],[1,0]])   Def 37a: Ra_1 := Fr_to_Ra([[1,0],[1,0]])  Def 38: [Rational Sum
```

Our subsequent work with rationals and reals will involve a great deal of elementary work with inequalities between sums and differences, for which the following theory of addition in ordered sets prepares.

```
THEORY Ordered_add(g,e,pluz,minz,rvz,nneg);      e in g & (FORALL x in g |           (x *pluz e = x & x *
```

The next four theorems give miscellaneous ordering properties of the signed integers used to prove corresponding properties of the rationals. If n is a signed integer, either n or -n is non-negative, and if both are non-negative then n is 0. The sum and product of two non-negative integers is non-negative.

```
Theorem 248: (X in Si) *imp ((is_nonneg(X) or     is_nonneg(S_Rev(X))) &     ((is_nonneg(X) & is_nonneg(S
```

Now we begin to work with rationals. Any fraction is a pair of signed integers with non-zero second component. Any member of a rational is a pair of signed integers, and, indeed, a fraction. If two pairs of fractions x,y and w,z are equivalent as rationals, then the sum of x and w is equivalent to the sum of y and z, and similarly for the products. The rational sum of a rational x with the class containing a fraction [y,z] can be obtained by adding any fraction in x to [y,z], and then forming the equivalence class of the result. Much the same statement applies to products of rationals.

```
Theorem 252: X in Fr *eq (X = [car(X),cdr(X)] & car(X) in Si &    cdr(X) in Si & cdr(X) /= [0,0]) Theorem
```

Continuing our work with rationals, we have: The fractions [n,m] and [-n,-m] are equivalent as rationals. If two equivalent fractions both have non-negative denominators, they both have non-negative numerators, and in this case so does their product. A fraction [n,m] is non-negative if and only if [-n,-m] is non-negative. If one of two equivalent fractions is non-negative, so is the other. Rational addition and multiplication are both commutative and associative. The rational sum of a rational x with the class containing a fraction [y,z] can be obtained by adding any fraction in x to [y,z] in the reverse order from that considered just above, and then forming the equivalence class of the result; similarly for products of rationals. The sum of a rational with its negative is the zero rational. The zero rational is the additive identity for rationals. The standard laws of subtraction apply to rationals.

```
Theorem 258: (X in Fr) *imp     Same_frac(X,[Si_Rev(car(X)),Si_Rev(cdr(X))]) Theorem 259: (X in Fr & Y in
```

The next fifteen theorems complete our collection of elementary results concerning rationals.

```
Theorem 274:    (K in Si & N in Si & M in Si & K /= [0,0] &       M /= [0,0]) *imp (Fr_to_Ra([N,M])
```

We have now proved enough about the rational numbers to be able to go on to define the set of real numbers and prove their basic properties. Historically this been done in several ways, which offer competing advantages when computer-based verification is intended. In Dedekind's approach, which is the most directly set-theoretic of all, a real number is defined simply as a set of rational numbers, bounded above, which contains no largest element and which contains each rational x smaller than any of its members. Sums are easily defined for rational numbers defined in this way, but it is only easy to define products for positive rationals directly. This forces separate treatment of real products involving negative reals, causing the proof of statements like the associativity of multiplication to break up into an irritating number of separate cases. For this reason, we choose a different approach, originally developed by Cantor in 1872, in which real numbers are defined as follows. Call an infinite sequence x_n of rational numbers a *Cauchy sequence* if, for every positve rational r, there exists an integer N such that the absolute value $abs(x_n - x_m)$ is less than r whenever m and n are both larger than N. Sequences of this kind can be added, subtracted, and multiplied componentwise and their sums, differences, and products are still Cauchy sequences. We can now introduce an equivalence relationship Same_real between pairs x,y of such sequences: Same_real(x,y) is true if and only if, for every positve rational r, there exists an integer N such that the absolute value $abs(x_n - y_n)$ is less than r whenever n is larger than N. The set of equivalence classes of Cauchy sequences, formed using the equivalence relationship Same_real, is then the set of real numbers. If two pairs of Cauchy sequences x,y and w,z are equivalent, then the (componentwise) sum of x and w is equivalent to the sum of y and z, and similarly for the products and differences. Hence these operations define corresponding operations on the real numbers, which are easily seen to have the same properties of associativity, commutativity, and distributivity, and the same relationship to comparison operators defined similarly.

Given any rational number r we can form a sequence repeating r infinitely often, and then map r to the equivalence class (under Same_real) of this sequence. This construction is readily seen to embed the rationals into the reals, in a manner that preserves addition, multiplication, and subtraction. The zero rational maps in this way into an additive identity for real addition, and the unit rational into the multiplicative identity for reals. If a Cauchy sequence y_n is not equivalent to the zero of reals, then it is easily seen that for all sufficiently large n the absolute values $abs(y_n)$ are non-zero and have a common lower bound. Hence for any other Cauchy sequence x_n we can form the rational quotients x_n/y_n for all sufficiently large n, and it is easy to see that this gives a Cauchy sequence whose equivalence class depends only on that of x and y. It follows that this construction defines a quotient operator x/y for real numbers, and it is not hard to prove that this quotient operator relates to real multiplication in the appropriate inverse way.

```
  Def 46: [The Real numbers as the set of Dedekind cuts]   Re := {s: s *incin Ra | s /= 0 & s /= Ra &      (
```

After the foregoing series of definitions and preparatory theorems we now begin to prove the basic properties of the real numbers. The zero and unit reals are both non-negative, and the unit is larger. The zero real is the additive identity for

reals. The sum and product of two reals and the negative of a real are both reals. The sum of any real and its negative is the zero real. Real addition and multiplication are commutative. The absolute value of a real x is a real which is non-negative and at least as large as x. The absolute value of a real x is x if x is non-negative, otherwise it is the negative of x. The absolute value of a real x is also absolute value of the negative of x.

```
Theorem 296: R_0 in Re & R_1 in Re & R_is_nonneg(R_0) &     R_is_nonneg(R_1) & R_1 *R_GT R_0  Theorem 299
```

Continuing our series of theorems giving elementary properties of real numbers, we have the following. The absolute value of a real number n is at least as large as n, and is non-negative. The sum of a non-negative real n a negative real m has an absolute value which is less than or equal to either n or the reverse of m. The sum of n and the absolute value of m is at least as large as m. The absolute value of n + m is no larger than the sum of the absolute value of m and the absolute value of n. The sum, product, and quotient of two real numbers is a real number. The absolute value of the product of n and m equals the product of the two separate absolute values, and a similar result holds for the quotient. Real addition and multiplication are commutative and associative, and multiplication is distributive over addition. The sum of two non-negative reals is non-negative. The negative of the negative of a real n is n. The unit real is the multiplicative identity, and the product of any nonzero real with its reciprocal is the unit real. Division of reals is the inverse of real multiplication. The only real number n for which n and -n are both non-negative is the real zero. If the sum of two non-negative reals m and n is zero, then both m and n are zero. If n is greater than n and k is positive, all being reals, then the product of n and k is greater than the product of m and k. The reciprocal of a positive real is positive. The average of two reals n and m lies between n and m. There is one and only one non-negative square root of a non-negative real. If both m and n are non-negative reals, the square root of their product is the product of their separate square roots.

```
Theorem: (N in Re) *imp (abs(N) in Re &    (abs(N) *R_GT N or abs(N) = N) &     (abs(N) *R_GT R_0 or abs(
```

This completes the elementary part of our work with real numbers. Since one of our main goals is to state and prove the Cauchy integral theorem, we must also define the complex numbers and prove their basic properties. This is done in the entirely standard way which traces back to Gauss. Complex numbers are defined as pairs of real numbers. They are added componentwise, and multiplied in a manner reflecting the desire to make [0,1] a square root of -1. The norm of a complex number is its length as a two-dimensional vector. The reciprocal of a complex number is obtained by reversing its second component and then dividing both components of the result by the square of its norm. The quotient of two complex numbers is the first times the reciprocal of the second. The zero complex number is the pair whose components are both the zero real. The unit complex number has the unit real number as its first component.

```
Def 58: [Complex Numbers] Cm := Re *PROD Re  Def 59: [Complex Sum] def(X *C_PLUS Y) :=     [car(X) *R_PLU
```

The basic elementary properties of the complex numbers are now established by a series of elementary algebraic proofs. Any pair of reals is a complex number and vice-versa. The complex sum and product of any two complex numbers is a complex number, and vice-versa. The zero complex number is the additive identity, and the unit complex number is the multiplicative identity. The negative of a complex number is its additive inverse. Complex addition and multiplication are commutative and associative; multiplication is distributive over addition. The norm of amy complex number is a non-negative real number. The negative of a complex number z has the same norm as z. The norm of a complex product is the product of the separate norms. The norm of a complex quotient is the quotient of the separate norms. Any nonzero complex number has a multiplicative inverse, the inverse of multiplication being given by the complex division operator, which is easily defined using the complex reciprocal.

```
Theorem: ((X in Re & Y in Re) *imp ([X,Y] in Cm)) &    ((m in Cm) *imp       (m = [car(m),cdr(m)] & car(m
```

Now we take our first steps into analysis proper, i.e. take up the theory of functions of real and complex variables. The set RF of real functions is defined as the set of all single-valued functions whose domain is the set Re of all real numbers and whose range is a subset of Re. The zero function is that element of RF all of whose values are zero. Functions in RF are added and multiplied pointwise, reversed pointwise, and compared pointwise. The least upper bound of any set of

functions in RF is formed by taking the least upper bound of the function values at each point. The positive part of a real function is formed by taking its pointwise maximum with the identically zero real function.

```
Def 64: [Real functions of a real variable]     RF := {f *incin (Re *PROD Re) | Svm(f) & domain(f) = Re}
```

The most elementary properties of real functions follow directly and trivially from these definitions. Addition and multiplication of real functions are commutative and associative; multiplication of such functions is distributive over addition.

```
Theorem: (N in RF & M in RF) *imp     (N *F_PLUS M = M *F_PLUS N)  Theorem: (N in RF & M in RF) *imp
```

To progress to less trivial results in real analysis we need to define various basic notions of summation and convergence. In order to arrive at our target, the Cauchy integral theorem, with minimal delay, we ruthlessly omit all results not lying along the direct path to this target, even though inclusion of many of these results would usefully illuminate the lines of thought that enter into the definitions, theorems, and proofs we are compelled to include. This may lead the reader not previously familiar with analysis to feel that we are giving many bones with little meat. For a fuller account of the historical and technical background of the results from analysis presented in this book, any introductory account of real and complex function theory can be consulted. Among these we note Douglas S. Bridges, *Foundations of Real and Abstract Analysis* (Springer Graduate Texts in Mathematics, v. 174, 1997); also the older classic *Foundations of analysis; the arithmetic of whole, rational, irrational, and complex numbers.* by Edmund Landau (Chelsea Pub. Co., New York, 1951).

The sum of the values of any real-valued mapping having a finite domain is defined by specializing the general 'Theory of Sigma' described above to this general case. We can then define the sum of a convergent series of positive real values (on any domain) as the least upper bound of all its finite sub-sums. (Note however that it can easily be shown that this value will only be a finite real if no more than a countable number of the function values are non-zero). By further specializing the 'Theory of Sigma' using real function addition rather than real addition we can define the notion of sum for finite series of real functions, and then by taking least upper bounds we can define the sum of a convergent series of positive real functions.

```
Def_by_app 73: [Sums for Real Maps with finite domains]    (Svm(f) & range(f) *incin Re & Finite(f)) *imp
```

It is now easy to give the basic definitions of the theory of integration of real functions. We first define the notion of a 'block function'. This is simply a function of a real variable which is zero everywhere outside a bounded interval of reals, and constant inside this interval. We introduce a name for the set of all such functions. The 'integral' of any such function is the length of the interval on which it is nonzero, times its value. The Lebesgue 'upper integral' of any positive real-valued function f of a real number is the greatest lower bound of all sums of integrals of countable sequences f_i of positive block functions for which the pointwise sum of the sequence of values $f_i(x)$ is at least as large as f(x) for each real x. (It is easily seen that this value depends only on the positive part of f). The (Lebesgue) integral of any real function f is the upper integral of f minus the upper integral of the negative of f. The key result at which these definitions hint (but, of course, do not prove), is that this integral is additive for a very wide class of functions, and that if a sequence g_n of functions in this class converges (in an appropriate sense) to a limit function g, then the integrals of the g_n converge to the integral of g.

```
Def 76: [Block function]  Bl_f(a,b,c) :=     {[x,if a R_le and x R_le b then c else R_0 end if]: x in R}
```

Note that this last definition describes the procuct of an arbitrary collection s of sets; this is the set of all members of any chosen member of s which belong to all the other mebers of s.

```
Def 80: [Lebesgue Upper Integral of a Positive Function] ULeInt(f) := GLB({{[n,BFInt(ser~[n])]: n in Z}:
```

We also need to develop some of the results concerning continuity and differentiability which lie at the traditional heart of analysis. We begin by giving the standard 'epsilon-delta' definition of continuity: a single-valued, real-valued function f of a real variable is continuous if for each x in its domain, and each positive real value 'ep', there exists some real value 'delt' such that the absolute value of the real difference f(x) - f(y) is less than 'ep' whenever y belongs to the domain of f and the absolute value of the real difference x - y is less than 'delt'. Since for later use we will need to generalize notions like this to the multivariable case, we also define the notion of Euclidean n-space (namely as the collection of all real-valued sequences of length n, i.e. the set of all real-valued functions defined on the integer n), and the standard norm, i.e. vector length in this space, which is the square root of the sum of squares of the components of a vector (i.e. the values of the corresponding function). We also need the notion of the (componentwise) difference of two n-dimensional vectors, which we define as the pointwise difference of the functions corresponding to these vectors. This lets us extend the 'epsilon-delta' definition of continuity from real functions of real variables to vector-valued functions of vector-valued variables, and also real-valued functions of vector-valued variables. A vector-valued function f of a vector-valued argument x is continuous if for each x in its domain, and each positive real value 'ep', there exists some real value 'delt' such that the norm of the vector difference f(x) - f(y) is less than 'ep' whenever y belongs to the domain of f and the norm of the vector difference x - y is less than 'delt'. The definition of continuity for real-valued functions of vector-valued variables is similar.

```
Def 82: [Continuous function of a real variable]    is_continuous_RF(f) :*eq f *incin (Re *PROD Re) & Svm
```

Our next aim is to define the notion of derivative in some convenient way. We do this by considering pairs of real-valued functions f, df of a real variable x, and forming the function g of two real variables x and y which equals the difference-quotient (f(x) - f(y))/ (x - y) if x and y are different, but df(x) if x = y. Then f is said to be (continuously) differentiable in its domain D if there exists some continuous function df having the same domain such that the function g, formed in this way, is continuous on the product set of D with itself. It is easily seen that f if is differentiable there can exist only one df which makes g continuous, allowing us to speak of *the* derivative of f if f has a derivative. It is also easy to see that if two functions f and h of a real variable have derivatives df and dh respectively, then so do their sum and product, and that the derivative of the sum is df + dh, while the derivative of the product is df * h + f * dh. (However, we do not give the proofs of these results).

```
Def 87: [Difference-and-diagonal trick]  DD(f,df) := {if x~[0] /= x~[1] then (f(x~[0]) *R_MINUS f(x~[1]))
```

Next we extend the preceding notions to complex functions of a complex variable (i.e. single-valued functions defined on the set of complex numbers whose range is included in the set of complex numbers), and to complex-valued functions on complex Euclidean n-space. This space is defined as the collection of all real-valued sequences of length n, i.e. the set of all complex-valued functions defined on the integer n, and the difference of vectors is defined as the pointwise difference of the corresponding functions. The norm for such vectors is defined as the sum of the squares of the absolute values of their (complex) components. Using this simple definition of norm, the standard 'epsilon-delta' definition of continuity extends readily to the complex case.

```
Def 89: [Complex functions of a complex variable]   CF := {f *incin (Cm *PROD Cm) | Svm(f) & domain(f) =
```

It is now easy to extend the 'difference-and-diagonal trick' used to define the derivative of real-valued functions of a real variable to the complex case. Again we consider pairs of functions f, df, this time complex-valued functions of a complex variable x, and form the function g of two complex variables x and y which equals the difference-quotient (f(x) - f(y))/ (x - y) if x and y are different, but df(x) if x = y. Then f is said to be (continuously) differentiable in its domain D if there exists some continuous function df having the same domain as f such that the function g, formed in this way, is continuous on the product set of D with itself.

```
Def 95: [Difference-and-diagonal trick, complex case]     CDD(f,df) := {if x~[0] /= x~[1] then (f(x~[0])
```

It has been known since the 1821 work of Cauchy that the consequences of differentiability for complex functions of a

complex variable (defined in an open subset of the complex plane) are much stronger than the corresponding assumption in the real case, a fact for which our target theorem, the Cauchy integral theorem, is central. Here a subset of the complex plane is sad to be *open* if it contains some sufficiently small disk around each point of its domain. Functions of a complex variable diffentiable in an open set are said to be *analytic* functions of the complex variable. One such function, of particular importance, is the complex exponential function, which can be defined as the unique analytic function 'exp' having the entire complex plane as its domain which is equal to its own derivative and takes on the unit complex value at the zero point of the complex plane. The two mathematical constants 'e' and 'pi' can both be defined in terms of this function, in the following way: 'e' is the value which exp takes on at the point [R_1,R_0] of the complex plane, and 'pi' is the smallest real positive x for which exp([R_0,x]) is R_rev(R_1). That is, we define 'pi' as the smallest positive root of Euler's famous, indeed ineffable, formula $e^{i * pi} = -1$.

```
Def 97: [Open set in the complex plane]     is_open_C_set(s) :*eq (FORALL z in s | (EXISTS ep in Re |
```

To move on to the statement, and eventually the proof, of Cauchy's integral theorem we must define the notion of 'complex line integral' involved in that theorem. For this, we need various slight modifications of the foregoing material, and in particular the notions of continuity and differentiability for complex-valued functions of a real variable. These involve the following easy modifications of the 'epsilon-delta' definition and the difference-and-diagonal trick described above. A ('closed') real interval is the set of all points lying between two real values (including these values themselves). A *continuously differentiable curve* in the complex plane is a continuous complex-valued function defined on an interval of the real line which is continuously differentiable on its domain. The complex line integral of a complex-valued function f defined on such a curve is defined by taking the complex product of f by the derivative of the curve, integrating the real part (i.e. pointwise first component) and the imaginary part (pointwise second component) of the resulting product function, and rejoining these two values into a complex number.

```
Def 101: [Continuous complex function on the reals]    is_continuous_CoRF(f) :*eq f *incin (Re *PROD Cm)
```

Now finally we can state the Cauchy integral theorem and the Cauchy integral formula derived from it. The Cauchy integral formula states that if f is an analytic function defined in some open subset of the complex plane, and if c_1 and c_2 are two continuously differentiable closed curves (i.e. curves which end where they start), both having ranges in s, and if each of the values of c_1 differs sufficiently little from the corresponding value of c_2, then the two line integrals of f over the two curves must be equal. This is proved by deforming the first curve smoothly into the second, and proving that the derivative of the resulting line integral in the deformation parameter must be zero: a function of a real parameter whose derivative is zero in an interval must be constant in that interval.

To avoid topological complications we state the Cauchy integral formula, which follows from the Cauchy integral theorem, in a somewhat special case: If f is a function analytic in an open set including the closed unit circle of the complex plane, and z is any point interior to that circle, then the line integral of the quotient f(w) / (2 * pi) * (w - z) over the unit circle is always f(z). Note that in the formal statement of this theorem given below, the unit circle is represented by the curve w = C_exp_fcn([R_0,x]), where the real parameter value x varies between 0 and 2 * pi. Though we do not follow up on its possible generalizations, Cauchy integral formula can be stated much more generally: it is true whenever f is analytic in a domain of any shape including the whole of any smooth closed complex curve in the complex plane and its interior, provided that w is a point interior to the curve about which the curve winds just once. But to state and prove the Cauchy integral formula in this generalized form we would need to develop the theory of winding numbers, which would extend the present work beyond its appointed length.

```
Theorem: [Cauchy integral theorem]   is_analytic_CF(f) *imp      (EXISTS ep in Re | (ep *R_GT R_0 &

Theorem: [Cauchy integral formula]    (is_analytic_CF(f) & domain(f) incs {z in Cm: R_1 *R_GE C_abs(z)}) *
```

3.13. Other Implemented Systems

It is useful to review a representative sample of the many proof verifiers and theorem provers which have been developed over the last three decades and to make some comments which compare them, emphasizing features and facilities of interest in connection with the design of the verifier described in this book.

(i) **The Boyer-Moore System.** The proof verifier system developed and refined by Boyer and Moore over the last 25 years is particularly rich and suggestive (cf. [ref. 20], [ref. 21]). This system is oriented toward proofs of properties of recursively defined functions of integers and nested structures built of ordered pairs. The system works very effectively with algebraic arguments involving such functions, and is also able to generate inductive arguments automatically. Typing of variables and functions is used effectively. The system uses generalization, i.e. substitutions of terms for variables, in order to obtain simpler formulae which are more readily handled by inductive arguments. Surprisingly effective heuristics for combination of propositional, algebraic, and inductive arguments allow it to generate interesting proofs in many cases. The system's recursive LISP-like formalism adapts it well to program verification applications, especially of programs which emphasize recursion, and it has been used successfully to verify a wide variety of theorems of number theory and logic, plus a collection of more applied, surprisingly complex programs (cf. [ref. 22]-[ref. 24]).

The Boyer-Moore system incorporates a metamathematical extension mechanism allowing addition of new primitive routines to its initial endowment (cf. [ref. 24]). Mechanisms for dealing with generalized 'quantifiers' (including finite set and sequence-formers, summation and product operators, existential and universal quantifiers) were incorporated into the system's second version (see [ref. 211]). (That this was required points implicitly at an advantage of the set-theoretic approach used in this book; since set theory is so general, extensions of this kind can be carried out in set theory itself. In more limited systems, new and possibly disturbing additions must be made to accomplish the same thing).

The central objects of the Boyer-Moore theory are maps which our proposed set-theoretic verifier would represent in a form like

```
f = {[n,f(n)]: n in s}.
```

or, to illustrate a multivariate case, in some such form as

```
f = {[[n1,n2,n3], f([n1,n2,n3])]: n1 in s, n2 in s,n3 in s]} .
```

(Here s is a well-founded set, typically either the integers or the set of all hereditarily finite lists). This makes it plain that the algebraic and inductive manipulations which appear in Boyer-Moore proofs have set-theoretic analogs (and generalizations).

The very effective Boyer-Moore proof-generation heuristics could be made available for use in our set theoretic verifier by developing a 'cross-compiler' which allows facts about set-formers (more properly, 'map-formers') of the kind shown above to be deduced in a Boyer- Moore like notation by a superstructure mechanism which automatically translates a Boyer-Moore proof into a sequence of steps valid to the inferential core of our set-theoretic verifier. However recursive proofs are much less typical for our set-theoretic verifier than they are in the Boyer-Moore system. This is because set theory emphasizes explicit, in effect iterative, modes of expression that are less readily expressed in studiously finite, LISP-inspired formalisms like that of Boyer and Moore. For example, in the Boyer-Moore formalism, integer addition and multiplication are defined recursively, and the natural proof of such facts as

```
x + (y+z) = (x+y) + z, x * (y * z) = (x * y) * z,
```

and

```
x * (y+z) = x * y + x*z
```

are inductive (where here, but not below, '+' and '*' designate arithmetic addition and multiplication respectively).. In a set theoretic formalism, these assertions follow more naturally from the properties of the cardinality operator #s, especially the fact that there exists a 1-1 map with range s and domain #s, and that #f = #domain(f) if f is single-valued, so that #f = #range(f) if f is 1-1. Then a quantity s is a cardinal if s = #s, arithmetic addition is defined by

```
m *PLUS n = #(m + {[m,x]: x in n}),
```

subtraction is defined by

```
    m *MINUS n = {k in m | k *PLUS n in m},
```

and multiplication by

```
    m *TIMES n = #{[x,y]: x in m, y in n} = #(m *PROD n),
```

where *PROD designates Cartesian product. The basic properties of the arithmetic operations follow from such arguments as

```
#((x *PROD y) *PROD z) =          §(#(x *PROD y)) *TIMES z = (x *TIMES y) *TIMES z ,
```

and similarly

```
    #(x *PROD (y *PROD z)) = x *TIMES (y *TIMES z)
```

so that

```
   (x *TIMES y) *TIMES z = x *TIMES (y *TIMES z)
```

since

```
((x *PROD y) *PROD z) and (x *PROD (y *PROD z))
```

are in natural 1-1 correspondence.

In much the same way, the basic properties of arithmetic exponentiation follow from the definition m**n = #maps(n,m); this approach is even more foreign to the recursive orientation of the Boyer-Moore prover.

The basic properties of sequence concatenations, another situation which the inductive approach of the Boyer-Moore prover handles successfully, furnishes another revealing comparison. In set theoretic terms, a sequence f is a finite map whose domain is an integer, i.e. satisfies domain(f) = #f. The concatenation of two such maps is most conveniently defined by

```
append(f,g) =   {[n,if n in #f then f(n) else g(n *MINUS #f))]: n in #f *PLUS #g}.
```

It follows that #append(f,g) = #f *PLUS #g, so that if h is a third sequence we have

```
append(append(f,g),h) = {n, if n in #f *PLUS #g then   append(f,g)(n) else h(n *MINUS #f *MINUS #g)]: n in
```

Similarly,

```
append(f,append(g,h)) = {[n,if n in #f then f(n)       elseif n in #g then g(n *MINUS #f)      else h(n *M
```

proving the associativity of the 'append' operator. Many other statements whose natural Boyer-Moore treatment is inductive will have relatively direct set/map former algebraic derivations in our set-theoretic environment.

The ideas concerning heuristics for finding induction proofs developed in the Boyer-Moore theorem prover are also exploited in the Karlsruhe induction theorem proving system (INKA) developed at the University of Karlsruhe (see [ref. 213]).

(ii) **Edinburgh LCF.** Another proof verifier which implements an induction schema (though one of quite a different flavor than that of the Boyer-Moore system) is the Edinburgh LCF system (see [ref. 54]). LCF is based on a typed g-calculus extended by a fixed point operator (polymorphic predicate lambda calculus) inspired by Dana Scott's work on continuous functions on lattices. LCF is further strengthened by inclusion of a strongly typed programming language ('ML') which allows programming of proof strategies or 'tactics', that can be added to improve LCF's ability to manage proofs in whatever specific theory one is working on ([ref. 76]). Problems concerning transformation of programs, translation between different programming languages, and other issues concerning the syntax and the semantics of programming languages have been handled elegantly in LCF.

(iii) **The NUPRL Verifier.** The NUPRL system developed by R. Constable and his associates at Cornell, see http://www.cs.cornell.edu/Info/Projects/NUPRL/NUPRL.html, is a full type-theory based proof verifier (see [ref. 82]). Its style of formalization extends the AUTOMATH approach, which uses on a very high level language to write mathematical text (see also [ref. 35]-[ref. 36]). NUPRL supports some of the basic notions of set theory, restricted, however, by its authors' commitment to constructivism and preference for a type-theoretic rather than a pure set-theoretic approach. Its definitional and proof-theoretic capabilities are sufficient to give NUPRL access to the whole of mathematics, at least in its constructivist version, e.g. the constructive reals can be defined, and in principle it should be possible to use NUPRL to formalize all the arguments in Bishop's well-known book on constructive analysis (cf. [ref. 109]). A programming superstructure (very close to that of the Edinburgh LCF system) is available and makes it possible to combine patterns of elementary inference steps into compressed 'tactics' for subsequent use.

NUPRL and its follow-up MetaPRL share some of the concerns of the present book but develop them in quite a different direction. NUPRL implements a form of computational mathematics and sees itself as providing logic-based tools that help automate programming. This is done by adhering carefully to a style of logic influenced by ideas developed by Martin-Löf, which embodies a constructive type theory, that guarantees that each proof of the existence of an object y satisfying a relationship R(x,y) translates into a program for constructing such an object. The authors of the NUPRL system note that "In set theory one can use the union and power set axioms to build progressively larger sets.. In type theory there are no union and power type operators... Both type theory and set theory can play the role of a foundational theory. That is, the concepts used in these theories are fundamental. They can be taken as irreducible primitive ideas which are explained by a mixture of intuition and appeal to defining rules. The view of the world one gets from inside each theory is quite distinct. It seems to us that the view from type theory places more of the concepts of computer science in sharp focus and proper context than does the view from set theory..." The system presented in this book resolutely ignores the constructivist concerns which have influenced the design of the NUPRL system, and which give it details different from those of standard, set-theoretically based mathematics. We aim instead at a thoroughly set-theoretic, often non-constructive form of logic coming as close as possible to the traditional and comfortable forms of reasoning customary in most mathematical practice.

In further comparing our verifier to the NUPRL system, we can emphasize two differences. Though adequate, the inference rules of NUPRL are considerably more rudimentary than those included in our verifier. For example, in our intended verifier the statement

```
(a incs b & c incs d) *imp pow(a + c) incs pow(b + d)
```

follows very quickly using an inference mechanism, described in Chapter 3, which exploits the fact that the powerset operator 'pow' on sets is monotone increasing. In NUPRL, even the statement of this fact raises some technical difficulties

(all the variables must be typed, the power-set operation pow(s) is not immediately available (though a NUPRL type such as Boolean to A, where A is the type corresponding to the set s, may be useable as a substitute), etc.). Although no one of these technical difficulties is difficult to overcome in isolation, their cumulative weight is substantial.

The proof of the well-known 'pigeonhole' principle, namely

```
#s > #t & Svm(f) & domain(f) = s & range(f) = t        *imp (EXISTS x in s, y in s | x /= y & f(x) = f(y))
```

furnishes another interesting comparison between NUPRL and the system described in this book. In NUPRL the proof of this fact, for finite sets, is inductive; see [ref. 82], pp. 221-228 for a proof sketch. Set-theoretically we can use the basic lemma

```
one_one(f) *imp #range(f) = #domain(f),
```

which expresses a basic property of the cardinality operator (for finite maps), to deduce

```
Svm(f) & (not one_one(f))
```

as an immediate consequence of the hypothesis of (1), from which

```
(EXISTS x in s, y in s | x /= y & f(x) = f(y))
```

follows almost immediately by definition and since s = domain(f).

In general, we believe that the very strong typeless variant of Zermelo-Fraenkel set theory used in our verifier provides a considerably smoother framework for ordinary mathematical discourse than the type-theory on which NUPRL is based. Moreover, we believe that this set theory is close enough to common mathematical reasoning to be acceptable to the ordinary mathematician as a straightforward formalization of his ordinary working style. In contrast, formalized type theories like that embodied in NUPRL seem to be substantially less familiar, and so substantially greater mental effort must be invested before one can become comfortable with the mathematical viewpoint they embody.

(iv) **The Stanford FOL system.** FOL is a pioneering proof checker developed at Stanford by J. McCarthy, R. Weyhrauch and their group during 1975-1990 (cf. [ref. 94]-[ref. 95]).

FOL is based on a natural deduction style of reasoning for first order logic. Its language is a multi-sorted first order language. Sorts are partially ordered (e.g. 'even integers' are contained in 'integers'), allowing for a mild degree of type checking.

Derivations in FOL can take advantage of a few basic decision procedures, e.g. a tautology checker, a procedure which decides formulae of the monadic predicate calculus, and a syntactic-semantic simplifier. FOL supports a mechanism of semantic attachment which associates LISP objects with functions and predicate symbols. This allows one to use the full strength of the underlying programming language LISP for certain limited proof portions. For example, one can deduce the equality

```
SQUARE(*PLUS '1' '1') = '4'
```

in one step, having attached + to *PLUS, x**2 to SQUARE and the numbers 1 and 4 to the numerals '1' and '4' respectively.

The syntactic simplifier of FOL is a pattern matching algorithm together with an extensive set of rewrite rules.

The most recent implementations of FOL allow one to give meta-theoretic arguments and to use them to extend the system. This allows heuristics to be described declaratively (at a meta-meta-level).

(v) **The EKL proof checker.** The EKL proof checker developed at Stanford (see [ref. 61]) is a finite-order predicate calculus proof verifier, close in spirit to the FOL system. As in FOL, a variant of natural deduction is supported. To accelerate predicate reasoning, variables and functions are systematically assigned types. EKL implements two special forms, FINSET (finite sets of the form $\{t_1,t_2,...,t_n\}$) and CLASS (set formers of the type $\{x: P(x)\}$, where P(x) is a predicate calculus formula), which coupled with rewriting rules, allow manipulation of elementary set- theoretic constructs.

Rewriting and simplification, based on a systematic use of equalities and rewriting rules, are the most basic and widely applied rules in the EKL system. As in ordinary mathematical practice, definitions are used extensively by EKL's unifier and rewriter. A small decision procedure which quickly verifies whether conditions for rewriting are met is provided. Also, semantic attachment of LISP objects to EKL constants and functions permits some automatic metatheoretic simplifications.An extensible predicate formula simplifier, based on systematic use of equalities and rewrite rules, and a global rewriting procedure like that of the lambda-calculus, is provided.

A notion of 'trivial' inference is captured in EKL's semi-decision procedure DERIVE, which handles deductions in a small fragment of predicate calculus.

EKL's administrative routines are written in LISP making the family of available routines easy to extend.

(vi) P. Suppes' Stanford University **EXCHECK** system is a third proof verifier developed at Stanford. This interesting variant of the FOL system was used for fifteen years in computer assisted instruction at the university level. In addition to its basic theorem proving module, EXCHECK includes unusual facilities which enlarge the interface between the system and its intended student audience, e.g. provides a speech synthesizer; a 'lecture' compiler; a course driver allowing several modes of lesson presentation (LESSON, BROWSE, and HELP), and procedures which analyze and summarize student-supplied proofs.

The EXCHECK prover module uses an adaptation of Suppes' variant of natural deduction. Predicated terms are assigned types and the program is able to compute the type of complex term on the basis of the information available to it. This lets the system avoid many useless invocations of domain-dependent rules of inference.

EXCHECK proof procedures fall into two general classes: inferential procedures and reductive procedures. Inferential procedures handle basic mathematical inferences, including both pure predicate rules and some rules specialized to particular mathematical theories which the system handles, for example

- TAUTOLOGY, for tautologies;
- TEQ, for tautologies plus substitution of equals;
- BOOLE, for elementary set algebra (i.e. the unquantified language of set union, intersection, difference, subset, identity, conjunction, implication, disjunction, and negation (see [ref. 212]).
- IMPLIES, for applying previous results or definitions;
- ESTABLISH, for deciding various other elementary set-theoretic statements. This is a partial decision procedure based upon natural deduction heuristics, some elementary set theoretic proof procedures, and a system of rewrite rules for elementary set theory that simplifies an inference and then uses other resolution or decision procedures.

EXCHECK also implements a class of reductive proof procedures, i.e. procedures which reduce problems to subproblems. Those include natural deduction rules such as elimination of conditionals, disjunctions, conjunctions, quantifiers, etc, and also lemma and subgoal reductions which allow the user to work with proof goals general enough to guide multiple proof steps.

EXCHECK's reductive proof procedures can sometimes discover how its user is structuring a proof, and so in some cases give aid, based on a proofs' goal-structure representation, to the user. EXCHECK's proof mechanisms often allow elementary set-theoretic proofs to be expressed quite succinctly, and so fit the system for its intended instructional use.

(vii) **The Argonne AURA, ITP, and LMA resolution packages.** L. Wos and his group at the Argonne National Laboratory have studied resolution based theorem proving very extensively, focusing on fully automatic theorem proving, i.e. on predicate inference algorithms which can prove substantial theorems without detailed guidance.

Many efficient variants of resolution and paramodulation (see above) have been developed by Wos and his collaborators (cf. [ref. 179]-[ref. 185]) and implemented, first in their very powerful theorem prover AURA (cf. [ref. 96]), and subsequently in the interactive theorem prover ITP, which is systematically organized using their Logic Machine Architecture (LMA) (cf. [ref. 67]-[ref. 73]).

The heuristic power of the Automated Reasoning Assistant AURA is shown by its ability to solve various open questions in ternary Boolean algebra (i.e. deciding the independence of three given axioms from a set of five others), in the calculus of equivalence, and in the subfield of formal logic known as R-calculus and L-calculus (cf. [ref. 97]). The burden of detail and exceeding length of the formulae involved in some of the highly combinatorial, intuition-free proofs that occur in these areas impedes human attacks, but proves to be manageable by AURA.

The collection of resolution procedures constituting the overall Argonne inference package LMA was documented systematically so that it could be used in other verifiers. The package is organized into four layers, atop a layer 0 which various utility data types to the Pascal language in which the system is written. Layer 1 implements the fundamental data type 'object' (representing a formula of logic) and a family of primitive operations on such objects, such as access, substitution, unification, etc. Layer 2 supports an extensive collection of predicate inference mechanisms, including hyperresolution, paramodulation, demodulation, subsumption, etc. Layer 3 provides a family of more composite theorem proving processes which implement various search heuristics.Layer 4 allows configurations of intercommunicating processes to be constructed out of the layer 3 subsystems.

(viii) **The University of Texas interactive theorem prover.** The UT interactive theorem prover is a natural deduction system developed by Bledsoe and his group at the University of Texas at Austin as part of a long-term project focused on theorems of analysis (see [ref. 4]-[ref. 14]). This system is structured around a central routine, IMPLY, which applies to skolemized formulae in implicative form.

IMPLY tries to find the most general substitution which satisfies the formula passed to it. This is accomplished by attempting to apply a list of subrules until one of them succeeds. These subrules can call IMPLY recursively; in such a case the substitution returned is obtained by composing the substitutions are returned by the separate calls to IMPLY. Each substitution instantiates one or more existentially quantified variables.

Some of the rules used by IMPLY are:

- ANCESTRY (checks whether the negation of the consequent of an implication is a proof goal for some prior rule in a recursive stack);
- AND-SPLIT (splits a current goal of the form H *imp A & B into two subgoals H *imp A and H *imp B);
- CASES (like AND-SPLIT, but for formulae of the form (A or B) *imp C);
- REDUCE (tries to apply one of a collection of domain dependent rewriting rules);
- INEQUALITY (calls a decision procedure for general inequalities; see [ref. 14]);
- MATCH (checks whether some conjunct of an hypothesis unifies with the consequent of the implication including it);
- BACK-CHAIN;
- SUBSTITUTION OF EQUALS;
- DEFINE-C (expands definitions from a stored list);

The inferential core of the UT system has been adapted to various fields of mathematics. In [ref. 13], for example, applications in elementary set theory are reported; [ref. 9] discusses applications to general topology. Other applications studied include elementary real analysis (limit theory), nonstandard analysis, problems arising in program verification, etc.; (see [ref. 8], [ref. 12]). Bledsoe and his group developed heuristics for each of these fields of mathematics. Careful use of reduction rules then allowed various standard lines of reasoning in these theories to be reconstructed procedurally. The UT prover has sometimes been able to prove theorems which would have required very long proofs and entry of numerous axioms if submitted to a resolution based theorem prover.

The UT prover includes routines which ease user interaction with the system. For example, one can guide proof search for by calling up previously proved lemmas using a command USE. Prestored results are not always invoked automatically, since this might lead to a combinatorial explosion. This approach gains efficiency in important elementary cases, but reveals that the IMPLY procedure is incomplete.

The Texas approach differs from that of our verifier in that we emphasize decision procedure more strongly and avoid substantial use of heuristics.

(ix) **The Kaiserslautern University theorem prover project.** A large resolution-based theorem prover was developed at Kaiserslautern University by J. Siekmann starting in 1976. This verifier aims to realize the most efficient variants of resolution developed by the AURA project [ref. 96], together with an induction prover improving on that of Boyer-Moore [ref. 20], plus various facilities, described below, which allow the system to be extended by heuristics.

The core of the system is an automated theorem prover for a multi- sorted first order propositional calculus, which uses Kowalski's connection graph proof procedure [ref. 176] extended with paramodulation techniques.

This 'Markgraph Karl Refutation Procedure' (MKRP) system is (as noted by its authors (cf. [ref. 60])) also designed to accept heuristics and domain-specific knowledge. A main aim of is design is to realize a theorem prover which allows its user to mitigate the combinatorial explosion which a brute force refutation search would generate. For this reason its authors have chosen to represent clauses by connection graphs, since such graphs can often be held to small size by providing appropriate reduction rules. However this sometimes causes the number of links in a connection graph to become very high, slowing the derivation process. To avoid this, the MKRP system integrates special ways of dealing with tautologies, subsumption, and certain other special connection graph forms. Unfortunately, this results in the loss of completeness and in certain cases (very unusual in ordinary practice) in loss of consistency. Theoretical studies were therefore undertaken to eliminate inconsistency without losing efficiency.

To further improve speed, the MKRP system makes use of a subprogram, TERMINATOR, which implements a non-recursive look-ahead for detecting conditions which guarantee the unsatisfiability of input formulae.

An intended application shaping the design of many theorem provers is to prove correctness of programs. Formulae produced by automatic generation of correctness conditions from annotated programs tend to be long, highly redundant, and shallow. To enhance its efficiency in dealing with theorems having these characteristics, the MKRP system includes a preprocessing phase, which applies simplification rules to arithmetic and logical expressions, equalities and inequalities, and simple statements in set theory and other simple logical constructs, together with other fast decision procedures which serve to simplify input formulae. Such procedures can be combined using a variant of the Nelson-Oppen method for quantifier-free theories [ref. 173].

An induction proof system has been implemented within the MKRP-system, also a compiler for a predicate logic programming language (PLL) suitable for expressing properties of inductively defined functions and predicates, along with a simplification module for PLL formulae, was implemented. Other extensions include

- a recursion-analysis module (for deciding whether newly defined functions and predicates satisfy a well-foundedness criterion);

- a module which uses program synthesis techniques to instantiate (and eliminate) existential quantifiers;
- a control module which handles interactions between the induction system and the underlying MKRP system.

A verification condition generator for programs written in PASCAL was developed, which together with MKRP can be used as a full program verification system for PASCAL. The Kaiserslautern group also studied the design of a verification condition generator for COBOL.

The MKRP group has designed a proof reporting module which can transform resolution style proofs into natural deduction proofs and then translate this natural deduction proof into a proof stated in natural language.

(x) **AFFIRM.** The AFFIRM theorem prover (cf. [ref. 49]-[ref. 50]) developed at the USC Information Sciences Institute is an interactive system, based on natural deduction, partly inspired by the prover developed by Bledsoe's Texas group. The formalism it uses is essentially predicate calculus, extended by special mechanisms for establishing properties of recursively structured abstract data types. AFFIRM supports all standard natural-deduction mechanisms, e.g. allows one to assume a lemma (to be proved later or which has been proved in another proof session), split the proof of a current goal into smaller subgoals, reason by cases, instantiate existentially quantified variables (subject to Skolem dependency constraints), perform substitutions using primitive definitions, etc.

Formulae are represented internally in a Skolem normal form, so that when defined symbols need to be expanded with their definition, there is no need to unskolemize and then reskolemize.

When its 'search' command is invoked, AFFIRM uses a 'chaining and narrowing' method to seek instantiations which allow proof of its current goal. Several options are provided which allow one to focus the prover on critical substeps. The system is sometines able to use simple dependence rules to determine the next subgoal to be proved.

AFFIRM also supports special mechanisms oriented toward proof by structural induction of properties of recursively defined, tree-like data- structures; base cases and inductive steps of such proofs can be set automatically. The system can apply automatic simplifications based on equational specifications for such data abstractions. Where possible, equations are handled as rewrite rules and are processed using a variant of the Knuth-Bendix technique.

(xi) **Mizar.** The Mizar system devloped by Andrzej Trybulec at the University of Bialystok, see http://mizar.org/ shares many of the goals of the present system, which are reflected in the extensive series of articles published in the Journal of Formalized Mathematics which it has spawned, see http://mizar.org/JFM. Fourteen volumes of the Journal of Formalized Mathematics have appeared.

A catalog listing many currently available proof verifier systems is available from Michael Kohlhase and Carolyn Talcott of Stanford University. See also the ORA bibliography page, and John Rushby's bibliograohy. Here are a few of its more interesting specific references: PVS.

Chapter 4. Undecidability and Unsolvability

For completeness sake and to enjoy the intellectual insight that these results provide, we derive several of the main classical results on undecidability and unsolvability in this chapter.

4.1. Chaitin's Theorem

Some of the most famous results concerning undecidability and unsolvability are easy to prove using an elegant line of argument due to Gregory Chaitin. Define the *information content* I(s) of a binary sequence s as the length (measured, like s, in bits) of the shortest program P which prints s and then stops. P should be written in some agreed-upon programming language L. We will see below that changing L to some other language L' leaves I(s) unchanged except for addition of a

quantity bounded by a constant C(L,L') depending only on the languages L and L'. Thus, asymptotically speaking, I(s) is independent of L.

Let |s| designate the length of the binary sequence s. Then, since s can always be printed by the program 'print(s)' (in which s appears as an explicit constant), it is clear that I(s) must be bounded above by |s| + C, where C depends only on the programming language L being used. Of course, this upper bound is sometimes far too large, since there are sequences s whose information content is much less than their length. For example, the information content of the decimal sequence consisting of the digit 1 followed by one trillion trillion zeros is not much larger than that of its defining expression $10^{10^{24}}$, whose binary form is only a few dozen bits long. On the other hand, a simple counting argument shows that most sequences of length n must have an information content close to n. Indeed, the number of programs representable by binary sequences of length at most n - c is less than $2^{n - c + 1}$, and not all of these programs print anything or stop, so the number of binary sequences of information content at most n - c (i.e. the set of outputs of all these programs) is less than $2^{n - c + 1}$. But, since the number of sequences of length n is 2^n, it follows immediately that the fraction of these sequences having information content no more than n - c is at most $2^{- c + 1}$. For c large enough this fraction will be very small, so most binary sequences of length n must have a larger information content.

Chaitin's theorem can now be stated as follows.

Theorem: If A is any consistent set of axioms for mathematics, then there is a constant c = c(A), depending only on A (and, indeed, only on the information content of A), such that no statement of the form I(s) > c can be proved using only the axioms A.

Proof: The proof is deliciously simple. For any constant k, let P(k) be the program which

1. Generates all possible sequences of formulae, in order of increasing length.
2. Checks these sequences to verify that their component formulae are syntactically well-formed and that each formula in the sequence follows directly (in terms of the rules of logical inference available) from the formulae which precede it. Sequences not having this property should immediately be dropped, and P should go on to examine the next sequence in turn.

   (As we have already emphasized, it is inherent in the very definition of formal logic that there must exist procedures for testing the well-formedness of formulae, and for determining whether one formula is an immediate consequence of others, since otherwise the logical system used would not meet Leibniz' fundamental criterion that arguments in it must be 'safe and really analytic').
3. Checks the final formula in each surviving sequence (this is the 'theorem proved') to determine whether it has the form 'I(s) > k'. If so, it prints s and stops. If not, it goes on to examine the next sequence in turn.

Observe that the length of the program P(k) equals L + log k, for a suitable constant L, if we assume that a binary encoding of k occurs in P(k). Let c be any constant such that c > L + log c.

If there exists any proof of a statement of the form 'I(s) > c' then plainly the procedure P(c) will eventually find a statement of this form, along with its proof. But then our program, whose length is less than c, prints a sequence s whose information content is provably greater than c, so that s cannot be printed by any program of length at most c. Our logical system is therefore inconsistent, contrary to assumption. **QED**.

The following variant of Chaitin's theorem can be proved in much the same way.

Theorem: There exists no program R which can determine the information content I(s) of an arbitrary binary sequence s.

Proof: Suppose that R exists, and write the program P which

1. generates all binary sequences s, in order of increasing length;
2. uses R to determine their information content;
3. stops when this content is seen to be large, say one million, and prints the sequence s; otherwise continues.

Since there are sequences of information content at least one million, P will eventually find one such and print it. But then this sequence is printed by the program P that we have just described, whose length is clearly much less than one million bits. Hence we have a contradiction, proving that R cannot exist. **QED**.

To see that the information content I(s) of a binary sequence varies only slightly when the programming language L used to define it is changed, we simply argue as follows. Programs written in any language L can be compiled to run on any adequate hardware system S. The size of the compiler required depends only on L, and so can be written as c(L). The instructions of S can be simulated in any other reasonable programming language L', and the size of the interpreter required for this depends only on L' and can therefore be written as c'(L'). This gives us a way of transforming any program P of length k and written in the language L into a program P' of length k + c(L) + c'(L') written in the language L' which produces the same results. Note also that P' eventually halts if and only if P does. Hence the minimum-length program in L' for producing s is of length no greater than k + c(L) + c'(L'). Since this same argument applies in the reverse direction, it follows that |I(s) - I'(s)| is bounded above by a constant

Undecidability results derivable from Chaitin's Theorem

It is now easy to derive the following results, some directly from Chaitin's theorem and the variant of it which we have stated, others by adapting Chaitin's line of argument.

(1) **Theorem** (Existence of undecidable statements): Let A be any consistent set of axioms for mathematics. Then there exists a mathematical formula F which such that neither F nor its negation (not F) can be proved using only the axioms A.

Proof: Consider the set of all binary sequences s of length c + k whose information content I(s) exceeds c, where c is the constant c appearing in Chaitin's theorem and k will be specified below. We know by Chaitin's theorem that none of the formulae I(s) > c involving these sequences s can be proved (even though all are true). Consider the set of such sequences s for which 'not (I(s) > c)' can be proved (several such proofs may be possible without inconsistency, even though all these statements are false). There can be at most 2^{c+1} such sequences, since we can prove (and indeed, have proved) that the total number of sequences of information content less than c is at most 2^c. For all the others, i.e. for all but a fraction 2^{-k} of statements of the form I(s) > c, neither the statement nor its negative is provable. So all these statements are undecidable in terms of the axioms A. **QED**.

(2) **Theorem** (Unsolvability of the halting problem [Turing's Theorem]): There exists no procedure R which, given the text of a program P, determines whether P eventually halts.

Proof: Let s be a binary sequence. Set up the program P which

1. generates all programs Q of length up to the length |s| of s;
2. uses R to determine whether Q eventually halts, and if not immediately eliminates Q;
3. progressively increments an integer number_of_steps, and then runs each of the remaining programs Q (i.e., under simulation) for number_of_steps, determining whether it has stopped or not, and if so whether it has printed s;
4. stops immediately once a program which prints s is found; otherwise continues;
5. stops once all the programs Q to be examined have halted.

It is clear from the description of this program that it will eventually halt (since it simulates only a finite number of programs, each of which eventually halts). When P halts it will have determined the information content of s (possibly by showing that this is at least the length of s). But, by the variant of Chaitin's theorem proved above, this is impossible. **QED**.

(3) **Theorem** (Nonexistence of a decision algorithm for elementary arithmetic): There exists no procedure R which, given a (quantified) formula of elementary arithmetic, determines whether or not P is true.

Proof: Since programs written in any programming language can be compiled to run (in assembly language) on any adequate hardware system, it follows from Turing's Theorem that there exists no procedure which, given some adequate computer system S, can determine whether an arbitrary assembly-language program for S stops. We take S to be a system easily modeled using arithmetic operations only. Specifically, we model the memory M of S as a large positive integer divided into W-bit 'words' (so that the j-th word of M is extracted by the operation

$$(M / 2^{jW}) \bmod 2^{W}.$$

Each of the registers of S, including its 'instruction location counter' ILC, is then modeled by an additional W-bit integer, which we can store at fixed low addresses in the memory integer M. To simulate one cycle of S's operation, we simply extract the instruction word addressed by ILC using the formula just displayed, use a similar formula to extract the memory words this instruction involves, and calculate the instruction result, which can always be expressed as a Boolean, and hence algebraic, combination of the registers it involves. The next value of ILC can be calculated in the same way for the same reason. To store a W-bit word X into memory location j, we simply change the integer M into

$$(M - (M \bmod 2^{jW})) + X * 2^{(j-1)W} + (M \bmod 2^{(j-1)W})).$$

This makes it plain that the effect of any individual operation of S can be expressed as an elementary operation on the integers used to represent the states of S. Hence, if the state of S on its j-th cycle is M, then the state of S on its j + 1'st cycle will be F(M), where F is some elementary arithmetic operation whose details reflect the architectural details of S.

Now suppose that the memory of S is initialized to M_0, and start S running. It will eventually halt iff there exists a sequence of M_i integers satisfying the quantified but otherwise elementary arithmetic formula

$$(i = 0 \text{ *imp } M_i = M_0) \ \& \ (\text{FORALL } i \mid M_{i+1} = F(M_i)) \ \& \ (\text{EXISTS } j \mid H(M_j)),$$

where H(M) is the elementary arithmetic predicate which expresses the condition that the operation executed when S is in state M is the 'Halt' instruction. So, if there existed an algorithm which could decide the truth of all formulae of the kind just displayed, we could use it to determine whether an arbitrary program P eventually halts, contradicting Turing's theorem. **QED**.

(4) **Theorem** (Nonexistence of a decision algorithm for predicate calculus [Church's Theorem]): There exists no procedure R which, given a (quantified) sentence of pure predicate calculus, determines whether or not P is valid, i.e. true irrespective of the meanings assigned to the constants and function symbols which appear in it.

Proof: We can encode the integers (in 'monadic' notation) as

$$0, \text{Succ}(0), \text{Succ}(\text{Succ}(0)), \text{Succ}(\text{Succ}(\text{Succ}(0))),...$$

In this universe of data objects, every integer except 0 has a predecessor Pred(n) such that n = Succ(Pred(n)). We can then express all other arithmetic functions recursively starting only with the constant '0' and the function symbols 'Succ' and 'Pred', e.g. as

```
function plus(n,m) {return if m = 0 then n
  else Succ(plus(n,Pred(m))) end if}

function times(n,m) {return if m = 0 then 0
  else plus(times(n,Pred(m)),n) end if}
```

```
function exp(n,m) {return if m = 0 then Succ(0)
  else times(exp(n,Pred(m)),n) end if}

function minus(n,m) {return if n = 0 then 0 elseif m = 0 then n
  else minus(Pred(n),Pred(m)) end if}

function gt(n,m) {return minus(n,m) /= 0}

function len_le(n,m) {return gt(exp(Succ(Succ(0)),m),n)}

function div(n,m) {return if m = 0 or gt(m,n) then 0
  else Succ(div(minus(n,m),m))}

function rem(n,m) {return minus(n,times(m,div(n,m))}
```

Continuing in the same way, we can build up a recursive function stops_and_outputs(P,m,s) which is true iff the program P (written in the assembly language of the abstract computer which appears in the proof of the immediately preceding theorem [Nonexistence of a decision algorithm for elementary arithmetic]) halts after m steps, having then produced the output s.

Such recursive functions can readily be mirrored in predicate calculus, e.g. by the quantified predicate statements

(FORALL n | (n = 0 or Succ(Pred(n)) = n))

(FORALL n | Succ(n) /= 0) & (FORALL n,m | Succ(n) = Succ(m) *imp n = m).

(FORALL n,m | Plus(n,0) = n & Plus(n,Succ(m)) = Succ(Plus(n,m)))

(FORALL n,m | Times(n,0) = 0 & Times(n,Succ(m)) = Plus(Times(n,m),n))

(FORALL n,m | Exp(n,0) = Succ(0) & Exp(n,Succ(m)) = Times(Exp(n,m),n))

(FORALL n,m | Minus(0,m) = 0 & Minus(n,0) = n & Minus(Succ(n),Succ(m)) = Minus(n,m))

(FORALL n,m | Gt(n,m) *eq Minus(n,m) /= 0)

(FORALL n,m | Len_le(n,m) *eq gt(Exp(Succ(Succ(0)),m),n))

(FORALL n,m | (Gt(n,m) *imp Div(n,m) = 0) & ((not Gt(n,m)) *imp Div(n,m) = Succ(Div(Minus(n,m),m))))

(FORALL n,m | Rem(n,m) = Minus(n,Times(m,Div(n,m))))

and so on, up to the point at which the function stops_and_outputs(P,m,s) is mirrored by a similar predicate formula. Since predicate substitution of formulae for variables, followed by simplification, generalizes the process of recursive evaluation of the procedures listed above, each recursive evaluation translates immediately into a predicate proof, so that whenever one of our functions, e.g. stops_and_outputs(P_0,m_0,s_0) evaluates to true for given constant values P_0,m_0,s_0 there will exist a predicate calculus proof of the statement stops_and_outputs(P_0,m_0,s_0). (This observation appears in Section ... as the 'Mirroring Lemma')

Now choose any sufficiently large integer k, and consider the predicate statement

```
(EXISTS P,n | Stops_and_outputs(P,n,s) & Len_le(P,k)       & (not Len_le(s,Times(Succ(Succ(0)),k)))).
```

Call this formula **F**. It simply states that the length of s is at least 2k and that the information content of s is no more than half its length k = (2k)/2. We have seen at the start of the present section that **F** can only be true for a small minority N of all sufficiently long sequences. This fact was established by an entirely elementary counting argument, readily translatable into predicate calculus terms.

It follows that, given any sufficient large k, the formula **F** can only be proved for a small minority of the sequences s_0 of length k. Indeed, if **F** could be proved for too many individual sequences s_0, the count N would be exceeded, and so the Peano axioms of elementary arithmetic would be self-contradictory within predicate calculus. Hence for any sufficiently large k there will exist many s for which **F** = **F**(s) is not provable.

Now suppose that a procedure R for deciding the provability of predicate-calculus formulae exists. Using R, construct the program which

1. examines all programs P, and sequences s of length at least k, in order of increasing total length;
2. uses R to determine whether **F**(s) is provable;
3. stops as soon as it finds an s such that **F**(s) is not provable, and prints s.

Since we have seen that there must exist many s such that **F**(s) is not provable, this procedure must eventually stop and print some such s. The s which is printed must have complexity at least k. Indeed, if this were false, there would exist a program P_0 of length less than k which stopped after some finite number n_0 of steps and printed s. Hence the value of the recursive functions stops_and_outputs(P_0,n_0,s), and len_le(P_0,k) would be true, implying, as we have seen above, that

$$\text{Stops\_and\_outputs}(P_0,n_0,s) \;\&\; \text{Len\_le}(P_0,k)$$

is provable; but we have chosen an s for which this is false.

Hence s has complexity at least k. But it is the output of the short program listed above. This is a contradiction for all sufficiently large k. Hence R cannot exist, and Church's theorem follows. **QED**.

4.2. The two Gödel Theorems

Next we turn to the proof of Gödel's two famous theorems. These rest upon the construction of a trick proposition G which asserts its own unprovability, and which therefore can be regarded as a technically precise rendering of the ancient paradoxical sentence 'This sentence is false'. Note that this sentence is troublesome for any system of formalized discourse in which it or anything like it can be given meaning, since it plainly can neither be true nor false.

Gödel's first theorem (in the improved form given to it by Rosser) asserts that (if we assume that the logical system in which G is being considered is consistent), then neither G nor its negative can be proved; hence G must be undecidable. (This is no longer so surprising as it was when first discovered by Gödel, since theorems like Chaitin's show the existence of large classes of undecidable statements). Gödel's second theorem uses much the same statement G as an auxiliary to prove that the logical theory containing G cannot be used to prove its own consistency.

All of Gödel's reasonings will become easy once we have clarified the foundations on which they rest. Since the line of argument used is somewhat more delicate than those needed in the preceding sections of this chapter, we begin with a more careful discussion of technical foundations than was given above. This more detailed discussion continues to emphasize the basic role of set theory. Note that the preparatory considerations which follow fall naturally into two parts, a first 'programming part' which is followed by a short discussion of the relationship of 'programming' to 'proof'.

Programming considerations

The mechanism of computation with hereditarily finite sets discussed earlier can easily be used to define strings and such basic operations on them as concatenation, slicing, and substring location. To this end, we use the definition of 'sequence of elements of a set s' given previously. We apply this to define the notion of 'a sequence of decimal digits', and of the integer value that such a sequence represents. This is done as follows:

```
function two(); return {{},{{}}}; end two;  function four(); return sum(two(),two()); end four;  functior
```

The standard abbreviations 0,1,2,3,4,5,6,7,8,9 for the ten members of ten() can now be introduced:

```
0 = {}, 1 = next(0), 2 = next(1), 3 = next(2), 4 = next(3),          5 = next(4), ...
```

along with the convention that a sequence $d_0d_1..d_n$ of such digit characters designates the decimal_value_of the decimal sequence

```
{ordered_pair(0,d0),ordered_pair(1,d1),...,ordered_pair(n,dn)}.
```

We also adopt the 'ASCII' convention that a character is simply an integer less than 256, and a string is simply a sequence of characters. That is,

```
function is_string(s); return is_sequence_of(s,256);     end is_string;
```

We also adopt the standard manner of writing strings within double quotes and the convention that a quoted sequence "$d_0d_1..d_n$" of characters designates the sequence

```
{ordered_pair(0,d0),ordered_pair(1,d1),...,ordered_pair(n,dn)}.
```

For example, "Abba" designates the sequence

```
{ordered_pair(0,65),ordered_pair(1,98),ordered_pair(2,98),        ordered_pair(3,97)}.
```

The string concatenation, slicing, and substring location functions now have the following forms.

```
function shift(s,n);     return if s = {} then {}       else shift(s less arb(s),n) with         ordered_
```

The next two functions respectively define the result of appending an additional component to each of the elements of a set s of sequences, and the collection of all ordered subsequences of a sequence s.

```
function append_to_elements(s,x);    return if s = {} then {}       else append_to_elements(s less arb(s)
```

It should be clear that, having arrived at this point, we can go on to define any of the more advanced string manipulation functions familiar from the computer-science literature, including functions which test a string for well-formedness according to any reasonable grammar, functions which detect and list the free variables of predicate and set-theoretic formulae, and functions which substitute specified terms for these free variables.

One such function that is needed below is that which tests an arbitrary hereditarily finite set to determine whether it is a sequence of strings. This is

```
function is_string_sequence(s);     return if s = {} then true      elseif not is_string(last_of(s)) then
```

We will not carry all of this out in detail, but simply note that full programming details very close to those alluded to here

form part of the code libraries which implement the verifier system discussed later in this book. (These are written in the SETL language, which is very close to the more restricted set-theoretic language considered above).

One other simple but more specialized string-manipulation function, which we call subst(s_1,s_2), will be used below. This is defined in the following way. Unless s_1 is a syntactically well-formed string of our language, subst(s_1,s_2) is the empty string {}. Otherwise the 'subst' operator finds the first free variable in s_1 and replaces every occurrence of this variable by an occurrence of the string s_2. For example,

```
subst("(FORALL z | F(x,g(x,y),z))","Abba") =     "(FORALL z | F(Abba,g(Abba,y)))"
```

but

```
subst("(FORALL z | F(x,g(x,y),z)","Abba") = {}
```

since its first argument string is syntactically ill-formed.

Note finally that any other universal form of computation, for example computation with strings or computation with integers, can substitute for the style of set-theoretic computation we have outlined. This follows from the fact that our set theoretic computations can be programmed to run on any standard computer, simply by encoding all hereditarily finite sets by bitstrings in any way that supports the primitive operations listed above and the simple style of recursion that we have assumed. It is even easier to program all the above set theoretic operations in a string language. To program them in pure arithmetic, one can simply regard strings as integers written to base 256.

Programming and proof; 'mirroring' programmable set-theoretic functions

A sequence of strings, every one of which is a syntactically legal formula of the language of logic, is a *proof* if every string in it is either an axiom or is derived via some allowed rule of inference from some finite subcollection of strings, each of which appears earlier in the sequence.

This definition can be applied in very general settings. We need not insist that the axioms allowed form a finite collection, but only that it must be possible to program the function

```
is_axiom(s)
```

which tests statements s to see if they are axioms. Similarly, we need not insist on any particular form for the rules of inference, but must only demand that we can program the function

```
last_is_consequence(s)
```

which tests a finite sequence of statements to verify that the last component s(prev(#s)) of the sequence is a valid immediate consequence of the formulae which precede it in s. We insist that 'last_is_consequence' (and 'is_axiom') must be programmable in order to prevent the acceptability of a proof from being a matter of debate. As a matter of convenience we assume that

```
is_axiom(x) *eq last_is_consequence({ordered_pair(0,x)})
```

Note that, given a procedure for testing 'last_is_consequence', the condition that a sequence of statements should be a proof is unambiguous, since this condition can be tested by calculating the value of the second function shown below.

```
function follows_from_element(list_of_subsequences,conclusion);    return if list_of_subsequences = {} th
```

A string is then a theorem if and only if it is the last element of some sequence of strings which is a proof.

The preceding definitions allow us to formulate the notion of 'logical system' in very general ways. But to relate a logical system to a computational system in the most useful way it is appropriate to impose a few additional conditions. First of all, we want each of the objects with which we will compute to have a representation in our logical system. The techniques described in our earlier discussion of computation with hereditarily finite sets can be used for this.To be sure that all such sets can be represented in our language, we can simply agree that some standard string representation of each such set must count as a syntactically well-formed term of our language, and that there should be a predicate Is_HF(s) which is true whenever s is the standardized string representation of such a set. Note that the condition that a string s is such a representation can easily be tested by a programmable function. Next, we agree that our system must include an equality predicate having all the customary properties, and must also include a collection of function symbols Singleton(s), Is_member(s_1,s_2), Arb(s), With(s_1,s_2), Less(s_1,s_2) having the indicated number of parameters. Moreover, every statement of a form like

```
Singleton(s1) = s2, Arb(s1) = s2, With(s1,s2) = s3,...
```

for which s_1, s_2, and s_3 are standard string representations of hereditarily finite sets and the corresponding Boolean value $\{s_1\} = s_2$, $arb(s_1) = s_2$, etc. is true must be an axiom (or theorem). There must also exist a predicate symbol In(s_1,s_2) of two variables for which the statement

```
In(s1,s2)
```

is an axiom or theorem whenever s_1 and s_2 are standard string representations of hereditarily finite sets and

```
s1 in s2
```

is true. Similarly, we require that there must exist a predicate symbol Incs(s_1,s_2) for which the statement

```
Incs(s1,s2)
```

is a theorem whenever s_1 and s_2 are standard string representations of hereditarily finite sets and

```
s1 incs s2
```

is true.

We also assume that the function Arb satisfies

```
(s = {} & Arb(s)= {}) or (In(Arb(s),s) &     (FORALL x | Is_HF(x) *imp (not (In(x,Arb(s)) & In(x,s))))),
```

whenever s is the standard representation of an hereditarily finite set, and that axioms are at hand allowing us to deduce such elementary set theoretic statements as

```
In(x,Singleton(y)) *eq x = y,   In(x,With(s,y)) *eq (In(x,s) or x = y),
```

etc. whenever the variables involved are standard string representations of hereditarily finite sets. Elementary set-theoretic facts of this kind will be used in what follows, where we will sometimes write predicates like In(x,s) in their more standard infix form.

We also wish to impose conditions that allow our logical system to imitate any function definition legal in the system for computation with hereditarily finite sets described earlier, and which ensure that any legal computation can be modeled by a corresponding proof. This can most conveniently be done as follows. We suppose our logical system to allow (1) variables, (2) predicate and function symbols of any number of arguments, which must be nestable to form expressions, (3) existential and universal quantifiers subject to the usual rules (so that our logical system must always include all the

standard mechanisms of the ordinary predicate calculus), and (4) conditional expressions formed using the keywords 'if...elseif...else...' in the usual way.

Recursive definition of new function and predicate symbols, for example in a style like

```
define Name(s1,s2,...,sn) := if cond1 then expn1      elseif cond2 then expn2      ...      elseif condm the
```

must also be possible under the same conditions in which the corresponding function definition would be allowed and would be certain to converge. (As for function definitions in a programming language, in such definitions the variables $s_1,s_2,...,s_n$ must all be distinct and the symbol 'Name' being defined must never have been used before). For function symbols such a definition must imply the universally quantified equality

```
(FORALL s1,s2,...,sn | Name(s1,s2,...,sn) = if cond1 then expn1      elseif cond2 then expn2      ...      e
```

and for predicate symbols the corresponding universally quantified logical equivalence. (More is said later about the situations in which such recursive definitions are legitimate, i.e. situations in which we can be sure, on essentially syntactic grounds, that the corresponding recursive functions would be certain to converge).

Provided that the conditions necessary for convergence discussed below are sytematically respected, any sequence of set- and Boolean-valued function definitions in our set-theoretic programming language can be 'mirrored' simply by translating each function definition

```
function name(s1,s2,...,sn);    return if cond1 then expn1      elseif cond2 then expn2      ...      elsei
```

into the definition

```
define Name(s1,s2,...,sn) := if cond1 then expn1      elseif cond2 then expn2      ...      elseif condm the
```

of a similarly-named logical symbol. For example, the definition

```
function union(s1,s2);     return if s1 = {} then s2       else union(s1 less arb(s1),s2) with arb(s1) end
```

of the function 'union' translates into the logical definition

```
define Union(s1,s2) := if s1 = {} then s2       else With(Union(Less(s1,Arb(s1)),s2),Arb(s1)) end if;
```

We will use the term 'mirroring', or more specifically 'mirroring in logic', for the systematic translation process just described. The 'mirrored' versions of the elementary set-theoretic functions appearing in our earlier discussion of computation with hereditarily finite sets will be written using the same names as the programmed functions, but the first letter of the name will be capitalized to indicate that it is a symbol of logic rather than a function name used in programming.

The elementary axioms and stipulations listed in the preceding paragraphs serve to ensure that all computations with hereditarily finite set described previously can be 'mirrored' by elementary logical proofs, in the manner described by our earlier 'Mirroring Lemma'. Note that the condition that a string should be any one of these required axioms (or theorems) can easily be tested by a programmable function. In addition to these required axioms (or some subset from which the rest of these required assertions can be proved), any number of other axioms are allowed.

We also want our logical system to include a principle of induction strong enough to subsume the ordinary principle of mathematical induction. Since integers are defined objects, rather than primitive objects, of our system, it is convenient to formulate our principle of induction in set-theoretic rather than integer-related terms. This can be done as follows. Since the sets spoken of in our logical system are all assumed to be finite (in fact, to be hereditarily finite), there can exist no

indefinitely long descending sequence of subsets of any of our sets. Hence any predicate which is true for the null set and true for a set s whenever it is true for all proper subsets of s must be true for all sets s. In formal terms this is

```
(P({}) &      (FORALL x | (Is_HF(x) & (FORALL y |        (Is_HF(y) & Incs(x,y) & x /= y) *imp P(y))) *i
```

Similarly, since the standard representation of any member of a set s must be shorter than that of s, there can be no indefinitely long sequence of sets, each of which is a member of the preceding set in the sequence. Hence any predicate which is true for the null set and true for a set s whenever it is true for all the members of s must be true for all sets s. In formal terms this second principle of induction (membership induction) is

```
(P({}) &      (FORALL x | (Is_HF(x) & (FORALL y |        (Is_HF(y) & y in x) *imp P(y))) *imp P(x)))
```

The first of these inductive principles (*subset induction*) is valid only for finite sets; the second (*membership induction*) carries over to the general set theory considered later in this book, which also allows infinite sets.

These statements are taken as axioms for every syntactically legal predicate formula P of our theory. Note that the condition that a formula should arise in this way, i.e. by substitution of a predicate formula for the symbol P appearing in the two preceding axiom templates, is computationally testable. Thus no incompatiblity arises with our general demand that there must always exist a programmed function which tests formulae to determine whether they are legal axioms.

Additional comments on the legitimacy of recursive definitions

Recursive definitions are legitimate in situations in which the corresponding function definitions are certain to converge. Consider a definition of the form

```
define Name(s1,s2,...,sn) := if cond1 then expn1      elseif cond2 then expn2      ...      elseif condm the
```

If this definition is not recursive, i.e. if all the only predicate and function symbols which appear in it have been defined previously, it is legitimate whenever it is syntactically legal. But if the defined function 'Name' appears within the body of the definition, then conditions must be imposed on the arguments of each such appearance for its legitimacy to be guaranteed. More specifically, consider any such appearance of 'Name', and suppose that it has the form Name(e_1,e_2,...,e_n). Then we must be able to prove that the sequence e_1,e_2,...,e_n is 'lexicographically smaller' than s_1,s_2,...,s_n, in the sense that there exists an integer k no larger than n such that

```
s1 incs e1,s2 incs e2,...,sk incs ek
```

can be proved in the context in which Name(e_1,e_2,...,e_n) appears, and that e_j /= s_j can also be proved for some j between 1 and k. These assertions must be proved under the hypothesis that all the conditions $cond_1$,$cond_2$,...,$cond_{h-1}$ appearing on if-statement branches preceding the branch containing the occurrence Name(e_1,e_2,...,e_n) are false, while $cond_h$ is true if Name(e_1,e_2,...,e_n) appears within $expn_h$.

As an example, consider the previously cited definition

```
define Union(s1,s2) := if s1 = {} then s2       else With(Union(Less(s1,Arb(s1)),s2),Arb(s1)) end if;
```

Since the function 'Union' being defined also appears on the right of the definition, this definition is recursive. To be sure that it is legitimate, we must show that the one appearance of 'Union' on the right, which is as the expression Union(Less(s_1,Arb(s_1)),s_2), has arguments certain to be lexicographically smaller than the initial arguments s_1,s_2, in the context in which Union(Less(s_1,Arb(s_1)) appears. This is so because we can show that its first argument Less(s_1,Arb(s_1)) is included in and not equal to s_1 in the context in which it appears. Indeed, in this context we can be certain that s_1 /= {}, so

```
s1 incs Less(s1,Arb(s1))
```

by elementary set theory, and since s_1 /= {}, Arb(s_1) is in s_1 but not in Less(s_1,Arb(s_1)), so

```
Less(s1,Arb(s1)) /=  s1.
```

Elementary arguments of this kind can be given in for each logical predicate and function-symbol definitions mirroring the programmed functions appearing in our earlier discussion of computation with hereditarily finite sets. We leave the work of verifying this to the reader. However, some of the arguments necessary appear in the following section on basic properties of integers.

Properties of Integers

To show that the inductive principles formulated above subsume ordinary integer induction, we can prove the few needed basic properties of integers as given by the set-theoretic encodings and definitions given earlier. For convenience, we list the recursive definitions corresponding to the relevant function definitions appearing in our earlier discussion of computation with hereditarily finite sets. These are

```
  define Next(s) := With(s,s);    define Last(s) :=       if s = {} then {} elseif s = Singleton(Arb(s))
```

We begin by showing that

```
(FORALL x | (Is_integer(x) *imp (x = {} or     (x = Next(Prev(x)) & Is_integer(Prev(x))))),
```

a statement which we will call the 'Next_Prev' lemma. To prove this, suppose that it is false. Then there exists an x such that

```
Is_integer(x) & (x /= {}) & (x /= Next(Prev(x))     or (not Is_integer(Prev(x)))).
```

On the other hand, by definition of 'Is_integer' we have

```
Is_integer(x) *eq if x = {} then true     else (x = Next(Prev(x)) & Is_integer(Prev(x))) end if.
```

It follows from the above that

```
(x = Next(Prev(x)) & Is_integer(Prev(x)) &     (x /= Next(Prev(x)) or (not Is_integer(Prev(x)))),
```

a contradiction proving the 'Next_Prev' lemma.

Another simple lemma of this kind needed below is what might be called the 'Prev_Next' lemma:

```
(FORALL x | Is_HF(x) *imp x = Prev(Next(x))).
```

By the definition of 'Next' this is equivalent to

```
(FORALL x | Is_HF(x) *imp x = Prev(With(x,x))).
```

Suppose that this is false, so that there exists a w such that Is_HF(w) & w /= Prev(With(w,w)).
By definition of 'Prev', and using the fact that With(x,x) /= {}, this means that

```
w /= Less(With(w,w),Last(With(w,w))),
```

so Last(With(w,w)) /= w. Hence our assertion will follow if we can prove that

```
(FORALL x | Is_HF(x) *imp x = Last(With(x,x))).
```

It is most convenient to prove this in the generalized form

```
(FORALL x,y | (Is_HF(x) & Is_HF(y)) *imp          (Incs(x,y) *imp (x = Last(With(y,x))))).
```

Suppose that this last statement is false. Then there exist u and v such that

```
Is_HF(u) & Is_HF(v) & Incs(u,v) & (u /= Last(With(v,u))).
```

Consider the predicate Q(x) defined by

```
Incs(u,x) *imp (u = Last(With(x,u))).
```

Applying the *subset induction principle* to Q, and noting that Q(v) is false gives

```
(not Q({})) or (not (FORALL x | (Is_HF(x) & (FORALL y |     (Is_HF(y) & Incs(x,y) & x /= y)) *imp Q(y))
```

If x = { }, then With(x,u) = Singleton(u), and therefore Last(With(x,u)) = u. This shows that Q({ }) is true, so the formula just displayed simplifies to

```
(not (FORALL x | (Is_HF(x) & (FORALL y |     (Is_HF(y) & Incs(x,y) & x /= y) *imp Q(y)))          *imp Q(
```

implying the existence of an x such that

```
(FORALL y | (Is_HF(y) & Incs(x,y) & x /= y & Incs(u,x)) *imp                      (u = Last(With(y,u))))
```

Using the definition of Last, and noting that With(x,u) /= { }, we have

```
Last(With(x,u)) =     if With(x,u) = Singleton(Arb(With(x,u))) then Arb(With(x,u))         else Last(Less
```

It follows that we cannot have With(x,u) = Singleton(Arb(With(x,u))), since u is in With(x,u) so this would imply that With(x,u) = Singleton(u), and so Last(With(x,u)) = u, contradicting u /= Last(With(x,u)). Hence x /= { } and also

```
Last(With(x,u)) = Last(Less(With(x,u),Arb(With(x,u)))).
```

Since With(x,u) /= { } we must have

```
Arb(With(x,u)) in With(x,u),
```

so either

```
Arb(With(x,u)) = u or Arb(With(x,u)) in x.
```

But Arb(With(x,u)) = u is impossible, since if y is any element of x, then, since Incs(u,x), y would also be an element of u, contradicting

```
Intersection(Arb(With(x,u)),With(x,u)) = {}.
```

It follows that we must have

```
Arb(With(x,u)) in x,
```

from which it follows that

```
Less(With(x,u),Arb(With(x,u))) =     With(Less(x,Arb(With(x,u))),u).
```

Then plainly

```
Incs(x, Less(x,Arb(With(x,u)))) & x /= Less(x,Arb(With(x,u))),
```

so from

```
(FORALL y | (Incs(x,y) & x /= y & Incs(u,x)) *imp     (u = Last(With(y,u))))
```

we have u = Last(With(Less(x,Arb(With(x,u))),u)), and therefore u = Last(With(x,u)), a contradiction completing our proof of the 'Prev_Next' lemma.

For what follows we also need the lemma

```
(FORALL x | Is_HF(x) *imp (x = {} or Last(x) in x)).
```

To prove this, suppose that it is false, so that there exists a u such that

```
Is_HF(u) & u /= {} & (not (Last(u) in u)).
```

Consider the predicate Q(x) defined by x = { } or (Last(x) in x). Applying the subset induction principle to Q, and noting that Q(u) is false gives

```
(not Q({})) or (not (FORALL x | (Is_HF(x) & (FORALL y |     (Is_HF(y) & Incs(x,y) & x /= y) *imp
```

so that there exists an x for which

```
(not Q({})) or ((FORALL y | (Is_HF(y) &         Incs(x,y) & x /= y) *imp Q(y)) & Is_HF(x) & (not Q(x))).
```

Since Q({ }) is true, this simplifies to

```
(FORALL y | (Is_HF(y) & Incs(x,y) & x /= y)         *imp Q(y)) & Is_HF(x) & (not Q(x)),
```

where plainly x /= { }. That is,

```
(FORALL y | (Is_HF(y) & Incs(x,y) & x /= y) *imp     (y = {} or (Last(y) in y))) & Is_HF(x) & x /= {} &
```

using the definition of 'Last' we have

```
Last(x) = if x = {} then {} elseif     if x = Singleton(Arb(x)) then Arb(x)         else Last(Less(x,Arb(
```

Since x /= { } the first of the cases appearing in this last formula is ruled out, and since it then follows that

```
Arb(x) in x
```

the second of these cases is excluded also. Hence we have Last(x) = Last(Less(x,Arb(x))). But then we also have

```
Incs(x, Less(x,Arb(x))) & Less(x,Arb(x)) /= x,
```

and so it follows from

```
(FORALL y | (Is_HF(y) & Incs(x,y) & x /= y) *imp (y = {} or     (Last(y) in y))).
```

that

```
Last(Less(x,Arb(x))) in Less(x,Arb(x)).
```

Since Incs(x, Less(x,Arb(x))) and Last(Less(x,Arb(x))) = Last(x), we see that Last(x) in x, completing our proof of the statement

```
(FORALL x | x = {} or Last(x) in x).
```

Peano's principle of mathematical induction. Now we prove that Peano's standard principle of mathematical induction applies to integers as we have defined them. This is done by showing that for every expression P with one free variable we have

```
P({}) & (FORALL y | (Is_integer(y) & P(y)) *imp P(Next(y)))     *imp (FORALL x | Is_integer(x) *imp P(x))
```

To prove this statement, suppose that it is false, so that there exists an x such that

```
P({}) & (FORALL y | (Is_integer(y) & P(y)) *imp P(Next(y))) &     Is_integer(x) & (not P(x)).
```

Consider the predicate expression Q(y) defined by 'Is_integer(y) *imp P(y)'. Since (FORALL z | Q(z)) is false (for z = x in particular), an application of the subset induction principle to Q gives

```
(not Q({})) or (not (FORALL x | (Is_HF(x) & (FORALL y |        (Is_HF(y) & Incs(x,y) & x /= y) *imp Q(y))
```

so that there exists an x such that

```
(not Q({})) or      ((FORALL y | (Is_HF(y) & Incs(x,y) & x /= y) *imp Q(y))           & Is_HF(x) & (not Q
```

that is

```
(Is_integer({}) & not P({})) or     (FORALL y | (Is_HF(y) & Incs(x,y) & x /= y) *imp          (Is_integer
```

Since P({}) must be true and Is_integer(y) implies Is_HF(y), this simplifies to

```
(FORALL y | (Incs(x,y) & x /= y & Is_integer(y)) *imp P(y))    & Is_integer(x) & (not P(x)).
```

Since P({}) is true, we have x /= {}, so by the Next_Prev lemma we have

```
(FORALL y | (Incs(x,y) & x /= y &     Is_integer(y)) *imp P(y)) & Is_integer(x) &     Is_integer(Prev(x))
```

Thus, since (FORALL y | (Is_integer(y) & P(y)) *imp P(Next(y))), we must have

```
(FORALL y | (Incs(x,y) & x /= y & Is_integer(y)) *imp P(y))     & Is_integer(Prev(x)) & (not P(Prev(x))))
```

Using the definition of 'Prev' we have

```
Prev(x) = if x = {} then {} else Less(x, Last(x)) end if,
```

and so, since x /= {}, we have Prev(x) = Less(x, Last(x)). But now, since it was proved above that Last(x) is in x whenever x /= {}, it follows that

```
Incs(x, Prev(x)) & x /= Prev(x) & Is_integer(Prev(x)).
```

But then

```
(FORALL y | (Incs(x,y) & x /= y & Is_integer(y)) *imp P(y))
```

implies that P(Prev(x)), contradicting (not P(Prev(x))), and so completing our proof of Peano's standard axiom of induction.

To complete our discussion it is worth proving the remaining Peano axioms for integers, which in our set-theoretic formulation are

```
 (i) Is_integer({});   (ii) (FORALL x | Is_integer(x) *imp Is_integer(Next(x)));   (iii) (FORALL x | Next
```

The first statements follows from the definition

```
(FORALL x | Is_integer(x) *eq if x = {} then true     else x = Next(Prev(x)) & Is_integer(x) end if).
```

Since by definition Next(x) = x with x, Next(x) /= { } follows by elementary set-theoretic reasoning.

Finally, since it has been shown above that Is_integer(x) implies x = Prev(Next(x)), statement (iv) follows immediately.

This completes our discussion of the relationship between standard integer induction and the set-theoretic induction principles stated above.

The recursive proofs given in the preceding pages are intended to typify the large but generally straightforward family of proofs needed to show that the elementary functions of sequences, lists of sequences, strings, etc. defined above necessarily have their familiar properties. For example, it can be shown in this way that string concatenation is associative, i.e. that

```
(FORALL x1,x2,x3 |     (Is_string(x1) & Is_string(x2) & Is_string(x3)) *imp      (Concatenate(Concatenate(x
```

Or, as another example, we can show that the same final string is obtained by first deleting the final character of a (non-empty) string x_2 and then appending the result to a string x_1 as would be obtained by first appending the two strings and then deleting the final character of what results. In formal terms this is

```
(FORALL x1,x2 | (Is_string(x1) & Is_string(x2)) *imp    Concatenate(x1,Less(x2,Ordered_pair(Prev(Card(x2)),
```

Since very many such proofs will be given later in this book in full, computer-verified detail (albeit in a somewhat different setting, viz general set theory rather than the more limited theory of hereditarily finite sets on which the present section concentrates), we prove no theorems of the kind illustrated by our two examples, other than those already proved above. Instead, we content ourselves with the broad claim that all the results of this elementary kind of whose correctness one could convince oneself after careful examination can also be proved formally. To attain conviction that this is so, the reader may wish to try his/her hand at a few such proofs, for example the proofs of the two examples just given. The line of reasoning found below depends on a few theorems of this kind, of which the statement

```
(FORALL x1,x2 | (Is_proof(x1) & Is_proof(x2)) *imp         Is_proof(Concatenate(x1,x2))),
```

where 'Is_proof' is the logical symbol mirroring the recursive function 'is_proof', is typical. Computer verification of the line of reasoning given below would require systematic, computer verified proof of a finite collection of such statements.

A final remark on proof and computation

The mirroring lemma shows that our logical system 'envelops' the process of computation with hereditarily finite sets, in the sense that any value name(c_1,c_2,...,c_n) derivable by computation can also be derived in another way, namely by proving a theorem of the form Name(e_1,e_2,...,e_n) = e_{n+1}. In cases favorable for proof the length of the proof required may be considerably shorter than the computation it replaces, even allowing for the great difference in speed between human thought and electronic computation. For example, we can easily prove that

```
Exp(10,40) = 10000000000000000000000000000000000000000,
```

supplying the required proof in a time much shorter than that needed to verify this same fact by direct computation. Similarly, given two functions f(s_1,s_2,...,s_n) and g(s_1,s_2,...,s_n) and the symbols F and G which mirror them, we may be able to prove a theorem of the form

```
(FORALL s1,s2,...,sn | F(s1,s2,...,sn) = G(s1,s2,...,sn)).
```

thus allowing replacement of f by g, whose computation may be much faster.

More generally, the mirroring lemma gives us the following result. Suppose that some logical system in which we are able to embed our computational system is consistent, and let the logical symbol 'Fcn' mirror some recursively defined function 'fcn'. Then, given representations of two hereditarily finite sets s_1 and s_2, there exists a proof of the logical statement 'Fcn(s_1) = s_2' if and only if fcn(s_1) evaluates to s_2. Indeed, the mirroring lemma shows that 'Fcn(s_1) = s_2' must be a theorem if fcn(s_1) evaluates to s_2. Conversely, if our logical system is consistent, there can exist at most one s_2 for which 'Fcn(s_1) = s_2' is provable, and so by the mirroring lemma the one s_2 for which this is provable must be the value of fcn(s_1) derivable by computation.

A technical adjustment

To avoid a technical issue that would otherwise arise it is convenient to formulate the 'follows_from_element' function defined at the start of Section ... ('Programming and Proof') in a slightly different way. To this end, we first define the auxiliary function

```
function ax(x); return if is_axiom(x) then x else "true" end if; end ax;
```

Then is_axiom(ax(x)) is always true, and in fact the expressions is_axiom(x) and x = ax(x) are always equal.

We also introduce the following function

```
function lic(x);     return if is_sequence(x) & last_is_consequence(x) then x         else ordered_pair(0
```

so that

```
last_is_consequence(lic(x))
```

is true for every x.

This enables us to write the follows_from_element in the following modified way, which will be more convenient in what follows:

```
function follows_from_element(list_of_subsequences,conclusion);  return if list_of_subsequences = {}     th
```

It is convenient to assume that there exists some reasonably small integer k_0 such that every sequence s for which last_is_consequence(s) is true has length at most k_0. All common logic systems satisfy this condition (generally with k_0 not much larger than 4), which in any case is easily replaced if necessary. At any rate, assuming this condition, we may as well assume that every sequence x for which last_is_consequence(x) is true is of length exactly k_0, since shorter sequences of hypotheses can be left-padded with an appropriate number of copies of the trivial hypothesis 'true'.

The 'provability' predicate Pr(s)

Given the availability of existential and universal quantifiers within a logic system, we can at once define a predicate which states that the string s is provable. This is simply

```
define Pr(s) := (EXISTS p | Is_proof(p) & Last_of(p) = s),
```

which we can also write as

```
define Pr(s) := (EXISTS p | Is_proof_of(p,s))
```

if we introduce the intermediate definition

```
define Is_proof_of(p,s) := Is_proof(p) & Last_of(p) = s
```

which states that p is a proof culminating in the statement s.

This predicate differs from those previously considered in one important respect. All of these others correspond to recursive functions which converge to some definite set-theoretic value whenever they are applied to the appropriate number of hereditarily finite arguments. Pr is a more abstract existential, which if programmed would correspond to a search loop not guaranteed to converge.

We shall now establish several important properties of this predicate, for subsequent use. Note first that the concatenation Concatenate(p_1,p_2) of any two proofs is also a proof, since each element of the first part of the concatenation is either an axiom or a consequence of preceding statements, and similarly for the second part of the concatenation. Moreover, if p is a proof and we concatenate any formula which is a valid consequence of a subsequence of the formulae in p to p, then the result is also a proof.

Next note that given s_1,s_2, and s_3 for which s_3 has the form s_1 *imp s_2 (i.e. , s_3 = Concatenate(Concatenate(s_1," *imp "),s_2)) then

```
(Pr(s1) & Pr(s3)) *imp Pr(s2).
```

For if not we would have Pr(s_1) & Pr(s_3) & (not Pr(s_2)), and so using the definitions of Pr(s_1) and Pr(s_3) we could find p_1 and p_3 for which we have

```
Is_proof(p1) & Last_of(p1) = s1       & Is_proof(p3) & Last_of(p3) = s3
```

But then the concatenation of p_1, p_3, and the single formula s_2 is a proof of s_2, since s_2 follows by propositional implication from the two formulae s_1 and s_3.

We can write this result as the implication

```
Pr(Concatenate(Concatenate(s1," *imp "),s2)) *imp          (Pr(s1) *imp Pr(s2)),
```

or, in a more drastically abbreviated notation,

```
Pr(s1 *imp s2) *imp (Pr(s1) *imp Pr(s2)),
```

We can show in much the same way that given s_1, s_2, and s_3, if s_3 has the form s_1 & s_2 (i.e. , s_3 = Concatenate(Concatenate(s_1," & "),s_2)), then

```
(Pr(s1) & Pr(s2)) *imp Pr(s3).
```

This result can be written as the implication

```
(Pr(s1) & Pr(s2)) *imp     Pr(Concatenate(Concatenate(s1," & "),s2)),
```

or, again abbreviating more drastically, as

```
Pr(s1) & Pr(s2) *imp (Pr(s1 & s2)).
```

Next suppose that Q(x) is any expression involving one free variable x for which we have

```
Pr((FORALL y | Q(y))
```

Then Is_proof_of(p,(FORALL y | Q(y)) for some p. But then, for any x, the concatenation p' of p with the single statement Q(x) is also a proof, so

```
Is_proof_of(Concatenate(p,        Singleton(Ordered_pair(0,Q(x)))),Q(x))
```

where x denotes the standard representation of any hereditarily finite set. This shows that

```
Pr((FORALL y | Q(y)) *imp (FORALL x | Pr(Q(x)))
```

Finally, if s is an axiom, i.e. if Is_axiom(s) where 'Is_axiom' is the predicate symbol that mirrors the Boolean function 'is_axiom', then the one-element sequence {Ordered_pair({},s)} is easily shown to satisfy Is_proof({Ordered_pair({},s)}). Since we must also have Cdr(Arb({Ordered_pair({},s)})) = Cdr(Ordered_pair({},s)) = s, it follows that Is_proof_of({Ordered_pair({},s)},s). Conversely any proof of length 1 must have the form {Ordered_pair({},s)} where s is an axiom, i.e. satisfies Is_axiom(s).

Our next goal is to prove the following

Proof Visibility Lemma

Let s be any string representing a syntactically well-formed logical formula. Then

```
Pr(s) *imp Pr(Pr(s)).
```

In intuitive terms this simply states that any proof of s can be turned (rather explicitly) into a proof that s has a proof; i.e. the existence of a proof of s is always 'visible' rather than 'cryptic'. We prove this as follows. Assuming that it is false, there is an s such that Pr(s) & (not Pr(Pr(s))), and so there exists a sequence p of strings such that

```
Is_proof(p) & s = Last_of(p) & (not Pr(Pr(s))).
```

Hence, since Last_of(p) = Value_at(p,Prev(Card(p))), there exists an integer n such that

```
Is_proof(p) & n in Card(p) and (not Pr(Pr(Value_at(p,n))))
```

Let n be the smallest such integer. Then, by definition of Is_proof, either Value_at(p,n) = Ax(Value_at(p,n)), or there exists a finite sequence $n_1,n_2,...,n_k$ of integers, all smaller than n, such that

```
Last_is_consequence([Value_at(p,n1),        Value_at(p,n2),..,Value_at(p,n1),Value_at(p,n)])
```

which, as noted previously, means in more formal terms that there exists an x such that

```
Value_at(p,n1) = Value_at(Lic(x),0) & ... &     Value_at(p,nk) = Value_at(Lic(x),k - 1) & Value_at(p,n)
```

where the function symbol 'Lic' mirrors the function 'lic' introduced above. In the first of these cases (Value_at(p,n) = Ax(Value_at(p,n))) we reason as follows. For every x the sequence of formulae

```
Is_proof(Singleton(Ordered_pair(0,Ax(x))))  Is_proof_of(Singleton(Ordered_pair(0,Ax(x)),Ax(x)))  (EXISTS
```

is the skeleton of a proof whose intermediate details the reader should easily be able to fill in. If we denote this completed proof by

```
...  Is_proof(Singleton(Ordered_pair(0,Ax(x))))  ...  Is_proof_of(Singleton(Ordered_pair(0,Ax(x)),Ax(x)))
```

by cp, then cp is an explicit proof whose final statement is Pr(Ax(x)), allowing us to conclude that Pr(Pr(Ax(x))) for every x. Thus, if Axiom(s), so that s = Ax(s), we have Pr(Pr(s)). This proves that the implication

```
Axiom(s) *imp Pr(Pr(s))
```

holds for all s, so that if Axiom(Value_at(p,n)) we have Pr(Pr(Value_at(p,n))) , contradicting (not Pr(Pr(Value_at(p,n)))), and so ruling out the case Axiom(Value_at(p,n)).

Next suppose that there exists a finite sequence $n_1,n_2,...,n_{k_0 - 1}$ of integers, all smaller than n, such that

```
Last_is_consequence(Value_at(p,n1),Value_at(p,n2),..,        Value_at(p,nk0 - 1),Value_at(p,n))
```

(Here and in what follows, k_0 is the common length of sequences x for which Last_is_consequence(x) is true). Then by inductive hypothesis we have Pr(Pr(Value_at(p,n_j))) for each j from 1 to k_0 - 1. Also the sequence of formulae

```
Suppose: (EXISTS y | Pr(Value_at(Lic(y),0)) & ... &         Pr(Value_at(Lic(y),k0 - 2)) &          (not Pr
```

is the skeleton of a proof whose intermediate details the reader should again be able to fill in. This proof (when completed) shows explicitly that

```
 (FORALL y | (Pr(Value_at(Lic(y),0)) & ... &     Pr(Value_at(Lic(y),k0 - 2))) *imp         Pr(Value_at(Li
```

and so we have

```
 Pr((FORALL y | (Pr(Value_at(Lic(y),0)) & ... &     Pr(Value_at(Lic(y),k0 - 2))) *imp         Pr(Value_at
```

From this it follows, as noted above, that

```
 (FORALL x | Pr((Pr(Value_at(Lic(x),0)) & ... &     Pr(Value_at(Lic(x),k0 - 2))) *imp         Pr(Value_at
```

By definition of 'Last_is_consequence' we have

```
(EXISTS y | Value_at(p,n1) =     Value_at(Lic(y),0) & ... & Value_at(p,nk0) =         Value_at(Lic(y),k0 -
```

so

```
Value_at(p,n1) = Value_at(Lic(x0),0) & ...      & Value_at(p,nk0) =          Value_at(Lic(x0),k0 - 2) &
```

for some x_0. From this it follows, using the last universally quantified formula appearing above, that

```
Pr((Pr(Value_at(p,n1)) & ... & Pr(Value_at(p,nk)))      *imp Pr(Value_at(p,n)))
```

Hence, using the previously established implication (Pr(a *imp b) *imp (Pr(a) *imp Pr(b)), we have

```
Pr(Pr(Value_at(p,n1)) & ... & Pr(Value_at(p,nk)))      *imp Pr(Pr(Value_at(p,n)))
```

and then by repeated use of the implication Pr(a) & Pr(b) *imp Pr(a & b) established earlier it follows that

```
(Pr(Pr(Value_at(p,n1))) & ... & Pr(Pr(Value_at(p,nk0))))      *imp Pr(Pr(Value_at(p,n))).
```

Since Pr(Pr(Value_at(p,n_j))) for all j from 1 to k_0, it follows that Pr(Pr(Value_at(p,n)), completing our demonstration of the Proof Visibility Lemma.

Gödel's trick sentence

Gödel's trick sentence G is now simply

```
not Pr(Subst("not Pr(Subst(x,x)","not Pr(Subst(x,x))")
```

where 'Subst' is the logical symbol that mirrors the two-parameter string function 'subst' introduced above.

In this statement, and repeatedly in what follows, the quoted string "not Pr(Subst(x,x))" appears. It should be kept in mind that this is simply an abbreviation for the character sequence

110, 111, 116, 32, 80, 114, 40, 83, 117, 98, 115, 116, 40, 120, 44, 120, 41, 41,

that is, for the set constant

```
{ordered_pair(0,110),ordered_pair(1,111),ordered_pair(2,116),  ordered_pair(3,32),ordered_pair(4,80),orde
```

Note that since x is the only free variable of the syntactically well-formed string "not Pr(Subst(x,x))", the functional expression

subst("not Pr(Subst(x,x)","not Pr(Subst(x,x)")

evaluates to

"not Pr(Subst("not Pr(Subst(x,x)","not Pr(Subst(x,x)")"

and therefore the logical statement

```
Subst("not Pr(Subst(x,x)",      "not Pr(Subst(x,x)") = "not Pr(Subst("not Pr(Subst(x,x))",     "not Pr(Sub
```

which mirrors this evaluation is a theorem. Therefore so is

```
Pr(Subst("not Pr(Subst(x,x))",      "not Pr(Subst(x,x))")) *eq             Pr("not Pr(Subst("not Pr(Subst
```

Hence

```
(not Pr(Subst("not Pr(Subst(x,x))","not Pr(Subst(x,x))"))) *eq     (not Pr("not Pr(Subst("not Pr(Subst(x,
```

is a theorem. defining G as (not Pr(Subst("not Pr(Subst(x,x))","not Pr(Subst(x,x))"))), we see that

G *eq (not Pr("G"))

is also a theorem.

Rosser's variant of Gödel's trick sentence

This variant is obtained by replacing the predicate 'not Pr(s)' , i.e.

```
(FORALL p | not Is_proof_of(p,s))
```

by the modified predicate Prr(s) defined by

```
Prr(s) := (FORALL p | (not Is_proof_of(p,s)) or     (EXISTS q | Shorter(q,p) & Is_proof_of(q,Neg(s))))
```

Here 'Neg' is a function symbol which mirrors the operation which simply negates a string by prepending 'not' to it, and Shorter is a predicate symbol which mirrors the function shorter(q,p) that tests one sequence of strings (its first parameter q) to verify that it is shorter than its second parameter.

Note that whereas 'not Pr(s)' states that s has no proof, Prr(s) states that either s has no proof or that, if it does, there exists a shorter proof of the negation of s. In a consistent logical system these two conditions are the same, since the added clause (EXISTS q | Shorter(q,p) & Is_proof_of(q,Neg(s)) implies that the negation of s has a proof and hence implies that s can have no proof. Nevertheless this technical reformulation of the condition 'not Pr(s)' is advantageous for the argument given two paragraphs below.

Rosser's trick sentence is now

not Prr(Subst("Prr(Subst(x,x)","Prr(Subst(x,x))")

where 'Subst' is as before. Reasoning as above we find that since x is the only free variable of the syntactically well-formed string "Prr(Subst(x,x))", the functional expression

subst("Prr(Subst(x,x)","Prr(Subst(x,x)")

evaluates to

"Prr(Subst("Prr(Subst(x,x)","Prr(Subst(x,x)")"

and so the logical statement

```
Subst("Prr(Subst(x,x)",     "Prr(Subst(x,x)") = "Prr(Subst("Prr(Subst(x,x))",         "Prr(Subst(x,x))"))"
```

which mirrors this evaluation is a theorem. Therefore so is

```
Prr(Subst("Prr(Subst(x,x))",     "Prr(Subst(x,x))")) *eq Prr("Prr(Subst("Prr(Subst(x,x))",         "Prr(Sub
```

defining Gr as Prr(Subst("Prr(Subst(x,x))","Prr(Subst(x,x))"))), we see that

Gr *eq Prr("Gr")

is also a theorem.

Proof of Rosser's variant of Gödel's first theorem

Given that we have just exhibited a proof of the statement Gr *eq not Prr("Gr"), it is now easy to complete the proof that (if the proof system T in which we reason is consistent) neither Gr nor (not Gr) can be provable, showing that Gr is undecidable in T, which is Rosser's strengthened version of Gödel's first theorem.

For suppose first that Gr is provable. Then using Gr *eq Prr("Gr"), we can conclude Prr("Gr"), i.e. that

```
(FORALL p | (not Is_proof_of(p,"Gr")) or     (EXISTS q | Shorter(q,p) & Is_proof_of(q,Neg("Gr")))).
```

So either Gr is not provable, or there exists a proof of the negation of Gr. But if our logical system is consistent, this last implies that Gr is not provable in either case.

Suppose next that 'not Gr' is provable, and let q_0 be a proof of 'not Gr'. Using Gr *eq Prr("Gr") once more we can deduce that (not Prr("Gr")), i.e. that

```
(EXISTS p | Is_proof_of(p,"Gr") &     (FORALL q | (not Shorter(q,p)) or         (not Is_proof_of(q,Neg("G
```

Let p_0 be a proof satisfying this existential statement, so that

```
Is_proof_of(p0,Gr) & (not Shorter(q0,p0)).
```

It follows that the length of the proof p_0 is less than or equal to the length of q_0, so that we can find p_0 explicitly by searching the finite collection of proofs that are no longer than q_0. But then we have proofs of both Gr and the negation of Gr, and so a contradiction, which we have assumed to be impossible. **QED**.

Proof of Gödel's second theorem

This states that if our logical system is consistent it must be impossible to prove 'not Pr(false)' in it. To show this, let p_0 denote the proof of the theorem G *eq (not Pr("G")) derived above. The implication G *imp (not Pr("G")) follows by one additional step. Hence, if p is a proof of G, then the concatenation of p and p_0, plus two additional steps, gives a proof of 'not Pr("G")', showing that Pr("G") implies Pr("not Pr(G)"). Thus we have proved Pr("G") *imp Pr("not Pr(G)").

Now, if a statement and its negative can both be proved, then by concatenating these two proofs and adding one more step we obtain a proof of Pr(false). Hence if our logical system is consistent and its consistency, i.e. the statement 'not Pr(false)', can be proved within it, then the implication

```
Pr("not Pr(G)") *imp (not Pr("Pr(G)")),
```

and so Pr("G") *imp (not Pr("Pr(G)")), follows from Pr("G") *imp (not Pr("Pr(G)")). But the implication Pr("G") *imp Pr("Pr(G)") was proved in an earlier section (Proof Visibility lemma). This proves that (not Pr("G")), and since G *eq (not Pr("G")) we have a proof of G, whose existence immediately implies Pr("G"), a contradiction. Thus it must be impossible to prove 'not Pr(false)' within our logical system, as asserted. **QED**.

4.3. Axioms of Reflection

The large cardinal axioms discussed in Chapter 3 give one way of extending the axioms of set theory to increase their power, but as these stand they are of little direct interest for the work to be done in the following chapters, since their most immediate consequences are relatively specialized theorems in set theory which we will not prove. There is, however, a different (but, as we shall see, not entirely unrelated), class of axioms, the so called *axioms of reflection*, which can be added to the axioms of set theory and are of more direct practical interest. These are axioms of the form

```
    Pr('F') *imp F,
```

that is, statements which assert that if a formula F has a proof (a fact that we may, for example, be able to establish nonconstructively), then F follows. The potential practical importance of statements of this type is that they make the collection of proof mechanisms available to us indefinitely extensible, since we may be able to establish general theorems of the form

```
  (FORALL s | A(s) *imp Pr(B(s))),
```

where A and B have recursive definitions that allow them to be calculated mechanically, or at least established easily, for hereditarily finite sets s. Then, whenever a formula F is seen to satisfy 'F' = B(s) for some s satisfying A(s), we may be able to deduce Pr('F') easily or automatically, and then F immediately by an axiom of reflection.

However, Gödel's second theorem tells us that the additional axioms we desire must be set up carefully, since it is not immediately obvious that added axioms of reflection do not introduce contradictions. Let T be a logical system to which Gödel's second theorem applies. That theorem implies that we cannot expect to prove all statements of the form

```
(+) Prᴛ('F') *imp F
```

in any consistent logical system, and indeed there is a strengthening of Gödel's theorem, known as Lob's theorem, which shows that (+) can only be proved in T if F itself can be proved in T. Nevertheless, one can ask whether the addition of all the statements T to a consistent system produces a more powerful by still consistent system T', and in particular whether the collection T of statements can be modeled in some more powerful system T^* in which (+) can be proved. We shall show in the following pages that if T^* is the set ZFC of Zermelo-Fraenkel axioms of set theory extended by some assumption implying the existence of an inaccessible cardinal N, if H(N) is the universe for the model of ZFC considered in Chapter 3, and if T is the weakening of T' which only asserts the existence of those inaccessible cardinals smaller than N, then (+) can be proved in T^*, so that T^* implies the consistency of the system obtained from T by adding the desired axioms of reflection.

Several technical problems must be handled along the way to this goal. The first of these lies in the fact that the axioms we want to add involve the predicate Pr(s), which is substantially more composite than the ordinary axioms of set theory. To handle this we write a set of auxiliary axioms, whose consistency with the axioms of set theory is not at issue since with suitable definition of the symbols appearing in them they are all provable consequences of the ZFC axioms. These auxiliary axioms allow us to write a formula for the needed predicate Pr. To avoid the addition of too much clutter to set theory's streamlined basic axiom set, we will put these new axioms rather more succinctly than is done in our main series of definitions and proofs, but always in a provably equivalent way. The series of statements which we will now develop accomplishes this. We begin with

```
  (x in {a,b}) *eq (x = a or x = b)  (x in a + b) *eq (x in a or x in b)  [x,y] = {{x},{{x},{{y},y}}}     I
```

The above statements define the notions of integers n and sequences of length n. Our next aim is to define the notion of (ZFC) 'formula', which for succinctness we define as what would normally be called the syntax tree of a formula. We encode these trees as collections of nodes, each node being a sequence whose first component is an integer encoding the

node type and whose remaining components are the syntactic supbarts appropriate for the type of node. The allowed node types, and their distinguishing codes, are as follows: 0: variable or constant, 1: &-operator, 2: or-operator, 3: *imp-operator, 4: *eq-operator, 5: not-operator, 6: false, 7: true, 8: equality sign, 9: FORALL, 10: = EXISTS, 11: in, 12: atomic formula involving predicate, 13: term involving function symbol. Note that all of the 'encoded' forms of axioms which appear in the following discussion are written using only the elementary set-theoretic constructions {x}, {x,y}, [x,y], x + y, Seq2(x,y), which is defined as {[0,x],[1,y]}, Seq3(x,y,z), which is defined as Seq2(x,y) + {[2,z]}, and operators of propositional and predicate calculus.

These conventions for encoding formulae are captured in the following (necessarily case-ridden) definition of the predicate Is_formula, which the reader will want to analyze closely.

```
Is_formula(f) *eq       (Is_seq(f) & (EXISTS g,h | Is_formula(g) & Is_formula(h) &    ((EXISTS v | Is_ir
```

Note that the encoding of formulae defined by the above formula uses sequences Seq2(0,v), where v can be any integer, to encode variables, function symbols, and predicate symbols, without explicitly stating that the integers v appearing in these three different usages must be distinct. Thus in one setting Seq2(0,1) may designate a particular variable, but in another this same pair may designate a predicate or function symbol, quite a different thing. Confusion is avoided by the fact that these usages are distinguished by the contexts in which these pairs appear. Specifically, predicate symbols (resp. function symbols) will only appear as the second component of a sequence whose first component is the code '12' (resp. '13'), where variables cannot appear. For example, in

```
{[0,12],[1,Seq2(0,1)],[2,Seq2(0,1)]}
```

the first occurrence of Seq2(0,1) unambiguously designates a predicate symbol and the second occurrence of Seq2(0,1) unambiguously designates a variable. Thus if we associate some predicate name like 'Foo' with appearances of Seq2(0,1) in predicate contexts and choose to associate strings like 'v_n' with appearances of Seq2(0,n) as variables, the sequence seen above will decode unambiguously as 'Foo(v_1)'.

It is now easy to state comprehensive set of rules for the operation 'Subst' which replaces every free occurrence of a variable x in a formula F with a designated subformula G. We also need the operation which calculates the set of all bound variables of a formula, and the operation which calculates all the free variables of a formula.

The following axiom defines the operation 'Subst'.

```
(f = Seq2(0,x) *imp Subst(f,x,g) = g) &  ((EXISTS y | (f = Seq2(0,y)) &     (not (y = x))) *imp Subst(f,x
```

The following axiom defines the set of all bound variables of a formula.

```
(((not Is_formula(f)) or [0,6] in f or [0,7] in f) *imp     Bound_vars(f) = {}) &   (([0,1] in f or [0,2]
```

The following axiom defines the set of all free variables of a formula.

```
((not Is_formula(f) or [0,6] in f or [0,7] in f) *imp     (Free_vars(f) = {})) &   (([0,12] in f or [0,13
```

Our next step is to define the notion of predicate axiom in coded form. This merely formalizes the statements made in our earlier discussion of the predicate calculus. We begin with encodings of the list of propositional axioms given earlier. Then encodings of predicate axioms (ii-v) follow, and finally encodings of the equality-related predicate axioms (vi-viii). Predicate axiom (v) is simplified slightly (but to an equivalent axiom) by insisting that a term substituted for a free variable in a formula f must have no variables in common with the bound variables of F.

```
Is_propositional_axiom(s) *eq   (EXISTS p,q, r | Is_formula(p) &     Is_formula(q) & Is_formula(r) &
```

Having now defined the notion of 'propositional axiom', we go on to descrbe that of 'predicate axiom'. Note that the coded forms of the predicate axioms listed previously appear in the following formula in the order Axiom (ii), Axiom (iii), Axiom (iv), Axiom (v).

```
Is_predicate_axiom(s) *eq (Is_propositional_axiom(s) or     (EXISTS p,q,x,y,z,u,v,w,c,c1 |       Is_form
```

The collection of axioms specific to set theory can now be defined in coded form. Of course, we need to define codes for the function and predicate symbols which appear in these axioms. We do this as follows: 20: [unordered pair], 21: [ordered pair], 22: union sign, 23: Is_next, 24: Is_integer, 25: Svm, 26: Is_seq, 27: Seq2, 28: Seq3, 29: nullset, 30: power set symbol 'Pow'. Note that the axioms needed fall into two groups, a first 'specialized' group corresponding to the ten set-theoretic axioms displayed earlier in this section, and a remaining 'general' group corresponding to the standard ZFC axioms. The first group is needed to define various predicates and operators which appear along the path to our final definition of the provability predicate 'Pr' which we must define formally in order to state our desired axioms of reflection.The second serve to ensure that the set theory in which we are working behaves in the standard way.

With this understanding, we can encode the collection of ZFC axioms as follows. Note that the first two axioms encoded in what follows are the axiom of subsets and the axiom of replacement, the latter of these in the form

```
(FORALL x,y,z | (f & Subst(f,y,z)) *imp (y = z)) *imp       (FORALL z | (EXISTS c | (FORALL y | (y in c) *
```

The remaining encoded axioms occur in the following order: axiom of subsets, nullset axiom, power set axiom, union set axiom. axiom of infinity, of choice, definition of 'Is_integer', of 'Is_map',of 'Is_seq', axiom of extensionality, definition of unordered pair, axiom of union of two sets, definition of ordered pair, and of 'Is_ext'.

```
Is_ZF_axiom(s) *eq    (EXISTS a,b,f,m,n,n,s,t,u,w,x,y,z,a1,b1,f1,m1,n1,n1,s1,        t1,u1,w1,x1,y1,z1
```

We may also want to include some large-cardinal axiom. We saw in our discussion of these axioms that multiple possibilities suggest themselves. Here is what is reqired for a formal statement of one of them.

```
Ord(o) *eq (FORALL x in o,y| ((y in x) *imp (y in o)) &       ((y in o) *imp (x = y or x in y or y in x))
```

The last formula in the group just shown asserts that there is an inaccessible cardinal M which is the M-th inaccessible cardinal. We saw in our earlier discussion of large-cardinal axioms that this is implied by the assumption that there exists a Mahlo cardinal. A somewhat weaker statement applies if we use M as our model of set theory. In this case all the inaccessible cardinals in M remain inaccessible, but now there are too many of them to constitute a set. The statement that applies is then

```
Large_cardinal_axiom_2 *eq     ((EXISTS o | Is_inaccessible_cardinal(o)) &       (not (EXISTS s | (FORALI
```

It results from our earlier discussion of set-theory models of the form H(n) that If the first set of large-cardinal axioms is assumed, it follows that there is a cardinal M such that H(M) is a model for the set of axioms obtained by replacing Large_cardinal_axiom_1 with the weaker Large_cardinal_axiom_2.

The encodings of the large-cardinal axioms just stated constitute the set of large_cardinal axioms, whose formal definition is as follows. (Note that we use the following codes for the function and predicate symbols which appear: 31: Ord, 32: As_many, 33: Card, 34: Is_regular_cardinal, 35: Is_strong_limit_cardinal, 36: Is_inaccessible_cardinal). Note that the encoded forms of the axioms listed above appear in what follows in the order definition of ordinal, of 'As_many', of 'Is_regular_cardinal', of 'Is_limit_cardinal', of 'Inacessible_cardinal', statement of Large_cardinal_axiom_1, of Large_cardinal_axiom_2.

```
is_largeN_axiom(s) *eq   (EXISTS v,o,x,u,y,w,s,s1,t,t1,f,f1 |     o = Seq2(0,v) & x = Seq2(0,u) & y = Seq2
```

The slightly weakened large cardinal axiom displayed above is encoded by the final clause of this last display.

We can now assert that a formula is an axiom if and only if it belongs to one of the three preceding groups of axioms, and go on to define the notions 'a sequence of statements is a proof', and finally our target 'f is provable'.

```
Is_axiom(f) *eq (Is_predicate_axiom(f) or is_ZF_axiom(f)       or is_largeN_axiom(f))  Is_proof(p) *eq (I
```

Note in connection with the preceding that

```
(EXISTS v | g = Seq3(9,v,h))
```

states that g arises from h by a generalization step, and that

```
[k,Seq3(3,h,g)] in s
```

states that g arises from h and some preceding formula by a modus ponens step.

Statement of the axioms of reflection. Having now managed to include the 'provability' predicate that concerns us in an extension of the ZFC axioms in whose consistency we have some reason to believe, we can reach our intended goal by stating the axioms of reflection. These are simply all statements of the form

```
Pr('F') *imp F,
```

where F is any syntactically well-formed formula of our set-theoretic language, and 'F' is its syntax tree, encoded in the manner described above.

Potential uses of these axioms have been explained above. Of course, we only want to add axioms of reflection to our basic set if inconsistency does not result. We shall now prove that this must true if we assume that there exists at least one inaccessible cardinal but do not include this assumption in the set of axioms which enter into the definition of the predicate 'Pr'. More generally, consistency is assured if we assume that there exists an inaccessible cardinal, but in the axioms which enter into the definition of the predicate 'Pr' we include only a large cardinality statement that is true for the set of all cardinals M less than N.

So the setting in which we work is as follows. We let N be an inaccessible cardinal, and let U = H(N) be the set defined recursively by

```
H_(x) := if x = {} then {}     else Un({pow(H_(y)): y in x}) end if
```

and

```
H(N) := {H_(n): n in N},
```

as in Chapter 2, so that as shown there all of the axioms of set theory remain valid if restrict our universe of sets to U. This statement makes reference to the set U, hence to N, and so is a theorem of the extension ZFC^+ of ZFC in which an axiom stating the necessary properties of N, for example stating that it is an incaccessible cardinal, is present.

For each syntactically well-formed formula F, we let F^U be the result of relativizing F to U in the following way. We process the syntax tree of F, modifying quantifier nodes but leaving all other nodes unchanged. Each universal quantifier

```
(FORALL x1,...,xn | P)
```

is changed into

```
(FORALL x1,...,xn | (x1 in U &...& xn in U) *imp P);
```

Each existential quantifier

```
(EXISTS x1,...,xn | P)
```

is changed into

```
(EXISTS x1,...,xn | x1 in U &...& xn in U & P);
```

We let A_0 be the assignment which maps the collection of predicate and function symbols which appear in the chain of definitions leading up to the definition of the provability predicate 'Pr' (or, more properly, maps the integers which encode these symbols) in the following way. (The symbols in question are '=', 'in', '{.}' (unordered pair), '[]' (ordered pair), '+', 'Is_next', 'Is_integer', 'Svm', 'Is_seq', 'Seq2', 'Seq3', '{.}' (nullset), 'Pow').

```
 '='   is mapped into      {[[x,y], if x = y then 1 else 0 end if]: x in U, y in U}   'in'   is mapped in
```

The following lemma states the intuitively obvious property of this assignment that we need below.

Evaluation Lemma: Let N, H(N), U, ZFC+, and A_0 be as above, and for each syntactically well-formed formula F let F^U be as above, and let 'F' denote the syntax tree of F. Given any list of variables $v_1,...,v_n$ and equally long list $x_1,...,x_n$ of elements of U let $A_0(v_1==>x_1,...,v_n==>x_n)$ be the assignment which maps each v_j into the corresponding x_j.

Then if $x_1,...,x_n$ is the list of free variables of F, and 'x_j' designates the symbol naming the j-th of these variables for each j between 1 and n, it follows that

```
(FORALL x1 in U,...,xn in U |     (Val(A0('x1'==>x1,...,'xn'==>xn),'F') = 1) *eq FU)
```

is a theorem of ZFC+.

Evaluation Lemma for terms: Let N, H(N), U, and A_0 be as above, and for each syntactically well-formed term F let F^U be as above, and let 'F' denote the syntax tree of F.

Then if $x_1,...,x_n$ is the list of variables of F, and 'x_j' designates the symbol naming the j-th of these variables for each j between 1 and n, it follows that

```
(FORALL x1 in U,...,xn in U |     Val(A0('x1'==>x1,...,'xn'==>xn),'F') = F)
```

is a theorem of ZFC+.

Corollary: Under the same hypotheses as above, suppose that the formula F contains no free variables. Then

```
(Val(A0,'F') = 1) *eq FU
```

is a theorem of ZFC+.

We prove the evaluation lemmas by induction of the size of the syntax tree of F, starting with the evaluation lemma for terms. If a term is just a variable x, then

```
Val(A0('x'==>x),'x') = x
```

for each x, so the lemma holds in this case. If a term is formed directly from one of the primitives used above by supplying the appropriate number of variables as arguments to the primitive, as for example in '{x,y}', then we have

```
Val(A0('x'==>x0,'y'==>y0),'{x,y}') =     {[[x,y],{x,y}]: x in U, y in U}(x0,y0) = {x0,y0}
```

for all x_0 and y_0 in U, so the lemma holds in this case also. Similarly elementary observations cover all the other function symbols appearing in the list displayed above, namely '[.]', '+', 'Seq2', 'Seq3', and 'Pow'. The reader is invited to supply details.

Now suppose that the evaluation lemma for terms fails for some f, and, proceeding inductively, that it fails for no term having a syntax tree smaller than that of F. Then F must have the form

```
f(t1,..,tk)
```

where f is one of the primitive function symbols appearing in the list displayed above t_1,..,t_k are subterms. By definition we have

```
Val(A0(v1==>x1,...,    vn==>xn),'f(t1,..,tk)') =       A0('f')(Val(A0(v1==>x1,...,vn==>xn),    't1'),...,Val(A
```

By inductive hypothesis the statement

```
(FORALL x1 in U,...,xn in U |     Val(A0('x1'==>x1,...,'xn'==>xn),tj) = tj)
```

is a theorem of ZFC+ for each j from 1 to k, where x_1,...,x_n is the list of all free variables appearing in any of the terms t_j. Hence we have

```
(FORALL x1 in U,...,xn in U | Val(A0('x1'==>x1,...,'xn'==>xn),    'f(t1,..,tk)') = A0('f')(t1,...,tk))
```

Now we need to consider the function symbols appearing in the list displayed above, namely '{.}', '[.]', '+', 'Seq2', 'Seq3', and 'Pow'. The argument is much the same in all of these cases. For example, for the power set symbol 'Pow' we have

```
A0('Pow')(t1) = {[x,{y: y *incin x}]: x in U}(t1)     = {y: y *incin t1} = pow(t1),
```

so

```
(FORALL x1 in U,...,xn in U | Val('x1'==>x1,...,'xn'==>    xn),'Pow(t1') = pow(t1),
```

proving our claim in this case. The reader is invited to supply the corresponding details in the remaining cases, namely '{.}', '[.]', '+', 'Seq2', and 'Seq3'. Together, these prove the evaluation lemma for terms in all cases. **QED**.

Next we prove the evaluation lemma for formulae, beginning with atomic formulae, whose lead symbols must be one of the predicate symbols appearing in the list above, namely '=', 'in', 'Is_next', 'Is_integer', 'Svm', Is_seq'. Consider for example the case if an atomic formula whose lead symbol is 'in'. This must have the form

```
't1 in t2'
```

where t_1 and t_2 are terms, and so by definition and using the evaluation lemma for terms we have

```
(FORALL x1 in U,...,xn in U |     Val(A0('x1'==>x1,...,'xn'==>xn),'t1 in t2')   A0('in')(Val(A0('x1'==>x1,...
```

Hence

```
(FORALL x1 in U,...,xn in U |   (Val(A0('x1'==>x1,...,'xn'==>xn),'t1 in t2') = 1) *eq t1 in t2),
```

proving our claim for atomic formulae whose lead symbol is 'in'. The reader is invited to supply the corresponding details in the remaining cases, namely '=', 'Is_next', 'Is_integer', 'Svm', Is_seq'. Together, these cover all atomic formulae.

General formulae are built from atomic formulae by repeated application of the propositional operators &, or, *imp, *eq, not, and the two predicate quantifiers. Inductive arguments like those just given apply in all these cases For example, if f has the form 'g & h' where g and h both satisfy the conclusion of the Evaluation Lemma, and our notations are as above, then

```
(Val(A0('x1'==>x1,...,'xn'==>xn),'g & h') = 1)   *eq Min(Val(A0('x1'==>x1,...,'xn'==>xn),'g'),     Val(A0('x1
```

Next suppose that f has the form

```
(+)  (FORALL y1,...,yk | g)
```

where g satisfies the conclusion of the Evaluation Lemma. Since both f^U and Val(A",f) (for any suitable assignment A") are unchanged if variables y_j not free in g and repeated copies of variable are dropped from $y_1,...,y_k$, we can suppose that there are none such. Hence, if the free variables of formula (+) are $x_1,...,x_n$, then the free variables of g are $x_1,...,x_n,y_1,...,y_k$. Therefore

```
(Val(A0('x1'==>x1,...,'xn'==>xn),'(FORALL y1,...,yk | g)') = 1)  *eq Min(Val(A0','g'),
```

where the minimum is extended over all all assignments A_0' which cover all the variables 'x_1',...,'x_n' and 'y_1',...,'y_k', and which agree with A_0('x_1'==>x_1,...,'x_n'==>x_n) except possibly on the variables 'y_1',...,'y_k'. That is,

```
(Val(A0('x1'==>x1,...,'xn'==>xn),'(FORALL y1,...,yk | g)') = 1)  *eq Min(A0('x1'==>x1,...,'xn'==>xn,    'y1'=
```

where now the minimum is extended over all possible values of $y_1,...,y_k$, all belonging to U. Hence, since by inductive assumption we have

```
(FORALL x1,...,xn,y1,...,yk |      (Val(A0('x1'==>x1,...,'xn'==>xn,'y1'==>y1,...,'yk'==>yk),'g') = 1) *eq g),
```

it follows that

```
(Val(A0('x1'==>x1,...,'xn'==>xn),'(FORALL y1,...,yk | g)') = 1)  *eq (FORALL y1,...,yk | Val(A0('x1'==>x1,...
```

verifying the Evaluation Lemma in this case also.

The reader is invited to supply the details of the remaining cases.

This concludes our proof of the Evaluation Lemma.

Next we note and will prove the following fact concerning the value Val(A_0,F) for every formula provable from the axioms of ZFC, as possibly extended by a collection of large cardinal axiom like those displayed above. To avoid too much obscuring detail, we shall state and prove this fact using English in the normal way to abbreviate formal set-theoretic details.

Lemma 2: Let ZFC* be the collection of axioms of ZFC, possibly extended by a set of large cardinal axioms like those displayed above. Let Pr be the proof predicate defined by these axioms in the manner detailed above. Let CFP all the function and predicate symbols that appear in the axioms ZFC*. Let Is_assignment(A,U) be the set-theoretic formula which asserts that A is an assignment with universe U, and let True_on_ax(A) be the formula which asserts that Val(A,s) = 1 for the encoded form of every axiomof ZFC* (The reader is invited to write out the details of these formulae). Then

```
(FORALL s | (Pr(s) & Is_assignment(A0,U) & True_on_ax(A0) &   domain(A0) incs (Free_vars(s) + CFP)) *imp V
```

is a theorem of ZFC.

Proof: Suppose not, so that there exist A_0, U, and s satisfying

```
Pr(s) & Is_assignment(A0,U) & True_on_ax(A0) &     domain(A0) incs (Free_vars(s) + CFP) & Val(A0,s) = 0.
```

Using the definition of 'Pr', it follows that there exists a p such that

```
Is_proof(p) & Is_integer(n) & [n,s] in p      & Is_assignment(A0,U) & True_on_ax(A0) &         domain(A0)
```

The induction principle available in ZF set theory allows us to improve this last statement to

```
Is_proof(p) & Is_integer(n) & [n,s] in p    & Is_assignment(A0,U) & True_on_ax(A0) &     domain(A0) incs (
```

By the definition of Is_proof, we have

```
(+)  Is_axiom(s)     or (EXISTS m in n,h | ([m,h] in p) &         (EXISTS v | s = Seq3(9,v,h))    or (EXIST
```

The alternative Is_axiom(s) of (+) is ruled out since in this case we have Val(A_0,s) = 1 from True_on_ax(A_0). In the second case of (+) there exist m, h, and v such that

```
m in n & [m,h] in p & s = Seq3(9,v,h).
```

It follows from 'm in n' that

```
(domain(A) incs (Free_vars(g) + CFP)) *imp         (Val(A,h) = 1)
```

for any assignment A that agrees with A_0 on CFP. Since s = Seq3(9,v,h), which states that the decoded version of s has the form

```
(FORALL v | d)
```

where d is the decoded form of h, it follows that for each assignment A' for which

```
domain(A') incs (Free_vars(s) + CFP))
```

that agrees with A_0 on CFP we must have Val(A',s) = 1, since this is a minimum of values Val(A,h), extended over assignments A that agree with A' on Free_vars(s) + CFP. Hence Val(A_0,s) = 0 is impossible in the second case of (+) also. In the remaining case there exist formulae h and g, and integers m and k in n, such that

```
[m,h] in p & [k,g] in p & m in n & k in n & g = Seq3(3,h,s).
```

it follows as above that

```
(domain(A) incs (Free_vars(g) + CFP)) *imp (Val(A,h) = 1 & Val(A,g) = 1),
```

for each assignment A that agrees with A_0 on CFP. From this it follows that

```
(++)  (domain(A) incs (Free_vars(g) + CFP)) *imp (Val(A,s)),
```

since g = Seq3(3,h,s) states that the decoded form of g is e *imp d, where e and d are the decded froms of h and s repectively. But any assignment A' such that

```
(*)  (domain(A') incs (Free_vars(s) + CFP))
```

can be extended to an assignment satisfying domain(A) incs (Free_vars(g) + CFP) by defining it arbitrarily on the elements of the difference set Free_vars(g) - Free_vars(s), and plainly Val(A',s) = Val(A,s) for any such assignment. Hence

```
  (domain(A') incs (Free_vars(s) + CFP)) *imp (Val(A,s')),
```

for each assignment A that agrees with A_0 on CFP. This rules out the third alternative of (+) and so concludes our proof of lemma 2. **QED**.

The following statement now follows easily from the Lemmas proved above.

Theorem: Let ZFC$^+$ be the ZFC axioms of set theory, supplemented by an axiom stating that there exists at least one inaccessible cardinal N. Let A_0 be the model of set theory with universe H_(N) described above. Then it is a theorem of ZFC$^+$ that A_0 is a model in which all the axioms of ZFC, plus all the reflection axioms

```
  Pr('F') *imp F,
```

where F is any syntactically legal formula and 'Pr' means 'provable from the axioms of ZFC' (without the axiom of existence of an incaccessible cardinal) are true. Hence it follows from the axioms of ZFC$^+$ that this set of axioms is consistent.

Proof: The proof is simply as follows. Let F be a syntactically legal formula without free variables. The Corollary to Evaluation Lemma tells us that

```
  (Val(A0,'F') = 1) *eq FU
```

is a theorem of ZFC$^+$, and Lemma 2 tells us that

```
  Pr('F') *imp (Val(A0,s) = 1)
```

is a theorem of ZFC. Our theorem follows immediately from these two statements. **QED**.

The authors are indebted to Prof. Mark Fulk for suggesting the line of thought developed in this section. Richard Weyhrauch has explored similar ideas (cf. Aiello and Weyhrauch., Using Meta-theoretic Reasoning to do Algebra, Lecture Notes in Computer Science v. 87, pp 1-13. Springer Verlag, 1980).

4.4. A digression concerning foundations

> *Absolute, true, and mathematical time, of itself, and from its own nature flows equably without regard to anything external... Absolute space, in its own nature, without regard to anything external, remains always similar and immovable.*
>
> Isaac Newton, Principia, 1687

Much has been written about the broader significance of Gödel's results. The first Gödel theorem (undecidability) is best viewed as a special case of Chaitin's more general result, which states a general limit on the power of formal logical

reasoning. Specifically, Chaitin exhibits a large class of statements which can be formalized but not formally proved. But why, in hindsight, should it ever have been expected that every statement which can be written in a formalism can also be proved in the formalism? There is plainly no reason why this should be so, and much prior mathematical experience points in the opposite direction (for example, few elementary functions have elementary integrals).

Gödel's theorems cast some light on the classical philosophical distinction between analytic (a-priori) knowledge and the kind of inductive knowledge for which the scientist strives. The boundary lines drawn between these two realms of knowledge have shifted in the course of time. As illustrated by the famous nineteenth century changes in the dominant view of geometry, the analytic realm has steadily lost ground to the inductive realm. Originally the axioms of geometry were seen as statements about the physical universe, involving idealizations (e.g. consideration of points of zero extension and lines of zero thickness) which did not misrepresent physical reality in any significant way. Now the dominant view is that space is probably not Euclidean, and may not even be continuous. This means that classical geometry is but a crudely approximate model of some aspects of physical reality, and that the best evidence for the consistency of its traditional axioms is the fact that they have a formal set-theoretic model. Few will now include more than arithmetic, and perhaps abstract set theory, in the analytic realm. But if the theorems of arithmetic are analytic knowledge, but not knowledge of the physical world, what are they knowledge about? Perhaps they represent a-priori knowledge about computation. Let us consider this possibility.

To use reasoning in a particular logical system as a surrogate for computation, we need only be certain that the logical system in which we reason should be consistent. But, since Gödel's second theorem tells us that formal proof of the consistency of a system L requires use of a system different from L and probably stronger than L, how can such certainty ever be achieved? One way in which such a belief, though of course no certainty, can arise is by the accumulation of experience with the objects of a certain realm, leading to formulation of statements concerning the entities of that realm. Generalized and formalized, these can subsequently become the axioms and theorems of a fully elaborated formal system. If experience then seems to show that the resulting formal system is consistent, this may be taken as evidence that its theorems are true statements about objects of some kind.

The hope of grounding mathematics, and hence the analytic truths which it may claim to embody, on a set of intuitions less tenuous than those available in a set theory incorporating Cantor's aggressive linguistic extensions has lent interest to various narrower formalisms. The most important of these is the pure theory of integers, in the form given it by Peano. Integers originally come into one's ken as the words 'zero, one, two, three,..' of a kind of poem which, as a child, one learns to repeat, and with whose indefinite extensibility one becomes familiar. (The simplest, though not the most convenient, form of the number poem is its monadic version: 'one, one-and-one, one-and-one-and-one,...) Experience with such collections of words leads to the following generalizations: (i) given any such word, there is a way of forming the very next word in the series; (ii) no word occurs twice in the series thereby generated; (iii) If one starts with any word in the sequence and repeatedly steps back from words to their predecessors, one will eventually reach zero.

Peano's axioms formalize these intuitions. They can be stated as follows:

(i) 0 is a natural number

(ii) For every natural number x there exists another natural number x' called the successor of x.

(iii) not (0 = x') for every natural number x (x' being the successor of x)

(iv) If x' = y' then x = y

(v) If Q is a property of natural numbers and x is a number such that Q(0) /= Q(x), then there exists a natural number y such that Q(y) /= Q(y').

A compelling way of linking Peano's axioms to primitive physical experience is to regard them as statements about integers in monadic notation, i.e. sequences of marks having the form

```
/ // /// //// ...
```

(The empty sequence is allowed). In physical terms, two such sequences are equal if their ends match when the sequences are set side by side. An extra mark can be added to the end of any such sequence, giving Peano's successor operation x'. If a sequence s is nonempty, we can erase one mark from its end, giving the 'Prev' operation. Experience indicates that Prev(x') = x (erasing a mark just added to x restores x), and that if x is not empty Prev(x)' = x (adding back a mark just removed from x restores x). Peano's axioms (ii-iv) follow readily from these statements. Experience also indicates that if marks are repeatedly erased from the end of any such x the empty string will eventually result. If Q(x) is any stable property of such sequences of marks, this gives us a systematic way of searching for two successive integers y, y' for which Q(y) and Q(y') are different, and so justifies Peano's axiom (v) of induction.

But the following objection can be raised to the universal quantification of these axioms. Physics tells us that sufficiently large sequences of marks need not behave in the same way as shorter sequences. For example, even if marks are stored in the most compact way likely to be feasible, as states of single atoms, it is probably not possible to store more than 10^{27} marks per kilogram of matter. 10^{57} marks would therefore have a mass roughly equal to that of the sun, and so could be expected to ignite a nuclear chain reaction spontaneously. The mass of our galaxy is unlikely to be more that 10^{13} times larger, so 10^{70} marks would have a mass larger than that of our galaxy. Compactly arranged, these marks would promptly disappear into a black hole. This makes it plain that integers larger than $2^{10^{70}}$ are fictive constructs, which can never be written out fully even in monadic notation. This suggests that denotations like $10^{10^{101010}}$ of much larger integers should be viewed as logical specifications Exp(10,Exp(10,Exp(10,Exp(10,10)))) of hypothetical computations that can in fact never be carried out. However, this does not make them useless, since we can reason about them in a system believed to be consistent, and such reasoning may lead us to useful conclusions about other, perfectly feasible, computations.

Note also that the direct evidence we have for the consistency of any logical system can never apply to proofs more than 10^{70} steps in length, since for the reasons stated above we can never expect to write out any such proof. Of course, this does not prevent us from reasoning about much longer proofs, but as above it may be best to regard such reasoning as the manipulation of marks in some formalized meta-logical system believed to be consistent.

Any attempt to move the consistency of Peano's full set of statements from 'belief' to 'certainty' would therefore seem to rest on a claim that there really exists a Platonic universe of objects, for example idealized integers, about which our axioms are true statements, and hence necessarily consistent. But how can truths about such Platonic universes be known reliably? Two methods, direct intuition and reasoning from consequences, suggest themselves. Claims concerning direct intuition are doubtful. If direct intuition is admitted as a legitimate source of knowledge, how can we decide between the rival claimants to direct intuition, if their claims differ? And, even if we can convince ourselves that all normal humans have the same intuition, why should the objective truth of this intuition be admitted? In biological terms the human differs little from the nematode, except for possessing limbs and an enlarged nervous system. Like the squid, we have a highly elaborated visual system. Beyond this, we have an ability to deal with language, and so to deal with abstract patterns. But, as work with visual illusions shows plainly, even stable, immediate, compelling, and universally shared perceptions about physical reality can be wrong. Even such immediate perceptions do not tell us what the world really contains, but only how our nervous systems react to that content. Why should the far more elusive mechanisms of logical intuition be more trustworthy? To move toward truth we must employ the patient and never conclusive methods of experimental science, and experience shows science to progress best when it tries actively to probe the limits of its own best current view.

A 'realist' or 'Platonist' view, reflecting the belief that the statements of mathematics are truths about some partly but progressively comprehended class of ideal objects can be stated as follows. Of all formally plausible axioms (for example, statements asserting the existence of various kinds of very large cardinal numbers) not provable from assumptions presently accepted, a growing and coherent collection, this view predicts, will prove to be particularly rich in consequences, including consequences for questions that can be stated in currently accepted terms, but not settled. These

new axioms may be taken to hint at an underlying truth. The negatives of these progressively discovered axioms will prove to be unfruitful and so will gradually die out as dead ends.

In contrast, a more purely formalist view of the situation suggests that this may not prove to be the case, but that as collections of axioms rich in interesting consequences are found these collections will prove to be mutually contradictory and so not suggestive of any progressively revealed underlying truth. It further suggests that a useful path to progress may lie in the attempt to undercut existing axiom systems by looking for competing and incompatible systems identical with current systems only in areas covered by actual experience.

The theory of hereditarily finite sets presented in the preceding pages is logically equivalent to Peano's theory of integers, in the sense that (as we have shown) integers can be modeled within this restricted theory of sets, while conversely the hereditarily finite sets can be encoded as integers. However the theory of hereditarily finite sets has a considerably richer intuitive content. Peano's system captures only the process of counting, but without anything to count, or indeed any evident way of capturing the notion of 1-1 correspondence fundamental to the actual use of counting. The axiomatization of hereditarily finite sets given above is better in this regard, since it makes notions like mapping and function value more accessible.

Cantor's full theory of infinite sets goes beyond Peano's system, and is in fact strong enough to allow proof that Peano's system is consistent. However, in formal terms the step from the theory of hereditarily finite sets to the full Cantor theory is slight. Three changes suffice. First, an axiom of infinity (existence of at least one infinite set) must be added, and the subset induction principle abandoned in favor of the more limited principle of membership induction. Then the existence of the set of all subsets of a given set, which follows as a theorem in the theory of hereditarily finite sets, must be assumed as an axiom. Why do we assume that these changes leave set theory consistent?

The answer is in part historical. The application of set-theoretic reasoning in geometric and analytic situations dominated by the notion of the 'continuum' led to Cantor's theory of infinite sets. In Descartes' approach to geometry the real line is ultimately seen as the collection of all infinite decimals, which naturally introduces work with sets whose elements have no natural enumerated order. Further geometric and analytic studies of important point loci lead to work with increasingly general subsets of the real line. Out of such work the conviction grows that much of what can be said about infinite collections is entirely independent of the manner in which they are ordered, and so requires no particular ordering. This linguistic generalization was made systematically by Cantor, who then moved on to unrestrained discussions of sets, involving such notions as the set of all subsets of an infinite set. This led him to consider statements concerning sets which are (more and more) uncountably infinite. Of course, as such generalized language moves away from the areas of experience in which it originates, the evidence that what is being said is more than word-play destined to collapse either in self-contradiction or in collision with some more useful logical system, becomes increasingly tenuous. Fears of this kind certainly surrounded Cantor's work in its early decades, as indicated by his 1883 remark '... I realise that in this undertaking I place myself in a certain opposition to views widely held concerning the mathematical infinite and to opinions on the nature of numbers frequently defended', and by Kronecker's remark that Cantor's new set theory was 'a humbug'. And in fact set theory, in the overgeneralized form initially given it by Cantor, does lead to inconsistencies if pushed to the limit, as Cantor was pushing so many of his set-theoretic ideas. However, this fact did not lead to the collapse of the theory, but only to its repair. Such repair, in a manner preserving the validity of Cantor's general approach and all of his appealing statements concerning infinite sets, underlies the formalized set theory used in this book. It leads to a formalism that, as far as we know, is consistent, and that provides a good foundation for the now immense accumulation of work in mathematical analysis. Cumulatively this work gives evidence for the consistency of set theory that is just as compelling as the like evidence of the consistency of arithmetic. Only the existence of a set theoretic proof that arithmetic is consistent, and of an arithmetic proof that no such proof is possible in pure arithmetic, would seem to justify a much greater degree of confidence in the consistency of the one rather than the other of these systems.

The remarks made in the preceding paragraphs suggest the following cautious view of the distinction between analytic and inductive knowledge. Inductive knowledge is the always uncertain knowledge of the physical universe gained by studying it closely and manipulating it in ways calculated to uncover initially unremarked properties of reality stable

enough to be understood. Analytic knowledge is knowledge of an aspect of reality, specifically certain aspects of the behavior of marks and signs (e.g. on paper or computer tape) limited enough for guessed generalizations to have a good chance of being correct. (It is in fact hard to do without these guesses, since the marks and signs to which they relate are the tools we use to reason in all other areas of science). What we believe about these signs is that a certain way of manipulating them (by logical reasoning) that is somewhat more general than the standard process of automated computation seems to be formally consistent. Although inconsistencies, requiring some kind of intellectual repair, might possibly be found in still unexplored areas of the realm of these signs, we still have no idea of how to convert this suspicion into something useful.