Journal of Automated Reasoning 1 (1985) 407–434. 0168-7433/85.15.
© 1985 *by D. Reidel Publishing Company.*

Towards Mechanical Metamathematics

N. SHANKAR
Institute for Computing Science, 2100 Main Building, The University of Texas at Austin, Austin, TX 78712, U.S A.

(Received: 9 July 1985)

Abstract. Metamathematics is a source of many interesting theorems and difficult proofs. This paper reports the results of an experiment to use the Boyer–Moore theorem prover to proof-check theorems in metamathematics. We describe a First Order Logic due to Shoenfield and outline some of the theorems that the prover was able to prove about this logic. These include the tautology theorem which states that *every tautology has a proof.* Such proofs can be used to add new proof procedures to a proof-checking program in a sound and efficient manner.

Key words: Metamathematics, mathematical logic, automatic theorem proving, metamathematical extensibility, automatic proof-checking, tautology theorem

But formalized mathematics cannot in practice be written down in full. . . . We shall therefore very quickly abandon formalized mathematics. N. Bourbaki [1]

1. Introduction

A *formal system* or a *formal logic* consists of *axioms* from which *theorems* are derived by repeated application of certain mechanical *rules of inference.* The derivations of theorems from the axioms are termed *formal proofs.* Many formal systems have been developed, mainly as a result of the increased attention paid to mathematical rigor in the 19th century. Formal logics provide us with the clearest definition of a valid mathematical argument. Another advantage of formal proofs is that they can be mechanically checked using programs known as *automatic proof-checkers.* In spite of this few mathematicians use formal proofs in everyday mathematical reasoning. One major reason for this is that the basic proof-steps allowed by most formal logics do not include many of the steps routinely used in informal mathematical reasoning and therefore constructing formal proofs a long and tedious process. The question as to whether formal mathematical reasoning is practically possible has engendered numerous debates and one view of this is provided by the Bourbaki remark above. To get around the difficulty of writing formal proofs, considerable effort has been devoted to showing that many of the proof-steps used in informal mathematical reasoning are formalizable in some formal logic. This has led to a mathematical study of formal systems, termed *metamathematics* or *mathematical logic* [7, 11]. In metamathematics, mathematical techniques such as induction are used to prove that a given proof-step is formalizable in a logic by showing that each application of that proof-step is reducible to some series of applications of the rules and axioms of that logic. A new proof-step which is formalizable in the given formal logic is termed a *derived inference rule* in the logic. This approach can also be applied to automatic proof-checkers to extend them to check new proof-steps. Thus, one may start with a simple proof-checker which checks only formal proofs. Its correctness can be established by careful inspection. It can then be extended by adding frequently used patterns of proof as new proof-steps. Before we do this, we must establish that the new proof-step is a derived inference rule. If this is so, the extension is said to preserve *soundness*. The purpose of this paper is to demonstrate the effective use of an automatic theorem prover in proving the soundness of significant extensions to a formal logic. This ability to make sound extensions to a proof-checker has been termed *metamathematical extensibility* [5].

There have been two approaches to building extensible proof-checkers. The first approach involves the use of the proof-checker to prove the soundness of new extensions to itself. This approach has been used by Weyhrauch in FOL [12] and by Boyer and Moore [3]. This approach, as exemplified by the FOL system, has two drawbacks:

1. Proofs of soundness of extensions are difficult and are tedious to carry out on a simple proof-checker.
2. The FOL system requires the user to supply the executable code to be added to the proof-checker corresponding the extension that has been proved sound. Human error can creep in at this point.

In the second approach to extensibility, one avoids proving the soundness of new extensions by expanding out the corresponding formal proof each time a new proof step is used. The expanded proof can be automatically generated by executing a program which ostensibly constructs the correct formal proof to justify the use of that proof-step. The expanded proof is then checked with the original proof-checker. This approach has been used in Edinburgh LCF [6] and by Brown [4]. The drawbacks of the LCF system are:

1. Checking the formal proof each time a new proof step is used is time-consuming.
2. The expanded proof corresponding to each application of a new proof-step is provided by a user-supplied program. While there are ways to ensure that a proof thus generated is always a correct proof, it might not be the proof that justifies the use of the new proof-step. This would mean that there was an error in the program which constructed the formal proof. Locating and fixing errors in such programs may not always be easy.

The approach discussed in this paper avoids the second drawback of the LCF system by mechanically proving that the expanded proof constructed by the program is always the correct one. Once we have proved this, we need never expand out the proof since it is sufficient to know that such a proof exists, thus avoiding the first drawback of the LCF system. Our approach is therefore quite similar to the FOL approach. Both the drawbacks of the FOL approach are also avoided by the use of the Boyer–Moore theorem prover [2], a powerful prover for LISP functions. This is done by representing the proof-checker as a function in the Boyer–Moore logic and then proving the soundness of the extensions using the Boyer–Moore theorem prover. This approach has the following advantages:★

1. The Boyer–Moore theorem prover proves properties of LISP functions and makes powerful use of induction. This makes it easy to express theorems about formal systems as well as verify them.
2. The theorem prover translates its functions into correct, efficient and executable LISP thus avoiding the second drawback of the FOL system [3]. This allows us to extract usable code from the theorems that we have proven and actually construct working proof-checkers.
3. Once we have proved the soundness of a derived proof step, we need never examine the formal derivation for each application of the proof step, but nevertheless retain the ability to do so.

A drawback of this approach is that it requires one to trust the soundness of the Boyer–Moore theorem prover. Since the prover has been widely used and is well-tested and documented, there is good reason to believe that it is in fact sound. Another drawback is that the person carrying out these extensions and their proofs must be knowledgeable about mathematics as well as the use of the theorem prover. This applies to the previous approaches as well.

The main result of this report is a metatheoretic formalization of Shoenfield's First Order Logic [11] in the Boyer–Moore Logic and a mechanical proof of the Tautology Theorem for the above logic using the Boyer–Moore theorem prover. The Tautology Theorem is a significant result in mathematics and was first proven by Emil Post in this doctoral dissertation [9]. A tautology is a Boolean expression whose truth-table evaluation under any assignment of **T** or **F** to the atomic expressions always yields **T**, given the usual interpretation of the logical connectives. Our version of the Tautology Theorem states that every tautology has a proof in the above-mentioned logic. This theorem justifies one of the most commonly used rules of inference in informal mathematical arguments. It also proves the propositional completeness of the formal logic.

Proofs in metamathematics involve two levels of reasoning. The proof itself is carried out at the meta-level using a meta-language. The systems of logic whose properties we wish to prove, constitute the object-level. When referring to objects at the object-level, we prefix the word 'formal' and to those at the meta-level, we add the prefix 'meta'. Thus we refer to, 'a formal variable' as opposed to a 'meta-variable'.

★ These comparisons are somewhat unfair since the aims of the FOL and LCF systems are slightly different from those of the project described in this paper. FOL is intended as an experiment to study the checking of proofs in various formal theories where the metatheory is itself formally expressed in First Order Logic. LCF is intended as a realistic interactive proof-checker for properties of programs, in which it is possible to build theories and experiment with various proof strategies. The project described in this paper is aimed at investigating the efficacy of the Boyer–Moore theorem prover in proving difficult theorems in proof-theoretic metamathematics.

In most textbooks, metal-level reasoning is done informally and the language used is some natural language, e.g. English, German. In the mechanical proofs we describe below, the meta-level reasoning is done formally using the Boyer–Moore theorem prover and the metalanguage is pure LISP. To further complicate matters, we also need an informal meta-language, English, to informally describe the proof. We will adopt the convention of using *italicised letters* for the object language, **bold-face letters** for special objects in the informal meta-language and UPPER-CASE letters for the formal meta-language pure-LISP.

To familiarize the reader with the LISP notation used in the proofs, Section 2 contains a brief overview of the Boyer–Moore Theorem Prover and the Boyer–Moore Logic. Section 3 introduces the formal system, a First Order Logic due to Shoenfield (termed SFOL below). Section 4 is an outline of the informal proof of the tautology theorem. Section 5 is the longest section in the paper and covers the main result. In it we describe the proof-checker which checks proofs in SFOL and show how this proof-checker was extended in a significant way by outlining the mechanical proof of the tautology theorem. In Section 6, we draw several conclusions the mechanical proofs.

The complete sequence of events leading to the proof of the tautology theorem covers about 35 typed pages. It has therefore been omitted. This sequence of events can be obtained from an earlier version of this paper [10].

2. The Boyer–Moore Theorem Prover

This section contains a brief overview of the Boyer–Moore theorem prover and its logic. Readers familar with the theorem prover may skip this section. A thorough survey of the prover can be found in Boyer and Moore's *A Computational Logic* [2].

2.1. THE LOGIC

The language of the Boyer–Moore theorem prover is a quantifier-free first order logic with equality. It employs a LISP-style prefix notation so that (FN X1 X2 . . . Xn) denotes the result of applying the function FN to the values of the arguments X1, X2, . . ., Xn. The basic theory includes axiomatizations of literal atoms, natural numbers, and lists.

Constants in the logic are functions with no arguments. The logical constants in the logic are (TRUE) and (FALSE), abbreviated as T and F respectively. The 3-place function IF is the only primitive logical connective. (IF X Y Z) is axiomatized to return Z if X is equal to F, and Y otherwise. Thus, (IF X Y Z) can be informally read as: *If* X *then* Y, *else* Z. Other logical connectives, such as OR, NOT, AND and IMPLIES can be defined in terms of IF.

Equality is represented by the dyadic function EQUAL. (EQUAL X Y) is axiomatized to return T if X and Y identical, and F, otherwise. Note that functions which return only T or F play the role of predicates. The theory includes the axiom of reflexivity of equality and an equality axiom for functions.

An assertion *p* in the logic is a theorem if and only if it can be proved that no instance of *p* is equal to F.

The *Shell Principle* allows the user to add axioms about inductively constructed objects. Lists are axiomatized by adding a shell with a recognizer, the one argument function LISTP; a constructor, the two argument function CONS; two destructors, the one argument functions CAR, and the one argument function CDR; and a bottom object, (NIL). This results in the creaction of axioms for lists which are similar to Peano's axioms for natural numbers.

To provide some intuition, we show the results of evaluating the above functions. (CONS X Y) returns a list whose first element is X and the remainder of the list is Y, e.g. (CONS '(1 2) '(3 4 5)) returns the list '((1 2) 3 4 5). Also, (CAR '((1 2) 3 4 5)) returns '(1 2), and (CDR '((1 2) 3 4 5)) returns '(3 4 5). Nested sequences of CARs and CDRs are abbreviated, e.g. (CADDR X) abbreviates (CAR (CDR (CDR X))). (LISTP '(1 2)) is T and (LISTP NIL) is F. (NLISTP X) abbreviates (NOT (LISTP X)).

The *Principle of Definition* is used to admit definitions of new functions as axioms. A function definition is accepted if it is recursively or non-recursively defined in terms of previous defined functions and there is some well-founded ordering, i.e. a partial ordering in which there are no infinite decreasing chains, on some measure of the arguments which decreases with every recursive call. The two-argument function LESSP, which is the standard ordering predicate on natural numbers, is the most commonly used well-founded ordering. Every evaluation of the functions admitted under this principle is guaranteed to terminate. This ensures that no new inconsistency is introduced by the addition of the new axiom.

The rules of inference in the Boyer–Moore logic consist of:

1. *Propositional Calculus*: All tautologies are theorems.
2. *A Principle of Noetherian Induction*: This allows the prover to formulate an induction that is justified by the well-founded orderings created under the principle of definition.
3. *Instantiation*: If *p* is a theorem, so is any instance *p′* of *p* that is got by replacing every occurrence of some variable in *p* by the same term.

2.2. THE THEOREM PROVER

The heuristics or techniques that the theorem prover employs to prove theorems support the use of induction. These heuristics include: Boolean simplification, tautology-checking, use of rewrite rules, a decision procedure for linear arithmetic, elimination of undesirable function symbols, generalization, careful type-checking, elimination of irrelevant hypotheses and induction.

The theorem prover is fully automatic, in the sense that once a purported lemma has been typed in, the user may not interfere with the mechanical proof. The user can however "train" the prover by proving an appropriate sequence of lemmas which can then be used as rewrite rules. In this manner, the theorem prover can be used as a high-level proof-checker.

The prover also has a simple hint facility by which the user can disable function definitions, suggest instances of lemmas to be used, and also suggest the induction to be employed. A nice feature of the prover is that it generates a cogent commentary on the proof being attempted. A careful examination of this commentary makes it easy to locate and correct mistakes in the statement of the proposed theorem.

The Boyer–Moore theorem prover has been used to prove theorems in number theory, recursive function theory and in program verification.

This concludes the discussion of the metatheory, the Boyer–Moore logic. Other details will be provided along the way.

3. The Formal Theory: Shoenfield's First Order Logic

The formal logic whose properties we wish to establish is Shoenfield's First Order Logic (SFOL). It has the advantage of being widely known and it is relatively simple and spare. In the following paragraphs, we provide a very brief description of SFOL, which is fully described Chapter 2 of Shoefield's *Mathematical Logic* [11].

3.1. THE LANGUAGE

The language of SFOL will be described by listing the symbols and the rules of syntax for forming expressions. The symbols in the language include variables: $x_1, \ldots, x_i, \ldots$; function symbols: $f_1, \ldots, f_n$; and predicate symbols: $P_1, \ldots, P_m$. Each function and predicate symbol has an arity associated with it. There is a special dyadic predicate symbol $=$, representing equality. The logic also contains the logical operators, $\neg$ and $\vee$, representing logical-not and logical-or respectively, and an existential quantifier $\exists$.

Expressions are formed by combining these symbols according to certain rules. A *term* is either a variable or an $\mathbf{n}$-ary function symbol followed by $\mathbf{n}$ terms. An *atomic formula* is an $\mathbf{n}$-ary predicate symbol followed by $\mathbf{n}$ terms. A *formula* is either an atomic formula; of the form $\neg \mathbf{A}$, where $\mathbf{A}$ is a formula; of the form $\vee \mathbf{AB}$, where $\mathbf{A}$ and $\mathbf{B}$ are formulas; or of the form $\exists \mathbf{xA}$, where $\mathbf{x}$ is a variable $\mathbf{A}$ is a formula.

We note certain assumptions regarding the use of meta-variables. The meta-variables $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$, range over formal variables; $\mathbf{f}$, $\mathbf{g}$, $\mathbf{h}$, range over function symbols; $\mathbf{p}$, $\mathbf{q}$, $\mathbf{r}$, range over predicate symbols; $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, range over terms; and $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, etc. range over formulas. From this point on, no specific variable, function or predicate symbols will appear in the text and meta-variables will be used to represent them.

An *elementary formula* is either an atomic formula or a formula of the form $\exists \mathbf{xA}$. The definitions of free and bound occurrences of a variable in a formula are well-known and will be omitted. $\mathbf{A_x[a]}$ denotes the result of replacing every free occurrences of $\mathbf{x}$ in $\mathbf{A}$ by $\mathbf{a}$. A term $\mathbf{a}$ is *substitutable* for $\mathbf{x}$ in $\mathbf{A}$, if and only if for each variable $\mathbf{y}$ in $\mathbf{a}$, no sub-formula of $\mathbf{A}$ of the form $\exists \mathbf{yB}$ contains an occurrence of $\mathbf{x}$ which is free in $\mathbf{A}$. All use of $\mathbf{A_x[a]}$ is restricted to when $\mathbf{a}$ is substitutible for $\mathbf{x}$ in $\mathbf{A}$. To increase readability, $\vee \mathbf{AB}$ will be replaced by $\mathbf{A} \vee \mathbf{B}$, and $= \mathbf{ab}$ by $\mathbf{a} = \mathbf{b}$. All operators will be assumed as being right-associative and parentheses will be introduced where needed. $\mathbf{A} \rightarrow \mathbf{B}$ is an abbreviation for $\neg \mathbf{A} \vee \mathbf{B}$.

3.2. THE AXIOMS

An *axiom* is one of the following:

1. A *propositional axiom*: Any formula of the form, $\neg \mathbf{A} \vee \mathbf{A}$.
2. A *substitution axiom*: Any formula of the form, $\mathbf{A}_{\mathbf{x}}[\mathbf{a}] \rightarrow \exists \mathbf{x} \mathbf{A}$.
3. An *identity axiom*: A formula of the form, $\mathbf{x} = \mathbf{x}$.
4. An *equality axiom for functions*: A formula of the form,

$$(\mathbf{x}_1 = \mathbf{y}_1 \rightarrow \ldots \rightarrow (\mathbf{x}_n = \mathbf{y}_n \rightarrow \mathbf{f}\mathbf{x}_1 \ldots \mathbf{x}_n = \mathbf{f}\mathbf{y}_1 \ldots \mathbf{y}_n) \ldots).$$

5. An *equality axiom for predicates*: Any formula of the form,

$$(\mathbf{x}_1 = \mathbf{y}_1 \rightarrow \ldots \rightarrow (\mathbf{x}_n = \mathbf{y}_n \rightarrow \mathbf{p}\mathbf{x}_1 \ldots \mathbf{x}_n \rightarrow \mathbf{p}\mathbf{y}_1 \ldots \mathbf{y}_n) \ldots).$$

3.3. THE RULES OF INFERENCE

The five rules of inference in SFOL are:

1. *Expansion rule*: Infer $\mathbf{B} \vee \mathbf{A}$ from $\mathbf{A}$.
2. *Contraction rule*: Infer $\mathbf{A}$ from $\mathbf{A} \vee \mathbf{A}$.
3. *Associative rule*: Infer $(\mathbf{A} \vee \mathbf{B}) \vee \mathbf{C}$ from $\mathbf{A} \vee (\mathbf{B} \vee \mathbf{C})$.
4. *Cut rule*: Infer $\mathbf{B} \vee \mathbf{C}$ from $\mathbf{A} \vee \mathbf{B}$ and $\neg \mathbf{A} \vee \mathbf{C}$.
5. $\exists$-*Introduction rule*: If $\mathbf{x}$ is not free in $\mathbf{B}$, infer $\exists \mathbf{x} \mathbf{A} \rightarrow \mathbf{B}$ from $\mathbf{A} \rightarrow \mathbf{B}$.

A *first order theory* may contain additional *non-logical axioms* e.g. axioms for natural numbers, but the language, logical axioms and the rules of inference remain as described above. This concludes the description of the formal theory SFOL.

4. The Informal Proof of the Tautology Theorem

In this section, we informally discuss the proof of the tautology theorem for SFOL. For this proof, we need only pay attention to the propositional part of the logic. This would mean that we need not attach any specific interpretation to quantified formulas or atomic formulas. Therefore, elementary formulas will be treated as *propositional atoms* (atoms, for short). *Boolean formulas* are constructed by combining atoms using the operators $\neg$ and $\vee$. It should be clear that any formula can be construed as a Boolean formula.

The proof consists of the following parts:

1. Definition of logical truth for Boolean formulas.
2. Definition of a tautology-checker.
3. The proof of a useful lemma which states that if one can prove the disjunction of a list of formulas, one can prove the disjunction of any list of formulas which contains them.

Logical truth is defined by the use of a truth-table. Given a Boolean formula, a *truth assignment* for that formula is a mapping from the set of propositional atoms to **T**,

F. The truth-table method for determining the *truth values* of a given Boolean formula is fairly well-known and we shall not go into its details. A *tautology* is defined as a Boolean formula whose truth value is **T** under all truth assignments. A *tautology-checker* is an algorithm that checks if a given Boolean formula is a tautology or not. We shall define one such tautology-checker. This tautology-checker works only on formulae of the form, $\mathbf{A}_1 \vee \ldots \vee \mathbf{A}_n$ where each $\mathbf{A}_i$ is termed a *disjunct*. Note that rearranging the disjuncts does not change the truth value, and that any formula **A** can be expressed in this form by simply setting $\mathbf{A}_1$ to be **A** and $\mathbf{n} = 1$. The recursive definition of the tautology-checker *TC*(**A**) is, as follows:

Base: (each $\mathbf{A}_i$ is either an atom or the negation of an atom)
If some $\mathbf{A}_j$ is the negation of some $\mathbf{A}_i$, $TC(\mathbf{A}) = \mathbf{T}$. Otherwise, $TC(\mathbf{A}) = \mathbf{F}$.

Recursive cases: (Some $\mathbf{A}_i$ is neither an atom nor the negation of an atom)

{Since the disjuncts can be rearranged without affecting the truth value, we can assume that $\mathbf{A}_1$ is such an $\mathbf{A}_i$. If this is the case, then $\mathbf{A}_1$ is either of the form $\mathbf{B} \vee \mathbf{C}$, the form $\neg(\mathbf{B} \vee \mathbf{C})$, or the form $\neg\neg\mathbf{B}$}
If $\mathbf{A}_1$ is of the form $\mathbf{B} \vee \mathbf{C}$: $TC(\mathbf{A}) = \mathbf{B} \vee \mathbf{C} \vee \ldots \vee \mathbf{A}_n)$ and
If $\mathbf{A}_1$ is of the form $\neg(\mathbf{B} \vee \mathbf{C})$: $TC(\mathbf{A}) = TC(\neg\mathbf{B} \vee \ldots \vee \mathbf{A}_n)$ and $TC(\neg\mathbf{C} \vee \ldots \vee \mathbf{A}_n)$.
If $\mathbf{A}_1$ is of the form $\neg\neg\mathbf{B}$: $TC(\mathbf{A}) = TC(\mathbf{B} \vee \ldots \vee \mathbf{A}_n)$.

Two things must be noted in the above definition of a tautology-checker:

1. The sum of the number of logical operators in the $\mathbf{A}_i$ decreases with each recursive call and therefore the algorithm always terminates.
2. In each recursive call, the truth value (with respect to any fixed truth assignment) of the formula recursed upon is the same as the truth value of the original formula. To show that the above tautology-checker is correct, we need to prove that $TC(\mathbf{A}) = \mathbf{T}$ if and only if **A** is a tautology. We can conclude that if $TC(\mathbf{A}) = \mathbf{T}$ then **A** is a tautology, by carrying out an inductive proof based on the recursion used in the tautology-checker. Showing that if **A** is a tautology, then $TC(\mathbf{A}) = \mathbf{T}$ is a little more difficult. The proof involves following the same induction to construct a truth assignment which makes the truth value of **A** equal to **F**, if $TC(\mathbf{A}) = \mathbf{F}$.

Now, we state the main theorem. $\vdash\mathbf{A}$ denotes '**A** is a theorem in SFOL'.

TAUTOLOGY THEOREM: *If* **A** *is a tautology, then* $\vdash\mathbf{A}$.

4.1. THE PROOF

We now describe an informal sketch of the proof of the Tautology Theorem, but omit most of the details. This sketch should serve as a useful guide to the mechanical proof presented in the following section.

Since **A** is a tautology if and only if $\mathbf{A} \vee \mathbf{A}$ is a tautology, and we can derive **A** from $\mathbf{A} \vee \mathbf{A}$ by the Contraction rule, we can restrict our attention to formulas of the form

$\mathbf{A}_1 \vee \ldots \vee \mathbf{A}_n$, where $\mathbf{n} > 1$. We now state without proof, a key lemma which is extremely useful in the proof.

A Lemma on disjuncts: *If* $\mathbf{A}_1, \ldots, \mathbf{A}_m$ *are all contained among* $\mathbf{B}_1, \ldots, \mathbf{B}_n$, *and if* $\vdash \mathbf{A}_1 \vee \ldots \vee \mathbf{A}_m$, *then* $\vdash \mathbf{B}_1 \vee \ldots \vee \mathbf{B}_n$.

We turn our attention now to the proof of the Tautology Theorem. The proof is by an induction that is identical to the recursion displayed by the tautology-checker. Since we have argued that the tautology-checker defined earlier is correct, we need only show that if the tautology-checker accepts a formula $\mathbf{A}$ of the form $\mathbf{A}_1 \vee \ldots \vee \mathbf{A}_n$ (where $\mathbf{n}$ is at least 2) as being a tautology, when we can construct a proof of it in SFOL. First, let us assume that the given formula is a tautology. Then the following cases arise:

1. **All $\mathbf{A}_i$ are atoms or negations of atoms:** By the definition of the tautology-checker, some $\mathbf{A}_j$ must be the negation of some $\mathbf{A}_i$. Then, $\vdash \mathbf{A}_j \vee \mathbf{A}_i$ (Propositional Axiom) from which we get $\vdash \mathbf{A}$ by applying the lemma on disjuncts.
2. **Some $\mathbf{A}_i$ is of the form $\mathbf{B} \vee \mathbf{C}$:** By rearranging the disjuncts using the Lemma on disjuncts, we can ensure that $\mathbf{i} = 1$. We have $\vdash \mathbf{B} \vee \mathbf{C} \vee \mathbf{A}_2 \vee \ldots \vee \mathbf{A}_n$ by the Induction hypothesis and the definition of the tautology-checker. From this we derive $\vdash \mathbf{A}$ by an application of the Associativity Rule.
3. **$\mathbf{A}_1$ is of the form $\neg\mathbf{B} \vee \mathbf{C}$:** Again, by examining the tautology-checker, we get, by the Induction Hypothesis: $\vdash \neg\mathbf{B} \vee \mathbf{A}_2 \vee \ldots \vee \mathbf{A}_n$ and $\vdash \neg\mathbf{C} \vee \mathbf{A}_2 \vee \ldots \vee \mathbf{A}_n$. We do not prove the lemma which allows us to derive $\vdash \mathbf{A}$ from these two formulas in the present discussion.
4. **$\mathbf{A}_1$ is of the form $\neg\neg\mathbf{B}$:** By the definition of the tautology-checker and the Induction Hypothesis, we have $\vdash \mathbf{B} \vee \mathbf{A}_2 \vee \ldots \vee \mathbf{A}_n$. The proof of the lemma which then allows us to derive $\vdash \mathbf{A}$ is also omitted from this discussion.

5 The Mechanical Proofs

In this section, we cover some of the highlights of the mechanical proof. The entire mechanical proof of the Tautology Theorem using the Boyer–Moore theorem prover consists of approximately 200 events (definitions or lemmas). These are listed in their entirety in the Appendix of an earlier technical report [10]. Many of the definitions and lemmas in the proof will be described in English. In some important cases, we will display the pure LISP version so that the careful reader can check whether these correspond exactly to the definitions and theorems in Sections 3 and 4. We remind the reader that a term of the form `(FN X1 . . . Xn)`, denotes the application of function `FN` to the *n* arguments, `X1` to `Xn`. The proof can roughly be divided into the following parts:

1. The definition of a proof-checker for SFOL.
2. The proof of the lemma on disjuncts.
3. The definition of the tautology-checker.

4. The proof of the Tautology Theorem.
5. The proof of correctness of the tautology-checker.

5.1. DEFINING THE PROOF-CHECKER

In this section we present the important definitions in the description of the SFOL proof-checker. The proof-checker corresponds closely to the formal theory described earlier. The basic Boyer–Moore prover contains only heuristics and contains no facts about lists or numbers [2]. The axioms for literal atoms, natural numbers and lists are loaded in by the event:

```
1.    (BOOT-STRAP)
```

We describe in English, the recognizers that were defined for the various classes of symbols:

(VARIABLE X): X is a variable iff it is a pair of the symbol 'X and an index number.

(FUNCTION FN): FN is a function symbol iff it is a triple of the symbol 'F, an index number and an arity number.

(PREDICATE PR): PR is a predicate symbol iff it is a triple of the symbol 'P, an index number and an arity number or the equality symbol 'EQUAL.

The INDEX of a symbol returns its subscript, and DEGREE returns the arity of a function of predicate symbol. The use of these metatheoretic definitions will be clarified in the descriptions that follow. The symbols $\neg$, $\vee$, $=$ and $\exists$ are represented by 'NOT, 'OR, 'EQUAL and 'FORSOME respectively.

The definition of TERMP displayed below is used to recognize EXP as either being a term, if FLG = T, or as being a list of terms of length COUNT, otherwise. The use of the flag FLG obviates the need for two mutually recursive definitions and will be used in other definitions as well. Informally, the definition asserts that EXP is a term if and only if EXP is non-empty and is either a variable, or a function symbol followed by a list of terms whose length is equal to the arity of the function symbol. In the case when FLG $\neq$ T (FLG is usually set to 'LIST in this case), EXP is a list of non-zero COUNT terms iff its first element is a term and the rest of the elements form a list of COUNT-1 terms.

```
40.  Definition.
     (TERMP EXP FLG COUNT)
        =
     (IF (EQUAL FLG T)
         (IF (NLISTP EXP)
             F
             (OR (VARIABLE EXP)
                 (AND (FUNCTION (CAR EXP))
                      (TERMP (CDR EXP)
                             'LIST
                             (DEGREE (CAR EXP))))))
        (IF (OR (NLISTP EXP) (ZEROP COUNT))
            (AND (EQUAL EXP NIL) (ZEROP COUNT))
            (AND (TERMP (CAR EXP) T 0)
                 (TERMP (CDR EXP)
                        'LIST
                        (SUB1 COUNT)))))
```

The next definition displayed below captures the notion of a formula/list of formulas. (ATOMP EXP) indicates that EXP is an atomic formula, i.e. a predicate symbol followed by a list of terms, of length equal to its arity. If FLG = T, EXP is a formula iff EXP is an atomic formula; or is 'NOT followed by one formula; or 'OR followed by two formulas; or 'FORSOME followed by a variable and a formula. If FLG $\neq$ T, then either EXP is an empty list of formulas and COUNT is zero, or COUNT is non-zero and EXP consists of a formula followed by COUNT-1 formulas. Note that the argument COUNT is irrelevant in the FLG = T case and hence we adopt the convention of setting it to zero.

```
45.  Definition.
     (FORMULA EXP FLG COUNT)
        =
     (IF (EQUAL FLG T)
         (IF (NLISTP EXP)
             F
             (OR (ATOMP EXP)
                 (AND (EQUAL (CAR EXP) 'NOT)
                      (FORMULA (CDR EXP) 'LIST 1))
                 (AND (EQUAL (CAR EXP) 'OR)
                      (FORMULA (CDR EXP) 'LIST 2))
                 (AND (EQUAL (CAR EXP) 'FORSOME)
                      (VARIABLE (CADR EXP))
                      (FORMULA (CDDR EXP) 'LIST 1))))
         (IF (OR (NLISTP EXP) (ZEROP COUNT))
             (AND (EQUAL EXP NIL) (ZEROP COUNT))
             (AND (FORMULA (CAR EXP) T 0)
                  (FORMULA (CDR EXP)
                           'LIST
                           (SUB1 COUNT)))))
```

Some other important definitions are:

(COLLECT-FREE EXP FLG) : which returns a list of all and only those variables that have free occurrences in EXP, with FLG used as before.

(COVERING EXP VAR FLG) : which returns a list of bound variables in EXP such that for each of these variables, say y, there is some sub-expression of EXP of the form $\exists y A$, such that EXP contains a free occurrence of the variable VAR.

(FREE-FOR EXP VAR TERM FLG) : which checks if TERM is substitutible for VAR in EXP, i.e. if (COVERING EXP VAR FLG) and (COLLECT-FREE TERM T) have an empty intersection

Now we introduce abbreviations for certain operations in the formal logic:

```
30.  Definition.
     (F-EQUAL X Y)
        =
     (LIST 'EQUAL X Y)
31.  Definition.
     (F-NOT X)
        =
     (LIST 'NOT X)
32.  Definition.
     (F-OR X Y)
        =
     (LIST 'OR X Y)
33.  Definition.
     (FORSOME X Y)
        =
     (LIST 'FORSOME X Y)
34.  Definition.
     (F-AND X Y)
        =
     (F-NOT (F-OR (F-NOT X) (F-NOT Y)))
35.  Definition.
     (F-IMPLIES X Y)
        =
     (F-OR (F-NOT X) Y)
```

Substitution of a term TERM for a free variable VAR in an expression EXP is one of the most important opertions in any formal system. It is also the easiest to get wrong. The recursive definition below is fairly subtle and requires careful study. The cases in the definition can be explained as follows:
If EXP is empty, return EXP itself.
If FLG = T, the following cases arise:

1. EXP is a variable and (EXP = VAR): Return TERM.
2. EXP is a variable and (EXP $\neq$ VAR): Return EXP.
3. EXP is of the form $\exists$xA, VAR = x: Return EXP.
4. EXP is of the form $\exists$xA, VAR $\neq$ x: Return $\exists$xA$'$, where A$'$ is the result of substituting TERM for VAR in A.
5. EXP is of the form $\mathbf{uu}_1 \ldots \mathbf{u}_n$, where **u** is either a predicate symbol, function symbol, or logical operator of arity n: Return $\mathbf{uu}'_1 \ldots \mathbf{u}'_1$, where $\mathbf{u}'_i$ is the result of substituting TERM for VAR in $\mathbf{u}_i$.
6. Otherwise: the expression is not well-formed and SUBST returns EXP itself.

In the case when FLG $\neq$ T EXP is a list of expressions and we perform the substitution on each member of EXP.

```
46.  Definition.
     (SUBST EXP VAR TERM FLG)
        =
     (IF (LISTP EXP)
         (IF (EQUAL FLG T)
             (IF (VARIABLE EXP)
                (IF (EQUAL EXP VAR) TERM EXP)
                (IF (AND (QUANTIFIER (CAR EXP))
                         (LISTP (CDR EXP)))
                    (IF (EQUAL (CADR EXP) VAR)
                        EXP
                        (CONS (CAR EXP)
                              (CONS (CADR EXP)
                                    (SUBST (CDDR EXP) VAR
                                           TERM 'LIST))))
                    (IF (OR (FUNC-PRED (CAR EXP))
                            (EQUAL (CAR EXP) 'NOT)
                            (EQUAL (CAR EXP) 'OR))
                        (CONS (CAR EXP)
                              (SUBST (CDR EXP) VAR TERM 'LIST))
                        EXP)))
        (CONS (SUBST (CAR EXP) VAR TERM T)
              (SUBST (CDR EXP) VAR TERM 'LIST)))
    EXP)
```

We now describe the SFOL proof-checker. We omit the definitions of several functions used to construct axioms and formal proofs corresponding to the rules and axioms of SFOL. Function names with the suffix PROOF construct formal proofs and will be called *proof-constructors*. The data-structure by which we represent formal proofs is a 4-tuple. The first element, (CAR PF), of this 4-tuple indicates the type of the final inference step; the second element is a sequence of hints (HINT1 PF), (HINT2 PF), (HINT3 PF) and(HINT4 PF); the third element is the conclusion of the proof and the fourth element, (SUB-PROOF PF) is a sub-proof or a list of sub-proofs leading to the final step. The function (CONC PF FLG) returns the conclusion/list of conclusions given a proof/list of proofs PF and a flag FLG. The skeletal control structure of the proof-checker will be presented before presenting the code for individual cases. (PRF PF) checks if PF is a correct formal proof. The comments within curly brackets in the skeletal code of the proof-checker indicate code to be presented later. Informally, if PF is empty, PRF returns F since it cannot prove anything. Otherwise, it checks if the first element of PF is either 'AXIOM or 'RULE corresponding to the cases when PF is the proof of a logical axiom, or involves an application of an inference rule in the final step, respectively. If it is none of the above, PRF returns F.

```
74.  Definition.
     (PRF PF)
        =
     (IF (NLISTP PF)
         F
         (IF (EQUAL (CAR PF) 'AXIOM)
             {code for checking proofs of axioms}
             (IF (EQUAL (CAR PF) 'RULE)
                 {code for checking proofs in which an inference
                  rule is used to derive the conclusion}
             F)))
```

5.1.1. *The Logical Axiom Case*

We present a similar skeletal control structure for the 'AXIOM case of the proof--checker PRF. In this section of the code, PRF checks the first member of the list of hints which form the second element of PF, i.e. (HINT1 PF), to see if it is one of 'PROP-AXIOM, 'SUBST-AXIOM , 'IDENT-AXIOM, 'EQUAL-AXIOM1 or 'EQUAL-AXIOM2. These correspond to the five types of axioms listed in Section 3.

```
(IF (EQUAL (HINT1 PF) 'PROP-AXIOM)
    {code for checking proofs of propositional axioms}
    (IF (EQUAL (HINT1 PF) 'SUBST-AXIOM)
        {code for checking proofs of substitution axioms}
        (IF (EQUAL (HINT1 PF) 'IDENT-AXIOM)
            {code for checking proofs of identity axioms}
            (IF (EQUAL (HINT1 PF) 'EQUAL-AXIOM1)
                {code for checking proofs of equality
                 axioms for functions}
                (IF (EQUAL (HINT1 PF) 'EQUAL-AXIOM2)
                    {code for checking proofs of equality
                     axioms for predicates}
                    F)))))
```

Now we fill in the code corresponding to each type of logical axiom. The terms (HINT2 PF), (HINT3 PF) and (HINT4 PF) will be referred to as the first, second and third hints, respectively. To check if PF is the proof of a proposition axiom, i.e. a formula of the form $\neg A \vee A$, PRF checks if the first hint **A** ((HINT2 PF)) is a formula and if PF is equal to (PROP-AXIOM-PROOF (HINT2 PF)), where PROP-AXIOM-PROOF constructs the required formal proof.

```
(AND (FORMULA (HINT2 PF) T 0)
     (EQUAL PF
            (PROP-AXIOM-PROOF (HINT2 PF))))
```

To check a given proof of a Substitution axiom three hints, (HINT2 PF), (HINT3 PF) and (HINT4 PF) below, are used. The code below checks if the first hint is a formula, the second hint is a variable, and if the third hint is a term. The fourth clause checks if the term is substitutible for the variable in the formula using the function FREE-FOR. The final clause checks if PF is the appropriate formal proof of a Substitution axiom given the above three hints.

```
(AND (FORMULA (HINT2 PF) T 0)
     (VARIABLE (HINT3 PF))
     (TERMP (HINT4 PF) T 0)
     (FREE-FOR (HINT2 PF)
               (HINT3 PF)
               (HINT4 PF)
               T)
     (EQUAL PF
            (SUBST-AXIOM-PROOF (HINT2 PF)
                               (HINT3 PF)
                               (HINT4 PF))))
```

The code for the identity axiom case checks if the first hint is a variable, say **x**, and checks if PF is the correct formal proof for the formal **x** = **x**.

```
(AND (VARIABLE (HINT2 PF))
     (EQUAL PF
            (IDENT-AXIOM-PROOF (HINT2 PF))))
```

The first hint in a given proof of an Equality axiom for functions is a function symbol, the second and third hints are two lists of variables of equal length. The function VARL–IST checks if the first argument is a list of variables of length given by the second argument. Then PRF checks if PF is the appropriate proof with respect to the above three hints.

```
(AND (FUNCTION (HINT2 PF))
     (VAR-LIST (HINT3 PF)
               (DEGREE (HINT2 PF)))
     (VAR-LIST (HINT4 PF)
               (DEGREE (HINT2 PF)))
     (EQUAL PF
            (EQUAL-AXIOM1-PROOF (HINT2 PF)
                                (HINT3 PF)
                                (HINT4 PF))))
```

The given proof of an Equality axioms for predicates is checked similarly. The only difference is that it now checks if the first hint is a predicate symbol.

```
(AND (PREDICATE (HINT2 PF))
     (VAR-LIST (HINT3 PF)
               (DEGREE (HINT2 PF)))
     (VAR-LIST (HINT4 PF)
               (DEGREE (HINT2 PF)))
     (EQUAL PF
            (EQUAL-AXIOM2-PROOF
              (HINT2 PF)
              (HINT3 PF)
              (HINT4 PF))))
```

This concludes the description of the 'AXIOM case of the definition of the SFOL proof-checker.

5.1.2. *The Inference Rules*

The second group of cases deals with the inference rules of SFOL viz. Expansion, Contraction, Associativity, Cut and ∃-Introduction. The rule that is used to derive the conclusion is supplied as (HINT1 PF). The control skeleton used to branch on this hint is similar to the one used to check proofs of axioms and is displayed below.

```
(IF (EQUAL (HINT1 PF) 'EXPAN)
    {code for checking Expansion step}
    (IF (EQUAL (HINT1 PF) 'CONTRAC)
        {code for checking Contraction step}
        (IF (EQUAL (HINT1 PF) 'ASSOC)
            {code for checking Associativity step}
            (IF (EQUAL (HINT1 PF) 'CUT)
                {code for checking Cut step}
                (IF (EQUAL (HINT1 PF) 'E-INTRO)
                    {code for checking ∃-Introduction step}
                    F)))))
```

Now we deal with the code for each individual case. A part of the checking is done within the proof-constructor functions and those details will not appear in the description below. Since an expression appearing as a proper part of a proven formula is also a formula and in such cases, we do not explicitly check if the expression is a formula.

To check an application of an Expansion step i.e. a derivation of **A** ∨ **B** from **B**, **A** and **B** are supplied as the two hints. Then the code checks if **A** is a formula, if PF is the appropriate proof of **A** ∨ **B**, and if (SUB-PROOF PF) is a proof of **B**.

```
        (AND (FORMULA (HINT2 PF) T 0)
             (EQUAL PF
                    (EXPAN-PROOF (HINT2 PF)
                                 (HINT3 PF)
                                 (SUB-PROOF PF)))
             (EQUAL (CONC (SUB-PROOF PF) T)
                    (HINT3 PF))
             (PRF (SUB-PROOF PF)))
```

In order to check an application of a Contraction step, i.e. a derivation of **A** from **A** ∨ **A**, the only hint supplied is **A**. We then check if PF is a properly constructed proof and if (SUB-PROOF PF) is a correct proof of **A** ∨ **A**.

```
        (AND (EQUAL PF
                    (CONTRAC-PROOF (HINT2 PF)
                                   (SUB-PROOF PF)))
             (EQUAL (CONC (SUB-PROOF PF) T)
                    (F-OR (HINT2 PF) (HINT2 PF)))
             (PRF (SUB-PROOF PF)))
```

The Associativity rule is used to derive (**A** ∨ **B**) ∨ **C** from **A** ∨ (**B** ∨ **C**) and **A**, **B**, and **C** are the three hints used in checking this. The code below checks if PF is a properly constructed proof using this inference step, and if(SUB-PROOF PF)is a correct proof of **A** ∨ (**B** ∨ **C**).

```
        (AND (EQUAL PF
                    (ASSOC-PROOF (HINT2 PF)
                                 (HINT3 PF)
                                 (HINT4 PF)
                                 (SUB-PROOF PF)))
             (EQUAL (CONC (SUB-PROOF PF) T)
                    (F-OR (HINT2 PF)
                          (F-OR (HINT3 PF) (HINT4 PF))))
             (PRF (SUB-PROOF PF)))
```

The Cut rule is employed to derive **B** ∨ **C** from **A** ∨ **B** and ¬**A** ∨ **C**. The three hints used are **A**, **B** and **C**. The code checks if PF is the appropriate proof and if (CAR (SUB-PROOF PF)) and (CADR (SUB-PROOF PF)) are the correct proofs of **A** ∨ **B** and ¬**A** ∨ **C**, respectively.

```
        (AND (EQUAL PF
                    (CUT-PROOF (HINT2 PF)
                               (HINT3 PF)
                               (HINT4 PF)
                               (CAR (SUB-PROOF PF))
                               (CADR (SUB-PROOF PF))))
             (EQUAL (CONC (SUB-PROOF PF) 'LIST)
                    (LIST (F-OR (HINT2 PF) (HINT3 PF))
                          (F-OR (F-NOT (HINT2 PF))
                                (HINT4 PF))))
             (PRF (CAR (SUB-PROOF PF)))
             (PRF (CADR (SUB-PROOF PF))))
```

The three hints used in checking the ∃-Introduction step in a proof are **x**, **A**, and **B**. The code checks if **x** is a variable, and if PF is a correctly constructed proof of

$\exists x\mathbf{A} \rightarrow \mathbf{B}$, where $\mathbf{x}$ is not a member of the list of variables appearing free in $\mathbf{B}$ and (SUB-PROOF PF) is a correct proof of $\mathbf{A} \rightarrow \mathbf{B}$.

```
(AND (VARIABLE (HINT2 PF))
     (EQUAL PF
            (FORSOME-INTRO-PROOF (HINT2 PF)
                                 (HINT3 PF)
                                 (HINT4 PF)
                                 (SUB-PROOF PF)))
     (NOT (MEMBER (HINT2 PF)
                  (COLLECT-FREE (HINT4 PF) T)))
     (EQUAL (CONC (SUB-PROOF PF) T)
            (F-IMPLIES (HINT3 PF) (HINT4 PF)))
     (PRF (SUB-PROOF PF)))
```

This completes the description of our implementation of a proof-checker for SFOL. The function PROVES defined below is a more usable form of the proof-checker.

```
78.  Definition.
     (PROVES PF EXP)
        =
     (AND (EQUAL (CONC PF T) EXP)
          (FORMULA EXP T 0)
          (PRF PF))
```

PROVES checks if PRF is a valid proof of EXP. This definition also presents a good example of 'cheating' in a proof. By 'cheating' we mean the avoidance of theorem proving in favor of computation. If we had proved that the conclusion of a valid formal proof is always a formula, the FORMULA clause in the definition of PROVES would have been unnecessary. This turns out not to matter since the body of PRF is replaced by a group of rewrite rules in which the redundant use of FORMULA is avoided. Thus, the 'cheating' here turns out to be a prudent decision.

This concludes the description of the SFOL proof-checker as defined relative to the Boyer–Moore theorem prover.

5.2. STEPS TO THE PROOF OF THE TAUTOLOGY THEOREM

In this section, we list some of the important lemmas involved in the proof of the Tautology Theorem. We also provide a glimpse of the approach that we have adopted in interacting with the prover. This is worth noting since the success of a mechanical proof effort depends quite heavily on the approach used. Immediately after defining the proof-checker, we replaced its definition by a series of rewrite rules. This is because the definition of PRF is long and this causes a great deal of garbage generation/-collection during the proof. In most of the lemmas that we prove, only a small part of the definition of the proof-checker is relevant. We provide only one simple example of a rewrite rule that captures one case of the definition of PRF, the one dealing with propositional axioms. The other cases are similar. The lemma PROP-AXIOM-PROVES attempts to rewrite any term in a proof of the form (PROVES (PROP-AXIOM-PROOF expression) conclusion) to T, if expression is a formula and conclusion is a propositional axiom involving expression. Thus, if PROP-AXIOM-PROOF had been used in a proof, this lemma would be invoked in the course of checking that proof. It is important to note that once we have this lemma, the actual definition of PROP-AXIOM-PROOF is no longer useful. The definition of this proof-constructor is disabled so that the prover does not expand an occurrence of PROP-AXIOM-PROOF into its definition during the proof of a theorem.

```
81.  Theorem.  PROP-AXIOM-PROVES (rewrite):
     (IMPLIES (AND (FORMULA EXP T 0)
                   (EQUAL CONCL (F-OR (F-NOT EXP) EXP)))
              (PROVES (PROP-AXIOM-PROOF EXP) CONCL))
```

At this point in the proof, all the primitive proof-constructors such as PROP-AXIOM-PROOF and the definitions of PRF and PROVES are disabled.

Now, we give the first example of the proof of soundness of a derived inference rule. This is the *Commutative Rule for logical-or*. This rule allows us to infer **B** ∨ **A** from a proof of **A** ∨ **B**. The first step in the proof is to define a proof-constructor COMMUT-PROOF corresponding to this rule.

```
103. Definition.
     (COMMUT-PROOF A B PF)
        =
     (CUT-PROOF A B A PF
                (PROP-AXIOM-PROOF A))
```

COMMUT-PROOF provides the formal justification for each application of the Commutative rule and this is given by the following lemma.

```
104. Theorem.  COMMUT-PROOF-PROVES (rewrite):
     (IMPLIES (AND (PROVES PF (F-OR A B))
                   (FORMULA (F-OR A B) T 0)
                   (EQUAL CONCL (F-OR B A)))
              (PROVES (COMMUT-PROOF A B PF) CONCL))
```

Now, the definition of COMMUT-PROOF is disabled as was done in the case of PROP-AXIOM-PROOF.

The next important step is the proof of the previously mentioned lemma on disjuncts. The proof of this lemma is an extremely difficult one and the reader is urged to read the informal exposition from Shoenfield's *Mathematical Logic* [11]. The mechanical proof of this lemma proceeds at roughly the same level of detail as the informal proof in Shoenfield's book. The lemma on disjuncts is an extremely derived inference rule. The lemma states: If $\mathbf{A}_1, \ldots, \mathbf{A}_m$ are all contained among $\mathbf{B}_1, \ldots, \mathbf{B}_n$, then we can infer $\mathbf{B}_1 \vee \ldots \vee \mathbf{B}_n$ from a proof of $\mathbf{A}_1 \vee \ldots \vee \mathbf{A}_m$. The proof is by strong induction on $\mathbf{m}$ with base cases for $[\mathbf{m} = 1]$ and $[\mathbf{m} = 2]$. First, we list some of the definitions used in the proof.

(MAKE-DISJUNCT FLIST) : Given a list of formulas FLIST, MAKE-DISJUNCT constructs the formula representing their disjunction.

(M1-PROOF EXP FLIST PF): This is the proof constructor in the $[\mathbf{m} = 1]$ case. If PF is a proof of EXP and EXP is a member of FLIST, then M1-PROOF constructs a proof of (MAKE-DISJUNCT FLIST).

(FORM-LIST FLIST): FORM-LIST checks if FLIST is a list of formulas.

(M2-PROOF EXP1 EXP2 FLIST PF): M2-PROOF is the proof-constructor in the [**m** = 2] case and constructs a proof of (MAKE-DISJUNCT FLIST), where EXP1 and EXP2 are members of FLIST and PF is a proof of (F-OR EXP1 EXP2).

(M-PROOF FLIST1 FLIST2 PF): M-PROOF constructs a proof of (MAKE-DISJUNCT FLIST2), where the list of formulas FLIST1 is contained in the list of formulas FLIST2 and PF is a proof of (MAKE-DISJUNCT FLIST1).

The lemma M1-PROOF-PROVES1 displayed below, expresses the [**m** = 1] the case of the proof.

```
122. Theorem.  M1-PROOF-PROVES1 (rewrite):
     (IMPLIES (AND (FORMULA (MAKE-DISJUNCT FLIST) T 0)
                   (MEMBER EXP FLIST)
                   (PROVES PF EXP))
              (PROVES (M1-PROOF EXP FLIST PF)
                      (MAKE-DISJUNCT FLIST)))
     Hint:  Disable FORMULA
```

The [**m** = 2] case of the proof is expressed by the lemma M2-PROOF-PROVES below. EXP1 and EXP2 are the two disjuncts that appear in FLIST.

```
137. Theorem.  M2-PROOF-PROVES (rewrite):
     (IMPLIES (AND (FORMULA (MAKE-DISJUNCT FLIST) T 0)
                   (FORMULA EXP1 T 0)
                   (FORMULA EXP2 T 0)
                   (MEMBER EXP1 FLIST)
                   (MEMBER EXP2 FLIST)
                   (PROVES PF (F-OR EXP1 EXP2)))
              (PROVES (M2-PROOF EXP1 EXP2 FLIST PF)
                      (MAKE-DISJUNCT FLIST)))
     Hint:  Disable FORMULA
```

Finally, M-PROOF-PROVES expresses the lemma on disjuncts. If FLIST1 is a list of disjuncts that are contained in FLIST2, and we have a proof of the disjunction of the disjuncts in FLIST1, then M-PROOF constructs a proof of the disjunction of the disjuncts in FLIST2. Note that the induction to be employed is supplied as a hint to the theorem prover.

```
150. Theorem.  M-PROOF-PROVES (rewrite):
     (IMPLIES (AND (FORM-LIST FLIST1)
                   (LISTP FLIST1)
                   (FORM-LIST FLIST2)
                   (LISTP FLIST2)
                   (SUBSET FLIST1 FLIST2)
                   (PROVES PF (MAKE-DISJUNCT FLIST1)))
              (PROVES (M-PROOF FLIST1 FLIST2 PF)
                      (MAKE-DISJUNCT FLIST2)))
     Hint:  Induct as for (M-PROOF FLIST1 FLIST2 PF).
```

The remainder of the description of the mechanical proof includes the definition of the tautology checker, the proof of the Tautology Theorem and the proof of the correctness of the tautology checker.

5.3. DEFINING THE TAUTOLOGY-CHECKER

The tautology-checker we define below is an implementation of the one described in Section Four. However, for efficiency reasons the tautology-checker below does not flatten out the entire given formula into disjunction of atoms or negations of atoms. Instead, it maintains a list of atoms and negations of atoms accumulated so far, and if this list ever contains an atom and its negation, we claim that the given formula is a tautology. A small amount of effort was expended in proving the admissibility of the tautology-checker in accordance with the Principle of Definition. The steps in the proof of admissibility will be omitted. As before, we describe the preliminary definitions in English and provide skeletal descriptions of the tautology-checker before going into the details. First, we define predicates which serve as recognizers for the various classes of formulas.

(PROP-ATOMP EXP) checks if EXP is an atom or the negation of an atom.

(OR-TYPE EXP) checks if EXP is of the form **A** $\vee$ **B**.

(NOR-TYPE EXP) checks if EXP is of the form $\neg$(**A** $\vee$ **B**).

(DBLE-NEG-TYPE EXP) checks if EXP is of the form $\neg\neg$**A**.

The function (LIST-COUNT FLIST) computes the measure based on which the tautology-checker is admitted. It takes as input a list and returns the sum of the sizes of all the elements. The size of each element is one more than the number of CONSes appearing in it.

(NEG-LIST EXP FLIST) checks if either EXP is the negation of some member of FLIST or if some member of FLIST is the negation of EXP.

Next, we examine the definition of the tautology-checker TAUTOLOGYP1 in detail. As in the case of the SFOL proof-checker, the pure-LISP definition will be annotated in English. The function TAUTOLOGYP1 takes two arguments, FLIST and AUXLIST. As mentioned earlier, if **A** is the formula being checked. **A** must be of the form $\mathbf{A}_1 \vee \ldots \vee \mathbf{A}_n$, and FLIST is the list $\mathbf{A}_1, \ldots, \mathbf{A}_n$. AUXLIST is an auxiliary argument that accumulates the atoms and negations of atoms encountered during the recursion of the tautology-checker. AUXLIST will have to be initially bound to NIL when invoking TAUTOLOGYP1. The control skeleton of TAUTOLOGYP1 is displayed below. If the FLIST is empty, we return F. Otherwise we check if the first element of FLIST is of one of the types: PROP-ATOMP, OR-TYPE, NOR-TYPE, or DBLE-NEG-TYPE, and branch off accordingly. Later on, we present a lemma which states that any formula must fall into one of the above types.

176. Definition.
```
    (TAUTOLOGYP1 FLIST AUXLIST)
        =
    (IF (NLISTP FLIST)
        F
        (IF (PROP-ATOMP (CAR FLIST))
            {code for case when the first element of FLIST
             is an atom or the negation of an atom}
            (IF (OR-TYPE (CAR FLIST))
                {code for case when the first element of FLIST is
                 of OR-TYPE}
                (IF (NOR-TYPE (CAR FLIST))
                    {code for case when the first element of FLIST
                     is of NOR-TYPE}
                    (IF (DBLE-NEG-TYPE (CAR FLIST))
                        {code for case when the first element of
                         FLIST is of DBLE-NEG-TYPE}
                        F)))))
    Hint: Consider the well-founded relation LESSP
          and the measure (LIST-COUNT FLIST)
```

We now turn to the individual cases of the above definition. The function (ARG1 EXP) returns **B** when given an expression EXP of the form $\neg$**B** or of the form **B** $\vee$ **C** and in the latter case, (ARG2 EXP) returns **C**. If the first element of FLIST is either an atom or the negation of an atom, TAUTOLOGYP1 first checks if its negation appears in AUXLIST and return T if that is the case. Otherwise, we add the first element of FLIST to the AUXLIST and recurse on the rest of the FLIST.

```
(OR (NEG-LIST (CAR FLIST) AUXLIST)
    (TAUTOLOGYP1 (CDR FLIST)
                 (CONS (CAR FLIST) AUXLIST)))
```

When the first element of FLIST is of OR-TYPE i.e. of the form **B** $\vee$ **C**, we add **B** and **C** to the rest of the FLIST and recurse with the AUXLIST unchanged.

```
(TAUTOLOGYP1 (CONS (ARG1 (CAR FLIST))
                   (CONS (ARG2 (CAR FLIST)) (CDR FLIST)))
             AUXLIST)
```

If the first element of FLIST is of the form $\neg$(**B** $\vee$ **C**), then TAUTOLOGYP1 is called twice, once with $\neg$**B** added to the rest of FLIST and another time with $\neg$**C** added to the rest of FLIST.

```
(AND (TAUTOLOGYP1 (CONS (F-NOT (ARG1
                                 (ARG1 (CAR FLIST))))
                        (CDR FLIST))
                  AUXLIST)
     (TAUTOLOGYP1 (CONS (F-NOT (ARG2
                                 (ARG1 (CAR FLIST))))
                        (CDR FLIST))
                  AUXLIST))
```

The only remaining possibility is that the first element could be of the form $\neg\neg$**B** in which case **B** is added to the rest of FLIST and a recursive call is made.

```
(TAUTOLOGYP1 (CONS (ARG1 (ARG1 (CAR FLIST)))
                   (CDR FLIST))
             AUXLIST)
```

This concludes the description of the tautology-checker for SFOL.

5.4. THE PROOF OF THE TAUTOLOGY THEOREM

In this section, we sketch some of the events leading to the proof of the statement that all tautologies have formal proofs within SFOL. The major task in the proof is to define the proof-constructor function which constructs formal proofs for those formulas on which the tautology-checker returns T. More accurately, the proof-constructor constructs a proof of (MAKE-DISJUNCT (APPEND FLIST AUXLIST)) where APPEND concatenates two lists, and MAKE-DISJUNCT returns the disjunction of a list of formulas. The case-structure and recursion scheme employed by the proof-constructor TAUTOLOGYP1 are identical to those of TAUT-PROOF1. The control skeleton of TAUT-PROOF1 is displayed below.

```
232. Definition.
     (TAUT-PROOF1 FLIST AUXLIST)
        =
     (IF (NLISTP FLIST)
         NIL
         (IF (PROP-ATOMP (CAR FLIST))
             {proof-constructor for PROP-ATOMP case}
             (IF (OR-TYPE (CAR FLIST))
                 {proof-constructor for OR-TYPE case}
                 (IF (NOR-TYPE (CAR FLIST))
                     {proof-constructor for NOR-TYPE case}
                     (IF (DBLE-NEG-TYPE (CAR FLIST))
                         {proof-constructor for DBLE-NEG-TYPE case}
                         NIL)))))
     Hint:  Consider the well-founded relation LESSP
            and the measure (LIST-COUNT FLIST)
```

The body of TAUT-PROOF1 makes calls to several other proof-constructors and we omit several lemmas which state that these functions construct the appropriate proofs.

In the PROP-ATOMP case, two possibilities arise depending on whether (NEG-LIST (CAR FLIST) AUXLIST) is T or not. If it is T, then PROP-ATOM-PROOF1 constructs the appropriate proof. If it is not T, then we recurse as in TAUTOLOGYP1 and PROP-ATOM-PROOF2 uses the proof constructed by the recursive call to construct the required final proof.

```
(IF (NEG-LIST (CAR FLIST) AUXLIST)
    (PROP-ATOM-PROOF1 FLIST AUXLIST)
    (PROP-ATOM-PROOF2 FLIST AUXLIST
                      (TAUT-PROOF1 (CDR FLIST)
                                   (CONS (CAR FLIST)
                                         AUXLIST))))
```

In the OR-TYPE case, OR-TYPE-PROOF constructs the proof of the disjunction of the formulas in FLIST and AUXLIST using the proof generated by the recursive call to TAUT-PROOF1.

```
(OR-TYPE-PROOF
    (ARG1 (CAR FLIST))
    (ARG2 (CAR FLIST))
    (APPEND (CDR FLIST) AUXLIST)
    (TAUT-PROOF1 (CONS (ARG1 (CAR FLIST))
                       (CONS (ARG2 (CAR FLIST))
                             (CDR FLIST)))
                 AUXLIST))
```

NOR-TYPE-PROOF constructs the proof in the NOR-TYPE case but this time there are two recursive calls to TAUT-PROOF1 as is also the case in TAUTOLOGYP1.

```
        (NOR-TYPE-PROOF
               (ARG1 (ARG1 (CAR FLIST)))
               (ARG2 (ARG1 (CAR FLIST)))
               (APPEND (CDR FLIST) AUXLIST)
               (TAUT-PROOF1 (CONS (F-NOT (ARG1
                                           (ARG1 (CAR FLIST))))
                                  (CDR FLIST))
                            AUXLIST)
               (TAUT-PROOF1 (CONS (F-NOT (ARG2
                                           (ARG1 (CAR FLIST))))
                                  (CDR FLIST))
                            AUXLIST))
```

Finally, in the DBLE-NEG-TYPE case, DBLE-NEG-TYPE-PROOF is used to construct the required proof from the proof generated by the recursive call to TAUT-PROOF1.

```
        (DBLE-NEG-TYPE-PROOF
                    (ARG1 (ARG1 (CAR FLIST)))
                    (APPEND (CDR FLIST) AUXLIST)
                    (TAUT-PROOF1 (CONS (ARG1
                                         (ARG1 (CAR FLIST)))
                                       (CDR FLIST))
                                 AUXLIST))
```

We now state the theorem which asserts that TAUT-PROOF1 constructs a correct proof of (MAKE-DISJUNCT (APPEND FLIST AUXLIST)) if (TAUTOLOGYP1 FLIST AUXLIST) is T and both FLIST and AUXLIST are lists of formulas. (FORM-LIST FLIST) checks if FLIST is a (possibly empty) list of formulas.

```
233. Theorem.  TAUT-THM1 (rewrite):
     (IMPLIES (AND (FORM-LIST FLIST)
                   (FORM-LIST AUXLIST)
                   (TAUTOLOGYP1 FLIST AUXLIST))
              (PROVES (TAUT-PROOF1 FLIST AUXLIST)
                      (MAKE-DISJUNCT (APPEND FLIST AUXLIST))))
     Hints:  Disable NEG-LIST-REDUC and FORMULA
             Induct as for (TAUTOLOGYP1 FLIST AUXLIST).
```

The theorem TAUT-THM1 captures the statement of the Tautology Theorem when AUXLIST is instantiated with NIL.

5.5. THE PROOF OF THE CORRECTNESS OF THE TAUTOLOGY-CHECKER

The final part of the proof consists in showing that tautology-checker TAUTOLOGYP1 corresponds to the truth-table definition of a tautology. Boyer and Moore [2] have carried out a similar proof of the correctness of a tautology-checker for IF-expressions. To prove the correctness of the above tautology-checker, we need to:

1. Define the notion of the logical truth of a formula by definining a function which evaluates the truth value of a formula with respect to a given truth assignment.
2. Using the above function, show that if TAUTOLOGYP1 asserts a given formula to be a tautology, then the truth value of that formula under any truth assignment is always **T**.

3. Prove that if TAUTOLOGYP1 claims that the given formula is not a tautology, a falsifying truth assignment exists, i.e. an assignment under which the truth value of the given formula is **F**.

The function EVAL below evaluates the truth value of the formula EXP with respect to the truth assignment ALIST and returns T or F accordingly. (ELEM-FORM EXP) checks if EXP is an atom. EVAL works as follows:

```
If EXP is an atom:
    Return T if EXP is a member of ALIST and F otherwise.
If EXP is of the form ¬A:
    Return the negation of the truth value of a on ALIST.
If EXP is of the form A ∨ B:
    Return T if at least one of A and B evaluates to T on ALIST and F otherwise.
If it is none of the above, EXP is not well-formed.
```

```
237. Definition.
     (EVAL EXP ALIST)
        =
     (IF (NLISTP EXP)
         F
         (IF (ELEM-FORM EXP)
             (MEMBER EXP ALIST)
             (IF (EQUAL (CAR EXP) 'NOT)
                 (NOT (EVAL (CADR EXP) ALIST))
                 (IF (EQUAL (CAR EXP) 'OR)
                     (OR (EVAL (CADR EXP) ALIST)
                         (EVAL (CADDR EXP) ALIST))
                     F))))
```

Having defined EVAL, we can state and prove the other two statements in the proof of correctness of TAUTOLOGYP1. The theorem TAUT-EVAL states that if (TAUTOLOGYP1 FLIST AUXLIST) is T then EVAL on (MAKE-DISJUNCT (APPEND FLIST AUXLIST)) returns T on any ALIST.

```
257. Theorem.  TAUT-EVAL (rewrite):
     (IMPLIES (TAUTOLOGYP1 FLIST AUXLIST)
              (EVAL (MAKE-DISJUNCT (APPEND FLIST AUXLIST))
                    ALIST))
     Hints:  Disable EVAL, EVAL-MAKE-DISJUNCT, ELEM-FORM,
                     PROP-ATOMP, OR-TYPE, NOR-TYPE, DBLE-NEG-TYPE,
                     APPEND, and NEG-LIST-REDUC
             Induct as for (TAUTOLOGYP1 FLIST AUXLIST).
```

Note that it is only in the proof of the statement that all non-TAUTOLOGYP1s are falsifiable do we need to prove that every formula is of one of the types: PROP-ATOMP, OR-TYPE, NOR-TYPE or DBLE-NEG-TYPE. This is state below as FORMULA-CASES1.

```
271. Theorem.  FORMULA-CASES1:
     (IMPLIES (FORMULA EXP T 0)
              (OR (PROP-ATOMP EXP)
                  (OR-TYPE EXP)
                  (NOR-TYPE EXP)
                  (DBLE-NEG-TYPE EXP)))
```

(FALSIFY-TAUT FLIST AUXLIST) constructs the truth assignment which falsifies (MAKE-DISJUNCT (APPEND FLIST AUXLIST)) when (TAUTOLOGYP1 FLIST AUXLIST) is F and its definition is similar to that of TAUTOLOGYP1. We state this as NON-TAUT-FALSE below. We restrict AUXLIST to being a list of propositional atoms whose disjunction is not by itself a tautology, but since we are only interested in the instance when the AUXLIST is NIL, this turns out not to matter. (FALSIFY AUXLIST) returns the truth assignment which falsifies (MAKE-DISJUNCT AUXLIST) when AUXLIST is a list of propositional atoms.

```
281. Theorem.  NOT-TAUT-FALSE (rewrite):
     (IMPLIES (AND (FORM-LIST FLIST)
                   (PROP-ATOMP-LIST AUXLIST)
                   (NOT (EVAL (MAKE-DISJUNCT AUXLIST)
                              (FALSIFY AUXLIST)))
                   (NOT (TAUTOLOGYP1 FLIST AUXLIST)))
              (NOT (EVAL (MAKE-DISJUNCT (APPEND FLIST AUXLIST))
                         (FALSIFY-TAUT FLIST AUXLIST))))
     Hints:  Induct as for (FALSIFY-TAUT FLIST AUXLIST).
             Disable NEG-LIST, EVAL-MAKE-DISJUNCT, NEG-LIST-REDUC,
                   PROP-ATOMP-REDUC, FORMULA, FALSIFY, APPEND, and
                   NOR-TYPE
```

Finally, we replace AUXLIST by NIL and derive more readable versions of the above theorems.

```
282. Definition.
     (TAUTOLOGYP FLIST)
        =
     (TAUTOLOGYP1 FLIST NIL)

283. Definition.
     (TAUT-PROOF FLIST)
        =
     (TAUT-PROOF1 FLIST NIL)

286. Theorem.  TAUTOLOGY-THEOREM (rewrite):
     (IMPLIES (AND (FORM-LIST FLIST)
                   (TAUTOLOGYP FLIST)
                   (EQUAL CONCL (MAKE-DISJUNCT FLIST)))
              (PROVES (TAUT-PROOF FLIST) CONCL))
     Hint:  Disable TAUT-PROOF1, TAUTOLOGYP1, FORMULA, and
                   NOT-FALSIFY-TAUT

288. Theorem.  TAUTOLOGIES-ARE-TRUE (rewrite):
     (IMPLIES (AND (FORM-LIST FLIST)
                   (TAUTOLOGYP FLIST))
              (EVAL (MAKE-DISJUNCT FLIST) ALIST))
     Hint:  Disable FORMULA, TAUTOLOGYP1, and NOT-FALSIFY-TAUT

290. Theorem.  TRUTHS-ARE-TAUTOLOGIES (rewrite):
     (IMPLIES (AND (FORM-LIST FLIST)
                   (NOT (TAUTOLOGYP FLIST)))
              (NOT (EVAL (MAKE-DISJUNCT FLIST)
                         (FALSIFY-TAUT FLIST NIL))))
     Hint:  Disable TAUTOLOGYP1, NOT-FALSIFY-TAUT, and FORMULA
```

5.6. A POST-SCRIPT

The main motivation for proving the Tautology Theorem was that it could then be applied to simplify some of the formal deduction steps in the metatheorems that were to follow. As it turned out, it was not directly usable since all our applications involve the use of meta-variables, i.e. variables in the Boyer–Moore logic, and the tautology-checker can only handle SFOL expressions. For instance, if we want to show that (F-OR (F-NOT A) A) is a tautology for any formula A, we cannot directly apply the tautology-checker without instantiating A. The Tautology Theorem was rendered useful by the contrapositive version of TRUTHS-ARE-TAUTOLOGIES displayed below as EVAL-TAUTOLOGYP. Since EVAL translates formal disjunctions and negations into disjunctions and negations in the Boyer–Moore logic, we can use it to translate tautologies containing meta-variables into statements which are tautologously true in the Boyer–Moore Logic. Now, in order to establish (TAUTOLOGYP FLIST), the theorem prover tries to prove that (EVAL (MAKE-DISJUNCT FLIST) (FALSIFY-TAUT FLIST NIL)) is tautologously true.

```
318. Theorem.  EVAL-TAUTOLOGYP (rewrite):
     (IMPLIES (AND (FORM-LIST FLIST)
                   (EVAL (MAKE-DISJUNCT FLIST)
                         (FALSIFY-TAUT FLIST NIL)))
              (TAUTOLOGYP FLIST))
     Hints:  Disable TAUTOLOGYP, FORM-LIST, and FALSIFY-TAUT
             Consider:
              TRUTHS-ARE-TAUTOLOGIES
             Enable TRUTHS-ARE-TAUTOLOGIES
```

6. Conclusions

This paper describes a project aimed at mechanizing proofs in metamathematics using the Boyer–Moore Theorem Prover [2, 3]. To this end, a proof-checker for Shoenfield's First Order Logic (SFOL) [11] was defined as a function in the Boyer–Moore logic. The theorem prover was then used to prove the tautology theorem for SFOL. The success of this proof effort leads us to believe that a significant part of proof-theoretic metamethematics can be mechanically proof-checked using the Boyer–Moore theorem prover. These mechanical proofs also demonstrate a method for making sound extensions to automatic proof-checkers. Such proofs make it possible to write correct formal proofs without laying out in tedious detail, all of the steps involved. This leads to a significant speed-up★ in the automatic checking of proofs and at the same time makes it more convenient for a human to construct proofs that will be automatically checked.

The proof was done by first writing up a list of events before attempting the mechanical proofs. This took about 4 weeks. It took about 3 or 4 weeks to complete a mechanical proof of the Tautology theorem. Only a few changes were made to the original outline of the proof. Since then, some revisions have been made to the

★ Some preliminary experiments were carried out in which the performance of the tautology-checker on some small tautologies and non-tautologies was compared to the performance of the SFOL proof-checker on the corresponding generated proofs. The difference in the respective timings was remarkable. On an example on which the tautology-checker took 0.1 secs, it took 12 minutes to generate and check the formal proof. A 7000-fold difference! Such experimental results should be taken with a lot of salt since the proof-checker is not a very efficient one and a smaller fraction of the time spent in garbage collection gets included in the smaller execution time. On simpler examples, the ratios of the two times ranged from 1 : 100 to 1 : 600.

statements of a few of the definitions and lemmas involved. The proofs were done on a Symbolics 3600 LISP Machine.

Was the mechanical proof significantly more difficult than the informal proof? Both of these have been described in a fair amount of detail so that the reader can independently judge the level of difficulty involved. For the most part, the theorem prover was given the same information as one would glean from a careful reading of Shoenfield's *Mathematical Logic*. The use of linguistic devices such as typed meta-variables, ellipses, and the use of the phrase 'of the form', lent brevity to the informal proof. The version of the Boyer–Moore theorem prover used in this proof had no corresponding devices. In this case, the task of going from an informal proof to a mechanical proof is comparable in difficulty to the task of going from a carefully stated program specification to an executable program satisfying those specifications. Both tasks involve capturing certain notions using only the data-structures and constructs provided by the theorem proving system or programming language. While no mistakes were found in Shoenfield's outline of the proof, a few small gaps were found in the exposition. The clarity and detail in Shoenfield's outline were of immense help in the formulation of the mechanical proof outline.

The only other mechanical proof of a metamathematical theorem of comparable difficulty is Paulson's proof of the correctness of a Unification algorithm using Cambridge LCF [8]. Since a considerable part of that proof effort was spent making changes to the LCF system, a proper comparison is not possible.

Acknowledgements

Bob Boyer and J Moore have been a steady source of advice and encouragement. It was at their suggestion that I embarked on this project to prove theorems in meta-mathematics using their theorem prover. The fact that they were around to do any trouble-shooting and make suggestions had a great deal to do with the success of the proof. I am also indebted to Dr. Normal Martin for educating me in the ways of formal logic. Dr. Woody Bledsoe was responsible for my initiation into mechanical theorem proving and has made several constructive suggestions. I became aware of the work in metatheoretic extensibility through conversations with Bill Young. Gael Buckley read drafts of this paper and made numerous corrections and suggestions. Mike Gordon, Larry Paulson and Alan Bundy made several illuminating comments and suggestions. During the period when this work was carried out, I was supported by a University of Texas graduate fellowship for which I am grateful. I also wish to thank SERC UK for sponsoring my visits to the University of Cambridge and the University of Edinburgh.

References

1. Bourbaki, N., *Elements of Mathematics*, Volume: *Theory of Sets.*, Addison-Wesley, Reading, Mass. 1968.
2. Boyer, R. S. and Moore, J S., *A Computational Logic*, Academic Press, New York, 1979.

3. Boyer, R. S., and Moore, J S. Moore, 'Metafunctions: Proving them correct and using them efficiently as new proof procedures,' in *The Correctness Problem in Computer Science*, (eds. R. S. Boyer and J S. Moore) Academic Press, New York, 1981.
4. Brown, F. M., 'An investigation into the goals of research in automtic theorem proving as related to mathematical reasoning', *Artificial Intelligence* **14**, 221–242 (1980).
5. Davis, M. and Schwartz, J., 'Metamathematical extensibility for theorem verifiers and proof-checkers.' *Courant Computer Science Report* **12**, Courant Institute of Mathematical Sciences, New York, 1977.
6. Gordon, M. J., Milner, A. J. and Wadsworth, C. P., *Lecture Notes in Computer Science*. Volume 78: Edinburgh LCF. Springer-Verlag, Berlin, 1979.
7. Kleene, S. C., *Introduction to Metamathematics*. North-Holland, Amsterdam, 1952.
8. Paulson, L., 'Verification the Unification Algorithm in LCF.' *Technical Report* 50, University of Cambridge Computer Laboratory, 1984.
9. Post, E. L., *Annals of mathematics series*. Volume: 'The two-valued iterative systems of mathematical logic.' Princeton University Press, Princeton, New Jersey, 1941.
10. Shankar, N., 'Towards Mechanical Metamathematics.' *Technical Report* 43, University of Texas Institute for Computing Science, 1984.
11. Shoenfield, J. R., *Mathematical Logic*, Addison-Wesley, Reading, Mass., 1967.
12. Weyhrauch, R. W., 'Prolegomena to a theory of mechanized formal reasoning'. *Artificial Intelligence* **13**, 133–170 (1980).