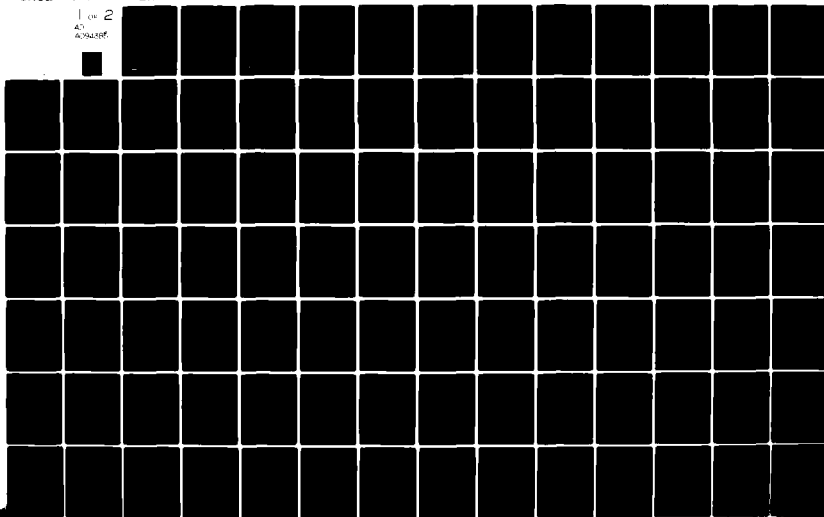


AD-A094 385

SRI INTERNATIONAL MENLO PARK CA COMPUTER SCIENCE LAB F/8 12/1
METAFUNCTIONS: PROVING THEM CORRECT AND USING THEM EFFICIENTLY --ETC(U)
DEC 79 R S BOYER J S MOORE N00014-75-C-0816
SRI/CSL-108 NL

UNCLASSIFIED

1 of 2
AD
A094385



ORG FILE COPY



81 2 02 188

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM	
1. REPORT NUMBER 14 SRI/CSL-108 ✓	2. GOVT ACCESSION NO. AD-A094 385	3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle) Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures.		5. TYPE OF REPORT & PERIOD COVERED 9) Technical Repts.	
7. AUTHOR(s) Robert S./Boyer 10) J. Strother/Moore		6. PERFORMING ORG. REPORT NUMBER	
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Laboratory SRI International Menlo Park, CA 94025		8. CONTRACT OR GRANT NUMBER(s) MCS-7904081 N00014-75-C-0816	
11. CONTROLLING OFFICE NAME AND ADDRESS Software Systems Science Office of Naval Research National Science Found. Department of the Navy Washington, D.C. 20550 Arlington, VA 22217		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS NR 049-378 12) 111	
14. MONITORING AGENCY NAME & ADDRESS (if diff. from Controlling Office) 15) NSA-74-75-2-1226 NSF-MZS 79-04081		12. REPORT DATE December 1979	
		13. NO. OF PAGES iv + 116	
		15. SECURITY-CLASS. (of this report) Unclassified	
16. DISTRIBUTION STATEMENT (of this report) Reproduction in whole or in part is permitted for any purpose of the United States Government. APPROVED FOR PUBLICATION DISTRIBUTION STATEMENT		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE	
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from report)			
18. SUPPLEMENTARY NOTES			
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) automatic theorem-proving metamathematics proof by induction symbolic manipulation proof checking			
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) We describe a sound method for permitting the user of a mechanical theorem- proving system to add executable code to the system, thereby implementing a new proof strategy and modifying the behavior of the system. The new code is mechanically derived from a function definition conceived by the user but proved correct by the system before the new code is added. We present a simple formal method for stating within the theory of the system the correctness of such functions. The method avoids the complexity of embedding the rules of inference of the logic in the logic. Instead, we define a meaning function that maps from			

DD FORM 1473
1 JAN 73
EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19. KEY WORDS (Continued)

20 ABSTRACT (Continued)

objects denoting expressions to the values of those expressions under a given assignment. We demonstrate that if the statement of correctness for a given "metafunction" is proved, then the code derived from that function's definition can be used as a new proof procedure. We explain how we have implemented the technique so that the actual application of a metafunction is as efficient as hand-coded procedures in the implementation language. We prove the correctness of our implementation. We discuss a useful metafunction that our system has proved correct and now uses routinely. We discuss the main obstacle to the introduction of metafunctions: proving them correct by machine.



CONTENTS

I	SUMMARY	1
	A. The Correctness Theorem	3
	B. The Implementation	6
	C. A Useful Metafunction	8
	D. Related Research	10
	E. The Key Problem: Theorem-Proving	13
	F. Organization of this Paper	15
II	AN EXAMPLE	17
	A. A Sketch of Our Formal Theory	17
	B. Abbreviations	20
	C. A Hypothetical Problem	22
	D. An Example of Alternative 1	23
	E. An Example of Alternative 2	24
	F. An Example of Alternative 3	26
	G. The Meta-Approach	27
III	FORMALITIES	41
	A. Alterations to A Computational Logic	41
	B. Histories and Theories	45
	C. Assumption of Consistency	46
	D. Explicit Value Terms	47
	E. Quotations	50
IV	THE METATHEOREM	53
	A. The Metaaxioms and Metadefinitions	53
	B. Statement and Proof of the Metatheorem	58
V	OUR IMPLEMENTATION OF METAFUNCTIONS	63
VI	INTERLISP REPRESENTATION OF TERMS	65
	A. The Role of Consistency	65

B.	Our Subset of INTERLISP	67
C.	Conventions for Mixing INTERLISP and the Theory . . .	68
D.	Basic INTERLISP Routines	69
E.	Global Variables	71
F.	The Definition of Terms	72
G.	Solidification	77
H.	Some Example INTERLISP Terms and Solidifications . . .	78
VII	PROOFS OF THE LEMMAS	81
A.	Lemmas 1 Through 7	81
B.	Lemmas 8 Through 18	86
C.	Lemma 19	94
VIII	EFFICIENT COMPUTATION ON EXPLICIT VALUES	97
IX	PROOF OF THE CORRECTNESS OF CANCEL	105
X	USING METAFUNCTIONS EFFICIENTLY	109
REFERENCES		113
INDEX		115

I SUMMARY*

How can the user of an automatic theorem-prover respond to the failure of the system to prove a given theorem? We know of three conventional responses: (1) modify the theorem-proving program itself, (2) guide the machine to the proof by interacting with a proof-checker-like facility, and (3) guide the machine to the proof by adding to its database of lemmas. Alternative (1) can be easy, but it may result in bugs in the theorem-prover and therefore requires extreme caution and expertise not to be expected of every user. Even if an error-free modification is made, it may amount to the assumption of what was supposed to be proved. Alternative (2) is safe and sound, but very tedious and it does not improve the theorem-prover for the next occasion on which a similar proof is required. Alternative (3) is also safe and sound if the theorem-prover proves the lemmas before accepting them. But this alternative, too, can be very tedious, or even hopeless, if the theorem-prover's heuristics fail to use the new lemmas in the ways intended by the user.

In this paper we describe and justify logically an implementation of a fourth alternative. We have improved the theorem-prover described in A Computational Logic [1] so that one of the alternatives now available to the user is to modify the theorem-prover by adding executable code. This code can cause the system to pursue new strategies and apply new proof techniques under arbitrary heuristic control. However, to ensure the soundness of the resulting theorem-prover, each purported proof technique must be proved correct by the theorem-prover before it is incorporated into the system.

* The work reported here was supported in part by NSF Grant MCS-7904081 and ONR Contract N00014-75-C-0816.

To extend our theorem-prover by adding a new piece of code the user proceeds as follows.

First, the user conceives some transformation from terms of the theory to terms of the theory that he wishes the theorem-prover would make.

Second, the user must understand a correspondence between terms of the theory and certain constants of the theory. This correspondence is simple and resembles the use of lists and atoms to represent the expressions of LISP.

Third, the user must define a new function in the logic of our system. While defining this function the user can think of himself as implementing the term transformation that he desires. He writes the function so that if his desired transformation takes a term t to a term t' , then his function maps the constant corresponding to t to the constant corresponding to t' . (Because our language is related to that of Pure LISP, it will often not be difficult for the user to define his function in our theory if he can define it in Pure LISP. In A Computational Logic [1], we present many examples of functions that perform simple list processing operations and we even present some functions that are actually simple theorem-provers.)

Fourth, the user presents the definition of his function to our theorem-prover. The theorem-prover attempts to check that the definition satisfies our principle of definition. If the theorem-prover is successful, then the definition is admitted and the user is assured that there does exist one and only one function satisfying his definition.

Fifth, if the definition is admitted, the user asks the system to prove a certain "correctness" theorem about the new function. The correctness theorem will be described informally in the next subsection.

Finally, if the correctness theorem is proved, the system incorporates into its simplifier new compiled code, derived from the function definition, that operates on the very INTERLISP data structures

used to represent terms in our theorem-proving program. If the new code is applied to an INTERLISP object that represents a term t , the result of the application will be an INTERLISP object that represents the term into which the user wished to transform t .

Once all these steps have been completed, the future behavior of the theorem-prover will be altered as follows. At a certain place in our theorem-prover's simplification routine, the code for the user's function is applied to the representation of the term that the theorem-prover is currently working on. The theorem-prover uses the output of that application as the representation of the new current term, thereby fulfilling the user's desire to transform terms.

In this paper we describe (a) the correspondence between terms and constants, (b) the correctness theorem for metafunctions, and (c) the translation from user definition to compiled code. We establish in this paper that the new compiled code is a correct simplifier. We illustrate all the ideas discussed with a metafunction that usefully extends our system and that has been proved correct. We also discuss the difficulty of proving useful metafunctions correct. Before presenting the details, we now sketch the entire paper and compare our work to that of others.

A. The Correctness Theorem

Suppose that the user has defined his function, fn , and that it has been accepted under our principle of definition. At this time, there are only a finite number of function symbols, say $f_1, \dots, f_m$, about which any axioms (e.g., definitions) have been made.

Before formulating the correctness theorem for fn , we first introduce and axiomatize in our current theory, T , two functions, FORMP and MEANING, which take one and two arguments, respectively. We assume, without loss of generality, that the symbols FORMP and MEANING are not among the $f_1, \dots, f_m$.

The precise axioms added to define these two functions are presented later. Intuitively, the axioms about FORMP are sufficient to

derive for any constant c whose only function symbols are in $f_1, \dots, f_m$ whether c corresponds to some term in the theory T . Intuitively, MEANING is axiomatized to take as its first argument a constant corresponding to some term in T and as its second argument an assignment of values to variables; MEANING returns the value of the term under the assignment.

For example, consider the term (PLUS X Y). The constant corresponding to this term is

(CONS "PLUS" (CONS "X" (CONS "Y" "NIL"))).

MEANING is axiomatized so that when it is applied to this constant and to the assignment that assigns 5 to "X" and 6 to "Y", then MEANING returns 11. (As will be explained, "X" is an abbreviation for a certain constant in our theory, namely (PACK (CONS 88 0)). 88 happens to be the ASCII code for the character X.)

The correctness theorem for the metafunction fn is:

(IMPLIES (FORMP X)
 (AND (EQUAL (MEANING X A)
 (MEANING (fn X) A))
 (FORMP (fn X)))).

That is, for all X , if X is a constant corresponding to some term, then for all assignments A , the MEANING of X under A is the same as the MEANING of $(fn X)$ under A . Furthermore, $(fn X)$ also corresponds to some term.

Suppose that the theorem-prover can prove the correctness theorem for fn . "So what," the reader may say, "if random axioms are assumed, then anything can be proved. How do I know that the axioms about FORMP and MEANING have any sense? Furthermore, even if they are sound, why should I be interested in a theorem that is a consequence of those axioms? In particular, how does your correctness theorem let me use fn as a proof procedure?" We now answer these questions by demonstrating how fn can be used to simplify terms.

First, let us delete the axioms about FORMP and MEANING that we added to the theory T after the definition of fn. Suppose that sometime later, perhaps even after adding some new function definitions (or even some other kinds of axioms), the user wishes some term p to be replaced by its transform. Let $f_1, \dots, f_p$ be the sequence of function symbols about which there are now axioms. Of course, all of the function symbols occurring in the term p will be among the $f_1, \dots, f_p$. Let us now define the functions MEANING and FORMP in such a way that the axioms that were previously added will be true. In our definition of MEANING, we shall adopt an entirely arbitrary position about the meaning of constants that contain function symbols other than $f_1, \dots, f_p$. We shall define FORMP so that it is false on constants not corresponding to terms of the current theory.

Because FORMP and MEANING are defined to satisfy the axioms we had previously added, there exists a proof of the correctness theorem for fn in our current theory.

Now suppose that c is the constant of the theory that corresponds to p. Since the definition of fn was accepted under our principle of definition, there exists a constant d such that

$$(EQUAL (fn c) d)$$

is a theorem. Since c corresponds to p, c satisfies FORMP. By the correctness theorem for fn, d will satisfy FORMP and will in fact be the constant corresponding to some term q. Furthermore, by the correctness theorem for fn, it will be a theorem that:

$$(EQUAL (MEANING c A) (MEANING d A)).$$

Finally, it can be shown that there will exist a trivial assignment a such that

$$(EQUAL (MEANING c a) p)$$

and

(EQUAL (MEANING d a) q))

are both theorems. To see that there always exists such a trivial assignment a, consider this example: let p be the term (PLUS X Y) and let c be corresponding constant

(CONS "PLUS" (CONS "X" (CONS "Y" "NIL")));

then for the assignment a

(CONS (CONS "X" X)
 (CONS (CONS "Y" Y)
 "NIL"))

it is the case that (EQUAL (MEANING c a) p).

Since the user's definition of fn transforms c into d, it is understood that the user wishes the theorem-prover to transform p into q. But we have proved that $p = (\text{MEANING } c \text{ } a) = (\text{MEANING } d \text{ } a) = q$. Hence, there is a proof that p is equal to q, and the theorem-prover is justified in replacing p with q.

B. The Implementation

In the preceding section we demonstrated how the proof of the correctness theorem for a function fn could be used to justify the transformation of some term p into another term q. It is not necessary to repeat the proof that such transformations are legal every time we make such a transformation. However, to take advantage of the metatheorem, we want our theorem-prover to obtain q from p efficiently. Specifically, we would like to obtain q from p with approximately the same speed that we could obtain q from p if we had hand-coded an INTERLISP function analogous to fn instead of introducing fn into our theory. There were three steps in computing q from p. The first step was finding the constant c corresponding to p. The second step was finding the constant d such that (EQUAL (fn c) d). And the final step was finding the term q to which d corresponded.

In our implementation of metafunctions we have arranged for the first and third steps to be exceedingly efficient: in fact, they literally take no time at all. The trick we use is to arrange our INTERLISP representation of terms so that if obj is an INTERLISP object representing a term t, then the INTERLISP list of length two whose first member is the atom QUOTE and whose second member is obj represents the constant corresponding to t.

Thus, if obj is an INTERLISP object we use to represent the term (PLUS X Y), then the INTERLISP object constructed by consing QUOTE onto obj onto NIL (in INTERLISP) is an object representing

(CONS "PLUS" (CONS "X" (CONS "Y" "NIL"))).

Thus, should we have a representation of p and desire to represent c, we embed the representation of p in a QUOTE. On the other side, should we have a representation of d and desire to obtain a representation of q, we take the cadr (i.e., car of the cdr) of the representation of d. It will turn out that we never actually have to represent c and d in going from p to q, but it is the term representation above that makes it possible. We will prove that if obj represents the term t then the result of embedding obj in a QUOTE represents a term whose MEANING under an appropriate assignment is t. The proof is complicated mainly by the limitations and restrictions imposed by efficiency considerations and INTERLISP (or any other real implementation language).

Now suppose we have the above representation of c. How can we obtain d quickly? Recall that d is the constant equal to (fn c). When a function is accepted under our definition principle our system compiles an INTERLISP routine whose body is analogous to the definition. For example, when the function APPEND is introduced with the definition:

```
Definition.  
(APPEND X Y)  
=  
(IF (LISTP X)  
    (CONS (CAR X) (APPEND (CDR X) Y))  
    Y).
```

the system generates and compiles the INTERLISP routine lappend:

```
(lappend (lambda (x y)
  (cond ((and (listp x) (neq (car x) 'sqm))
    (cons (car x) (lappend (cdr x) y)))
    (t y)))).
```

The relationship between the mathematical function APPEND and the INTERLISP routine lappend is then as follows. If obj_1 and obj_2 are INTERLISP objects that, when embedded in QUOTES, represent the constants c_1 and c_2 in the theory, then the INTERLISP object computed by (lappend obj_1 obj_2), when embedded in a QUOTE, represents a constant term d such that (EQUAL (APPEND c_1 c_2) d) is a theorem.

Thus, if fn has been accepted by the principle of definition the INTERLISP routine lfn has also been introduced. Suppose that after we have proved the correctness theorem for fn we desire to use fn to transform p to q . Suppose $objc$ represents p . Let $objc'$ be the result of embedding $objc$ in a QUOTE. Then $objc'$ represents c . Let $objd'$ be the result of embedding in a QUOTE the result of executing lfn on the cadr of $objc'$. Then $objd'$ represents d . Let $objd$ be the cadr of $objd'$. Then $objd$ represents q . By the metatheorem, p and q are provably equal, so we may substitute $objd$ for $objc$ in the representation of the conjecture being proved. But if x' is the result of embedding x in a QUOTE, the cadr of x' is x . Thus, the above scenario is equivalent to applying lfn to $objc$ (the representation of p) to obtain $objd$ (the representation of q).

C. A Useful Metafunction

We have used metafunctions to improve the power of the theorem-prover described in A Computational Logic. That theorem-prover was powerful enough to prove its way from the Peano-like axioms for the natural numbers and sequences to the existence and uniqueness of prime factorizations without any built-in arithmetic procedures or heuristics. However, it could not cancel an addend occurring arbitrarily deeply on both sides of an equation. The reason was that it was not possible to

state any useful lemma describing a schematic transformation. After implementing metatheoretic extensibility as described, we used it to add schematic cancellation.

The metafunction CANCEL was defined so that when given the symbolic expression representing the equation:

```
(EQUAL (PLUS B (PLUS C (PLUS I X)))  
        (PLUS (PLUS A (PLUS I J)) (PLUS K X)))
```

CANCEL produces the symbolic expression for

```
(EQUAL (PLUS B C)  
        (PLUS A (PLUS J K))).
```

CANCEL works by computing the fringe of the two PLUS-trees on each side of the symbolic equation, intersecting the fringes with the "bag" intersect function, subtracting the bag of common addends from each fringe, and then reconstituting the modified fringes into right-associated PLUS-trees in a new symbolic equation. However, to be correct CANCEL must take into account the typeless syntax of our theory. Thus, when given

```
(EQUAL A (PLUS A B))
```

it returns

```
(IF (NUMBERP A)  
    (EQUAL 0 (FIX B))  
    (FALSE)).
```

Furthermore, it does not bother to construct this expression if one side of the equation is not an element of the fringe of the other because it would be a heuristic mistake.

CANCEL is fairly complicated. In all, its definition (together with those of its subfunctions) requires 100 lines of "prettyprinted" text. In this paper we carefully describe the cancellation function and the proof of its correctness. The proof is constructed by our theorem-

prover from the axioms of Peano integers, atoms, and ordered pairs, without any built-in knowledge of arithmetic. We also explain the INTERLISP code generated for the function and explain how it is integrated into our automatic theorem-prover. The incorporation of this new proof procedure, which was mathematically defined and mechanically proved correct, increases the power of the system without noticeably affecting its speed.

D. Related Research

We now compare our approach to extensibility with recent work by others on the same subject. The basic premise of all work on extensible theorem-provers is that it should be possible to add new proof techniques to a system without endangering the soundness of the system. It seems possible to divide current work into two broad camps. In the first camp are those systems that allow the introduction of arbitrary new procedures, coded in the implementation language, but require that each application of such a procedure produce a formal proof of the correctness of the transformation performed. In the second camp are those systems that contain a formal notion of what it means for a proof technique to be sound and require a machine-checked proof of the soundness of each new proof technique. Once proved, the new proof technique can be used without further justification. Our system is in the second camp.

The LCF system, described by Milner et al. [5], is an example of a system in the first camp. The LCF metalanguage is a programming language that provides the data type "theorem." "Theorems" can be produced only by the basic rules of inference, which are implemented as procedures. The user can define new rules of inference as procedures that produce theorems by calling lower-level procedures under the control of arbitrary heuristics. The new rules of inference are sound (when they do not cause run-time errors) since the result produced by any given application must in fact have been produced by a correct sequence of applications of the lowest-level rules of inference.

Brown, in [3], proposes another system in the first camp. He suggests that each new proof procedure be coded in some conventional implementation language (e.g., LISP or machine code) but have an auxiliary procedure capable of producing a formal justification of any given application. To illustrate the idea, he exhibits a LISP program to find and cancel a single common addend on each side of an equation. As one example justification he suggests the proof procedure that derives the output from the input using only the associative, commutative, and cancellation laws for PLUS.*

In essence systems from the first camp are extensible because they provide a facility whereby the user can define succinct abbreviations that may be mechanically translated into long sequences of proof steps. The advantage such systems have over those of the second camp is that new proof procedures can be used without having to prove them correct. The primary disadvantage we see is one of efficiency: no matter how elaborate one's new rules of inference are, the system must plod through proofs at the lowest level.

Weyhrauch's work on FOL [7] exemplifies the second camp. He has implemented in FOL a system in which the formulas of one theory are the objects in another. In the upper theory he formalizes the syntax and rules of inference of the lower theory. To prove that a function in the upper theory is a sound simplifier for the lower theory one must prove in the upper theory that there exists a proof in the lower theory of the equality of the input and output. To apply such a metafunction to a formula during a proof at the lower level, Weyhrauch "reflects" the formula into a constant at the upper level, symbolically applies the metafunction to that constant, and then reflects the result back down. To make the process more efficient, Weyhrauch provides the perilous act

* As a second justification Brown uses a meaning function, virtually identical to ours and described earlier by Brown in [2], to express the schematic cancellation law. However, he does not express the law in a way that permits its mechanical application. In fact, he says that all of his mathematical justification procedures are sufficiently inefficient that they should be run to obtain a formal proof only when a step of the informal proof is "challenged."

of "semantic attachment" by which the user can associate programming entities (data structures and procedures) with logical entities (formulas and functions). Of course, perilous acts, while perfectly legitimate in the hands of a careful implementor, are to be considered illegal in the hands of careless users. Using semantic attachment, Weyhrauch arranges for the programming objects that represent formulas at one level to represent objects at the other. He can also arrange for certain built-in metafunctions (namely, those corresponding to the proof procedures in his system) to be executed very efficiently (as calls to the appropriate procedures).

Another example of the second camp was proposed by Davis and Schwartz in [4]. Like Weyhrauch they propose to embed formally the rules of inference of their logic in the logic. Unlike Weyhrauch they do not introduce a new "metatheory" but rather embed the rules of inference in a decidable subtheory. Like us, they then provide a MEANING-like function to map from formulas in the logic to constants. They propose to prove the correctness of "metafunctions" by proving that there exists a constant that is a "proof" of the equivalence of the input and output of the function.

Of course, while the second camp has only in the last few years begun to attract the attention of researchers in automatic theorem-proving, Gödel lit the campfire in 1931 when he showed that one can define functions that are proof-checkers for the theory containing them.

We can thus summarize the relationship between our work and that of others as follows. Our work is different from that of Milner et al. [5] and Brown [3] primarily because we are in the second camp. Our theoretical approach is different from Weyhrauch's [7] and Davis and Schwartz's [4] because we avoid the complexity of embedding the rules of inference of our logic in our logic and (unlike Weyhrauch) do not have to formalize the notion that one theory is the metatheory of

another.* Our implementation is different from any reported implementation of metatheoretic extensibility because we show how the user can achieve efficiency comparable to hand-coded procedures in the implementation language without availing himself of perilous acts.

E. The Key Problem: Theorem-Proving

Once the theoretical justification and practical implementation of metatheoretic extensibility is completed, the researcher must confront the fundamental problem for those in the second camp: proving the correctness of new metafunctions with a mechanical theorem-prover. If it is not practical to prove the correctness of new procedures with the tools provided, then -- depending on whether users can add new axioms -- the extensibility is either unusable or unsafe because users will add axioms stating the correctness of new procedures. The latter is little better than the ad hoc approach of alternative (1), i.e., the arbitrary hand-recoding of the theorem-prover.

We did not begin to consider seriously the incorporation of metafunctions until we had some evidence that our system could prove the correctness of metafunctions that would actually improve the system. The evidence came in September, 1978, when we had the system prove that CANCEL preserved MEANING (even though at that time the system could not employ that result metatheoretically).

The proof of the correctness theorem for the cancellation function did provide an interesting exercise for our theorem-prover. However, the number and difficulty of the intermediate lemmas that we formulated and the theorem-prover proved on the way to the main correctness theorem were less than the number and difficulty of the lemmas used in our proof of the correctness of a tautology-checker in Chapter IV of A Computational Logic. To formulate the lemmas and get the theorem-

* In defense of Weyhrauch's logical machinery it must be observed that his goal is the study of formal theories of reasoning -- in which metatheoretic reasoning plays a crucial role -- while ours is the much less ambitious one of getting permission to apply user-supplied proof procedures.

prover to prove the correctness theorem took one of the authors less than a day from the time the exercise was conceived. The earliest proof attempt found a bug in CANCEL: it cancelled multiple occurrences of an addend on one side against one occurrence on the other, due to the use of "list difference" rather than "bag difference." Because of the small amount of user effort necessary to introduce a correct cancellation procedure we are optimistic that our approach to extensibility may be feasible.

However, we conclude with three observations.

First, no implementation of metatheoretic extensibility will be feasible unless the mechanical theorem-prover can prove theorems about inductively defined concepts such as terms, formulas, and their meanings.

Second, it is interesting to ask whether a sound and practical approach to metatheoretic extensibility can be based on a simpler theorem-prover than ours. We suspect that it might take weeks to prove the correctness of a useful metafunction, such as our cancellation function, if one used a simple proof-checker.

Third, some theorem-prover researchers who, like us, are in the business of building theorem-provers to be used by a large community of users, may regard the provision for user extensibility (via either camp) to be an adequate response to the constant appeals from users to improve the power of the system. After all, extensibility gives the user the ability to tailor the system to his needs. But we do not see extensibility as a panacea for the current lack of theorem-proving power. It is a solution to a relatively simple problem: how to obtain insurance against unsoundness. The truly hard intellectual problem remains: the discovery of harmoniously cooperating heuristics for marshalling a very large number of facts and constructing difficult proofs.

F. Organization of this Paper

The structure of our presentation is as follows. In Section II we illustrate the introduction and use of a metafunction after briefly reviewing our formal logic as it was presented in [1]. In Section III we describe certain minor revisions made to the logic described in [1] to undertake the meta-approach conveniently, and we present some formal nomenclature used in the proof of the Metatheorem. In Section IV we state and prove the Metatheorem, which establishes that metafunctions can be applied. In Section V we outline our INTERLISP implementation of metafunctions in our theorem-proving program. In Section VI we describe the representation of terms in our theorem-proving program and in Section VII we prove some lemmas used in the demonstration that we have correctly implemented the Metatheorem. In Section VIII we explain how we translate user-defined functions into efficient INTERLISP procedures. In Section IX we describe the mechanical proof of the correctness of the example metafunction described in Section II and we comment on the difficulty of such proofs. In Section X we describe details of the implementation of metafunctions and give some output generated by our theorem-prover while using a metafunction.

II AN EXAMPLE

In this section we illustrate the use of metafunctions by writing in our logic a recursive function for cancelling all common addends on opposite sides of an equation. Our example is similar to but more elaborate than the one used by Brown in [3].

A. A Sketch of Our Formal Theory*

A term is either a variable symbol (which we define precisely in Section III), or else it is a sequence consisting of a function symbol of n arguments, followed by n terms. We use the prefix syntax of Church to write down terms. For example, if PLUS is a function symbol of two arguments, we write (PLUS X Y) where others might write PLUS(X,Y) or X+Y.

Our theory is obtained by starting with the axioms and rules of inference of propositional calculus with equality and function symbols (including the rule of inference that any instance of a theorem is a theorem) and adding (a) axioms for certain basic function symbols, (b) a rule of inference permitting proof by induction, (c) a principle of definition permitting the introduction of total recursive functions, and (d) the "shell principle," permitting the introduction of axioms specifying "new" types of inductively defined objects.

The basic function symbols are TRUE, FALSE, IF and EQUAL. The first two are function symbols of no arguments and may be thought of as distinct truth values. IF is a function symbol of three arguments and is axiomatized so that (IF X Y Z) = Z if X = (FALSE) and (IF X Y Z) = Y if X $\neq$ (FALSE). EQUAL is a function symbol of two arguments and axiomatized so that if X = Y, (EQUAL X Y) = (TRUE), and if X $\neq$ Y, (EQUAL

* Our formal theory is described in detail in Chapter III of A Computational Logic.

$X Y) = (\text{FALSE})$. (The "=" sign used here is the usual equality predicate.)

In our logic, terms also play a role similar to the one that formulas play in predicate calculus. For example, by an abuse of the word "theorem" (which is usually only applied to formulas), when we say $(\text{EQUAL } X X)$ is a theorem we mean $(\text{EQUAL } X X) \neq (\text{FALSE})$ is a theorem. Using IF we define the function NOT, of one argument, that returns (TRUE) if its argument is (FALSE) and returns (FALSE) otherwise. We similarly define the dyadic functions AND, OR, and IMPLIES.

Our principle of induction is based on the notion of well-founded relations (i.e., relations for which there exist no infinite sequence of successively smaller objects). Suppose r is a well-founded relation and that the measure m of $(d X)$ is r -smaller than m of X when X has property q . Then the induction principle permits one to prove $(p X Y)$ by proving two other conjectures. The first, called the "base case," is that $(p X Y)$ is true when X does not have property q . The second, called the "induction step," is that $(p X Y)$ is true when X has property q and $(p (d X) a)$ is true.

Our principle of definition provides the ability to introduce new recursive function definitions, provided certain theorems can be proved beforehand. The theorems require the exhibition of a measure and well-founded relation under which the arguments to recursive calls are getting smaller. Such theorems, together with some trivial syntactic requirements, are sufficient to guarantee the existence and uniqueness of a function satisfying the defining equation.

Finally, the "shell principle" provides a means for introducing "new" types of inductively defined objects that may be thought of as typed n -tuples with type restrictions on the components. The shell principle allows the user of the theorem-prover to characterize the desired objects by specifying n , the type restrictions, and (new) names for the primitive functions on the new type. Provided certain trivial syntactic requirements are met, the shell principle adds to the theory a set of axioms describing the new type. Using the shell principle we

introduce three sets of objects into the initial version of the theory.

These initial shells are:

- * The Peano integers, with recognizer NUMBERP, bottom object (ZERO), constructor ADD1 of one argument which must be a NUMBERP or else defaults to (ZERO), accessor SUB1, and well-founded relation SUB1P.
- * The "literal atoms," with recognizer LITATOM, bottom object (NIL), constructor function PACK of one argument of arbitrary type, accessor UNPACK (which returns (ZERO) on non-LITATOMs), and well-founded relation UNPACKP.
- * The "ordered pairs" or "lists," with recognizer LISTP, no bottom object, constructor function CONS of two arguments of arbitrary type, accessors CAR and CDR (which default to (NIL) on non-LISTPs) and well-founded relation CAR/CDRP.

The "recognizer" function is axiomatized to return (TRUE) or (FALSE) according to whether its argument is a member of the new type. The optional "bottom object" function of no arguments represents an "empty" object of the new type. The "constructor" function takes n arguments and has as its value an n -tuple of the new type. If the i th argument position has a "type restriction" that is not satisfied by the i th argument, the argument is "coerced" into the right type by being replaced by a "default value." The "type restriction" either requires that the argument be of one of a finite number of types or requires that the argument not be of one of a finite number of types. The i th "accessor" function is axiomatized so that when applied to an n -tuple of the new type it returns the i th component. When applied to an object other than a tuple of the new type, the i th accessor returns the i th default value. Finally, the "well-founded relation" is axiomatized so that the components of an n -tuple are smaller than the tuple.

We complete the initial development of the theory by introducing the well-founded relation and the measure function that are most commonly used in our theory: the "less than" relation on the Peano integers and the "size" of a shell object. The "less than" relation is introduced as the recursively defined function LESSP, which returns (TRUE) if its first argument is less than its second, and (FALSE) otherwise. LESSP treats any nonnumeric argument as though it were (ZERO).

The "size" of an object is computed by the function COUNT, which is defined to be (ZERO) on bottom objects and nonshells and one plus the sum of the sizes of the components on n-tuples.

The function PLUS is defined to compute the sum of its two arguments. Our theory does not provide a "typed" syntax. Thus terms such as (PLUS (TRUE) (TRUE)) are well formed. Our definition of PLUS "coerces" nonintegers to (ZERO). In particular, we define PLUS with the equation:

```
Definition.  
(PLUS X Y)  
=  
(IF (ZEROP X)  
    (FIX Y)  
    (ADD1 (PLUS (SUB1 X) Y))),
```

where (ZEROP X) is defined to be (TRUE) when X is (ZERO) or not a number, and (FALSE) when X is a non-(ZERO) number; (FIX Y) is defined to be Y if Y is a number and (ZERO) otherwise.

That completes our brief sketch of the theory.

B. Abbreviations

It is convenient to be able to write down certain terms succinctly. In [1] we introduce certain abbreviations, such as using (AND P Q R) as an abbreviation of (AND P (AND Q R)), using 3 as an abbreviation of (ADD1 (ADD1 (ADD1 (ZERO)))), and using (CADDR X) as an abbreviation of (CAR (CDR (CDR X))).

In this paper we modify one of our conventions and introduce two new ones.

We modify the convention in [1] under which expressions such as "X" and "PLUS" were abbreviations for certain LITATOM constants. We continue to use quotation marks to abbreviate LITATOMs, but we change the encoding. That is, we here adopt a new convention under which "X" is an abbreviation for a LITATOM, but for a different LITATOM than specified in [1]. Our new encoding (which makes it easier to implement

metafunctions efficiently) is as follows. Suppose wrd is a sequence of ASCII characters $c_1, \dots, c_n$ satisfying the definition of a "symbol" (see Section III). Suppose the ASCII character codes for $c_1, \dots, c_n$ are the integers $i_1, \dots, i_n$. Then "wrđ" is an abbreviation of

(PACK (CONS i_1 (CONS i_2 ... (CONS i_n 0) ...))).

Thus, "NIL" is an abbreviation of

(PACK (CONS 78 (CONS 73 (CONS 76 0)))).

and "QUOTE" is an abbreviation of

(PACK (CONS 81 (CONS 85 (CONS 79 (CONS 84 (CONS 69 0)))))).

One of the axioms for the PACK shell is that (EQUAL (PACK X) (PACK Y)) is true if and only if (EQUAL X Y) is true. Thus, "NIL" is not "QUOTE" because, using the similar axioms about the CONS and ADD1 shells, (CONS 78 ...) is not equal to (CONS 81 ...). In general, two abbreviated literal atoms are EQUAL if and only if the abbreviations are identical.

We introduce the following two new abbreviation conventions.

First, following LISP, we use (LIST $t_1 \dots t_n$) as an abbreviation of

(CONS t_1 (CONS t_2 ... (CONS t_n "NIL"))).

Thus, (LIST A B C) is an abbreviation of (CONS A (CONS B (CONS C "NIL"))).

Second, in Section III we will introduce a shell representing the negative integers and we shall there adopt a convention for abbreviating negative constants.

C. A Hypothetical Problem

We will now describe a realistic scenario in which the user of an automatic theorem-prover is confronted with the inadequacy of the system and is forced to consider the various alternative means of overcoming the problem.

Suppose we have a mechanical theorem-prover for the logic just described and that the theorem-prover can use equations as rewrite rules. Further suppose that we had instructed our theorem-prover to prove and then use the following equation as a rewrite rule:

$$\begin{aligned} &(\text{EQUAL } (\text{EQUAL } (\text{PLUS } X \ Y) \ (\text{PLUS } X \ Z)) \\ &\quad (\text{EQUAL } (\text{FIX } Y) \ (\text{FIX } Z))). \end{aligned}$$

This theorem is the cancellation law for addition. Roughly speaking, it says that if X is an addend on both sides of an equation it can be "cancelled." FIX is used because PLUS coerces its arguments to integers.

Applying this lemma as a rewrite rule from left to right allows the system to rewrite the equation:

$$\begin{aligned} &(\text{EQUAL } (\text{PLUS } I \ (\text{PLUS } X \ Y)) \\ &\quad (\text{PLUS } I \ (\text{PLUS } J \ K))) \end{aligned}$$

to the equation

$$\begin{aligned} &(\text{EQUAL } (\text{PLUS } X \ Y) \\ &\quad (\text{PLUS } J \ K)) \end{aligned}$$

(since (FIX (PLUS x y)) reduces to (PLUS x y) because PLUS is always numeric). So, apparently, our rewrite-driven system now "knows" how to cancel common addends.

But consider the following equation:

$$\begin{aligned} &(\text{EQUAL } (\text{PLUS } (\text{PLUS } A \ I) \ (\text{PLUS } B \ K)) \\ &\quad (\text{PLUS } J \ (\text{PLUS } K \ (\text{PLUS } I \ X)))). \end{aligned}$$

The cancellation law cannot be applied here, because the law requires

that the common addend be the first argument of the outermost PLUS-expression. Here we want to cancel the second and fourth addends on one side against the third and second on the other.

How might the user of our system respond to this failure of the system to carry out such a step in the proof? We consider the three alternatives sketched in the introduction and then the meta-approach.*

D. An Example of Alternative 1

Alternative (1) is to recode the theorem-prover. One suitable modification would be to build in an associative-commutative unification routine that "knows" PLUS as such a function and thus allows the cancellation law, in the form in which it was stated, to apply. A more direct solution would be to write a special-purpose routine for cancellation of PLUS. Roughly speaking, the code for such a modification would be as follows. If the expression in question is of the form (EQUAL t_1 t_2), regard t_1 and t_2 as trees of addends and compute their fringes. The intersection of the two fringes is the list of common addends. The result of cancelling all common addends is then obtained by removing each common addend from each fringe, reconstituting two PLUS expressions from the altered fringes and constructing the equation of those two expressions.

Such a program would correctly transform:

```
(EQUAL (PLUS (PLUS A I) (PLUS B K))
        (PLUS J (PLUS K (PLUS I X))))
```

to

```
(EQUAL (PLUS A B)
        (PLUS J X)).
```

* Lest the reader think that a mechanical theorem-prover without built-in cancellation is a straw man designed to show off the use of metafunctions, it should be observed that our theorem-prover, as described in [1], has no built-in arithmetic of any sort and yet can prove its way from the Peano axioms through the prime factorization theorem. Nevertheless, the addition of a cancellation mechanism improves the power and performance of the system.

However, one must be careful. For example if the intersect and delete operations do not respect duplications, one is liable to incorrectly simplify:

$$\text{(EQUAL (PLUS A (PLUS A (PLUS B C)))}$$
$$\text{(PLUS A (PLUS X Y)))}$$

to

$$\text{(EQUAL (PLUS B C)}$$
$$\text{(PLUS X Y)),}$$

cancelling two occurrences of A on the left against only one on the right. In addition, one must remember that PLUS coerces its arguments. For example, the simplification of:

$$\text{(EQUAL (PLUS A B) (PLUS A (PLUS C D)))}$$

to

$$\text{(EQUAL B (PLUS C D))}$$

is invalid, because the former might be true for a nonnumeric B while the latter would be false.

Thus, the implementor of the theorem-prover must consider these issues carefully before modifying the system. A less expert user of the system should not be allowed to make such a change.

E. An Example of Alternative 2

Alternative (2) is to carry out the cancellation by directing a proof-checker-like facility. This assumes the system has been well enough engineered to allow the user to intervene at this step in the proof without disabling all the desirable aspects of the automatic theorem-prover. But suppose we can so intervene. Recall the equation we wish to simplify

$$\text{(EQUAL (PLUS (PLUS A I) (PLUS B K))}$$
$$\text{(PLUS J (PLUS K (PLUS I X))))).$$

To describe the proof steps we must refer to individual PLUS-expressions in the formula. We number the PLUS-expressions consecutively from 1 to 6, in the left-to-right order in which they appear. Each time we change the equation, we renumber the PLUS terms with the same algorithm. Here is one of many possible sequences for simplifying the formula above, assuming we have proved that PLUS is associative and commutative:

```

Commute 2
Reassociate 1
Commute 5
Reassociate 5
Commute 4
Reassociate 4
Cancel
Commute 2
Commute 1
Reassociate 1
Commute 4
Reassociate 3
Cancel

```

The result is

```

(EQUAL (PLUS B A)
  (PLUS X J)).

```

This alternative does not solve the general problem of enabling the automatic theorem-prover to carry out arbitrary cancellations. Consequently, the user of the system must still be prepared to intervene when opportunities for cancellation arise in the future.

To solve the general problem with this technique we would have to write a program that detects the presence of common addends and generates a sequence of proof steps for cancelling them. This is just the approach of the first "camp" described in Section I. The program could use the fringe-intersection technique described above to identify the common addends. Then, for each common addend, t , the program could generate a sequence of commute and associate instructions intended to move t into the first argument of the outermost PLUS on each side, and then generate a cancel instruction. Finally, the entire sequence of instructions would be given to the proof-checker and actually carried

out. Of course, we do not have to worry about a mistake in our program rendering our theorem-prover unsound, but the process of generating the proof steps and then carrying them out is far more tedious than the ad hoc approach of the first alternative.

F. An Example of Alternative 3

Alternative (3) is to prove sufficient lemmas to let the theorem-prover carry out the necessary proof steps. In this case, it is sufficient to prove the ugly lemma:

$$\begin{aligned} &(\text{EQUAL } (\text{EQUAL } (\text{PLUS } (\text{PLUS } A \ I) \ (\text{PLUS } B \ K)) \\ &\quad (\text{PLUS } J \ (\text{PLUS } K \ (\text{PLUS } I \ X)))) \\ &(\text{EQUAL } (\text{PLUS } A \ B) \\ &\quad (\text{PLUS } J \ X))) \end{aligned}$$

by induction. This lemma is merely an ugly version of the PLUS-cancellation law.

Once again we see that the solution to the specific problem does not solve the general one of enabling the rewrite rules to carry out an arbitrary cancellation. For example, each of the equations below requires different versions of the cancellation law.

$$\begin{aligned} &(\text{EQUAL } (\text{PLUS } X \ Y) \\ &\quad (\text{PLUS } X \ Z)) \\ &(\text{EQUAL } (\text{PLUS } A1 \ (\text{PLUS } X \ Y)) \\ &\quad (\text{PLUS } B1 \ (\text{PLUS } X \ Z)) \\ &(\text{EQUAL } (\text{PLUS } A1 \ (\text{PLUS } A2 \ (\text{PLUS } X \ Y))) \\ &\quad (\text{PLUS } B1 \ (\text{PLUS } B2 \ (\text{PLUS } X \ Z)))) \\ &\dots, \end{aligned}$$

not to mention the "skewed" versions such as:

```
(EQUAL (PLUS A1 (PLUS A2 (PLUS X Y)))  
      (PLUS X Z))
```

```
(EQUAL (PLUS A1 (PLUS A2 (PLUS X Y)))  
      (PLUS B1 (PLUS X Z)))
```

....

It should be clear that no finite set of such rewrite rules will suffice to carry out all cancellations (unless we opted for alternative (1) and first built in some facts about the equivalence classes of PLUS expressions under associativity and commutativity).

G. The Meta-Approach

So much for the conventional alternatives. The meta-approach proposed in this paper is to encode the cancellation algorithm as a function in the logic itself and to prove it correct. We first describe how we represent symbolic expressions as objects in our theory, then we derive a definition of the cancellation function, and the statement of its correctness. Finally, we show how the statement of correctness, once proved, enables us to perform arbitrary cancellations from within the theory.

1. Symbolic Expressions

Since we want to write recursive functions on symbolic expressions we have to represent such expressions in terms of the objects of our theory, e.g., LITATOMs and LISTPs. Our symbolic expressions will be either variable symbols or the applications of function symbols to argument expressions. We represent function and variable symbols by LITATOMs. We represent the application of a function symbol to some arguments by the LISTP object whose CAR is the function symbol and whose CDR is a list of the appropriate number of argument expressions. Thus, the LITATOM (PACK (CONS 65 0)), abbreviated as "A", is a symbolic expression that can be thought of as representing a variable symbol. The LISTP object


```
(CONS "PLUS" (CONS "A" (CONS "I" "NIL")))
```

which may be abbreviated as

```
(LIST "PLUS" "A" "I")
```

is a symbolic expression representing the application of the function symbol "PLUS" to two variable expressions, "A" and "I".

Intuitively, the symbolic expression above corresponds to (PLUS A I), the application of the function PLUS to two arguments. Eventually we will formally assign meanings to symbolic expressions, making clear the connection between the LITATOM "PLUS" and the function PLUS. But at the moment, the reader is advised to ignore that aspect of the problem, forget that we are in a mathematical logic, and just pretend we are writing a program to manipulate such expressions according to the intuitive notion of their semantics.

2. The Cancellation Algorithm

We want a function, which we will call CANCEL, that when applied to a symbolic expression representing an equation such as

```
(EQUAL (PLUS (PLUS A I) (PLUS B K))  
        (PLUS J (PLUS K (PLUS I X))))),
```

yields the symbolic expression representing the cancelled equation,

```
(EQUAL (PLUS A B)  
        (PLUS J X)).
```

Here is how our function works. We first ask whether the expression is an equality with PLUS-expressions in both arguments. If so, we compute the fringe of the two PLUS-trees and intersect them (with a "bag intersection" function which respects duplications) to obtain a list of common addends. We subtract the common addends from each fringe (with "bag difference" which also respects duplications). Finally, we construct two new PLUS-trees from the two resulting bags of addends and embed them in an EQUAL expression.

We thus need functions for recognizing symbolic equations and PLUS-expressions, a function for computing the fringe of a tree of PLUS-expressions, the bag intersection and difference functions, and the function for constructing a tree of PLUS-expressions given the list of addends.

The function PLUS.TREE?, defined below, returns (TRUE) or (FALSE) according to whether its argument is a symbolic expression representing the application of the function symbol "PLUS":

```
Definition.
(PLUS.TREE? X)
=
(AND (LISTP X)
      (EQUAL (CAR X) "PLUS")).
```

If (PLUS.TREE? X) is (TRUE) we call X a "PLUS-tree." We could have defined PLUS.TREE? to check that (CDR X) is a list of two elements, but we will always be able to derive that if X is known to be well-formed. If X is a PLUS-tree then (CADR X) is the first argument expression and (CADDR X) is the second. The function EQUALITY? is similarly defined but recognizes symbolic equations.

We define the "fringe" of an expression with the function FRINGE. If its argument is a PLUS-tree, FRINGE recursively determines the fringe of the two arguments and concatenates them with the function APPEND. If its argument is not a PLUS-tree, FRINGE returns the singleton list containing that argument.

```
Definition.
(FRINGE X)
=
(IF (PLUS.TREE? X)
    (APPEND (FRINGE (CADR X))
            (FRINGE (CADDR X)))
    (CONS X "NIL")).
```

Before the equation above is admitted into the theory, the definitional principle requires the exhibition of a measure under which the argument is getting smaller according to some well-founded relation.

The measure COUNT and relation LESSP are sufficient -- in particular both (CADR X) and (CADDR X) have smaller COUNT than X when X is a LISTP even if X is not a well-formed PLUS-expression. Thus, the equation above is satisfied by one and only one function (as proved in [1]).

If X is

```
(LIST "PLUS"
      (LIST "PLUS" "A" "I")
      (LIST "PLUS" "B" "I"))
```

then (FRINGE X) is

```
(LIST "A" "I" "B" "K").
```

To operate on fringes we need the bag intersection and difference functions. Since the definitions are similar, we consider only the bag intersection function. The usual list intersection function asks of each element, e, in its first argument whether e is in the second. If so, e is put into the answer list, and if not, e is not put into the answer list. If e occurs m times in the first argument and at least once in the second, it is put into the answer m times. This will not do for our purposes, since it would lead us to believe we could cancel m occurrences of e. We must pay special attention to duplications. In particular, if e occurs in the first argument m times and in the second n times, then it must occur in the answer min(m,n) times. This can be arranged by deleting an occurrence of e from the second argument as soon as it has been used against an occurrence in the first argument. Here is the definition of the bag intersection function:

```
Definition.
(BAGINT X Y)
=
(IF (LISTP X)
  (IF (MEMBER (CAR X) Y)
    (CONS (CAR X)
          (BAGINT (CDR X)
                  (DELETE (CAR X) Y)))
    (BAGINT (CDR X) Y))
  "NIL").
```

For example,

```
(BAGINT (LIST "B" "C" "C" "D" "D")
        (LIST "A" "C" "C" "D" "E" "F"))
```

is equal to (LIST "C" "C" "D").* The bag difference function, BAGDIFF, is similarly defined.

Finally, we must define the function PLUS.TREE that converts a list of addends into a tree of PLUS-expressions. Recall that PLUS.TREE is used to "reconstitute" a PLUS-expression from its fringe minus any common addends. There are several special cases. If the new fringe is empty, it means all the elements of the old fringe were cancelled. Thus, PLUS.TREE should return the term representing 0. If the new fringe contains only one addend, x, then PLUS.TREE should return a symbolic term that "coerces" x to a number since that is what the original PLUS expression would have done. A suitable expression is (FIX x). Otherwise, PLUS.TREE builds a right-associated PLUS-tree from the list.

```
Definition.
(PLUS.TREE L)
-
(IF (NOT (LISTP L))
    (LIST "ZERO")
    (IF (NOT (LISTP (CDR L)))
        (LIST "FIX" (CAR L))
        (IF (NOT (LISTP (CDDR L)))
            (LIST "PLUS" (CAR L) (CADR L))
            (LIST "PLUS"
                  (CAR L)
                  (PLUS.TREE (CDR L))))))).
```

For example, when PLUS.TREE is given the list containing the symbolic

* The reader may be uncomfortable with the claim that BAGINT is the bag intersection function. How do we know we have thought of all the cases? The fact is that it does not matter. Since our functions are introduced under the principle of definition we are certain they are functions. Our bag intersection function might not be the same function the reader is thinking of, but it does exist and is uniquely defined. The proof of the correctness of CANCEL will establish that it has the necessary properties.

expressions for the variables A, B, and C, it returns the symbolic expression for

```
(PLUS A (PLUS B C)),
```

that is, (PLUS.TREE (LIST "A" "B" "C")) is

```
(LIST "PLUS"  
      "A"  
      (LIST "PLUS" "B" "C")).
```

We are now prepared to write a preliminary definition of CANCEL:

Definition.

```
(CANCEL X)
```

```
  (IF (AND (EQUALITY? X)  
           (PLUS.TREE? (CADR X))  
           (PLUS.TREE? (CADDR X)))  
      (LIST "EQUAL"  
            (PLUS.TREE  
              (BAGDIFF (FRINGE (CADR X))  
                       (BAGINT (FRINGE (CADR X))  
                               (FRINGE (CADDR X)))))  
            (PLUS.TREE  
              (BAGDIFF (FRINGE (CADDR X))  
                       (BAGINT (FRINGE (CADR X))  
                               (FRINGE (CADDR X)))))  
            X)).
```

But this definition of CANCEL does not handle the cancellation suggested by (EQUAL (PLUS A (PLUS B C)) A) because the second argument to the EQUAL is not a PLUS-tree. This situation will be handled specially. It is incorrect to follow the paradigm above and produce (EQUAL (PLUS B C) 0), because if A is nonnumeric, the former equation is (FALSE) while the latter might be (TRUE). A correct way to cancel (EQUAL (PLUS A (PLUS B C)) A) is to produce:

```
(IF (NUMBERP A)  
    (EQUAL (PLUS B C) 0)  
    (FALSE)).
```

We therefore add two more cases to the definition of CANCEL, one to handle the possibility that the second argument to the equation

is not a PLUS-tree but is a member of the fringe of the first, and the other to handle the symmetric case.

Here is the final definition of CANCEL.

Definition.

```
(CANCEL X)
=
(IF (AND (EQUALITY? X)
        (PLUS.TREE? (CADR X))
        (PLUS.TREE? (CADDR X)))
    (LIST "EQUAL"
        (PLUS.TREE
            (BAGDIFF (FRINGE (CADR X))
                     (BAGINT (FRINGE (CADR X))
                              (FRINGE (CADDR X)))))
        (PLUS.TREE
            (BAGDIFF (FRINGE (CADDR X))
                     (BAGINT (FRINGE (CADR X))
                              (FRINGE (CADDR X)))))
    (IF (AND (EQUALITY? X)
            (PLUS.TREE? (CADR X))
            (MEMBER (CADDR X) (FRINGE (CADR X))))
        (LIST "IF"
            (LIST "NUMBERP"
                (CADDR X))
            (LIST "EQUAL"
                (PLUS.TREE
                    (DELETE (CADDR X)
                           (FRINGE (CADR X))))
                (LIST "ZERO"))
            (LIST "FALSE"))
        (IF (AND (EQUALITY? X)
                (PLUS.TREE? (CADDR X))
                (MEMBER (CADR X)
                        (FRINGE (CADDR X))))
            (LIST "IF"
                (LIST "NUMBERP"
                    (CADR X))
                (LIST "EQUAL"
                    (LIST "ZERO")
                    (PLUS.TREE
                        (DELETE (CADR X)
                               (FRINGE (CADDR X)))))
                (LIST "FALSE"))
            X)))
```

The table below illustrates CANCEL's input-output behavior. If c is a symbolic expression corresponding to the equation in some row of the "input" column, then the equation corresponding to (CANCEL c) is given in the same row of the "output" column.

Input	Output
(EQUAL (PLUS (PLUS A I) (PLUS B K)) (PLUS J (PLUS I (PLUS K X))))	(EQUAL (PLUS A B) (PLUS J X))
(EQUAL (PLUS A X) (PLUS A (PLUS B X)))	(EQUAL (ZERO) (FIX B))
(EQUAL A (PLUS A B))	(IF (NUMBERP A) (EQUAL (ZERO) (FIX B)) (FALSE))

The reader may be discouraged by the complicated nature of the cancellation algorithm. However, the algorithm is no more complicated than the logic requires and raises the very issues we would have to face were we to build a general-purpose cancellation algorithm into the theorem-prover by any of the alternatives sketched. Furthermore, in stark contrast to alternative 1, we will here have our fears of lurking bugs eradicated by the system's proof of the correctness of the algorithm.

3. Correctness of CANCEL

What does it mean to say that CANCEL is correct? Intuitively, we would like to require that the output equation have the same truth value under all assignments as the input equation. To express this exactly, we introduce the notion of the "value" or "meaning" of an expression under an assignment to the variables in it.

For example, the meaning of (LIST "PLUS" "A" "I") under a given assignment is the sum of the meanings of "A" and "I" under the assignment. Suppose the only function symbols in which we were interested were FALSE, ZERO, FIX, NUMBERP, PLUS, TIMES, EQUAL, and IF.

Then we can define the function MEANING of two arguments, an expression and a list of pairs associating variables (in the CAR of each pair) with values (in the CDR). Given an atomic symbol, MEANING looks up and returns its value under the list of pairs using the function LOOKUP.*

Given an expression representing the application of one of the function symbols above, MEANING returns the result of applying the corresponding function to the recursively obtained MEANINGs of the arguments. Give any other object, MEANING returns the arbitrarily chosen value (TRUE).

```

Definition.
(MEANING X A)
=
(IF (NOT (LISTP X))
    (LOOKUP X A)

    (IF (EQUAL (CAR X) "FALSE")
        (FALSE)

        (IF (EQUAL (CAR X) "ZERO")
            (ZERO)

            (IF (EQUAL (CAR X) "FIX")
                (FIX (MEANING (CADR X) A))

                (IF (EQUAL (CAR X) "NUMBERP")
                    (NUMBERP (MEANING (CADR X) A))

                    (IF (EQUAL (CAR X) "PLUS")
                        (PLUS (MEANING (CADR X) A)
                             (MEANING (CADDR X) A))

                        (IF (EQUAL (CAR X) "TIMES")
                            (TIMES (MEANING (CADR X) A)
                                    (MEANING (CADDR X) A))

                            (IF (EQUAL (CAR X) "EQUAL")
                                (EQUAL (MEANING (CADR X) A)
                                        (MEANING (CADDR X) A))

                                (IF (EQUAL (CAR X) "IF")
                                    (IF (MEANING (CADR X) A)
                                        (MEANING (CADDR X) A)
                                        (MEANING (CADDR X) A))
                                    (TRUE))))))))))

```

 * LOOKUP is defined in Section III

There is nothing magic or "meta" about this function.* The equation defines a unique function and is accepted under the principle of definition because in each recursive call the COUNT (i.e., size) of the first argument gets smaller according to the well-founded relation LESSP. We happen to use the function PLUS to compute the value of a form beginning with the LITATOM "PLUS", but this association between the two is only an artifact of our definition of MEANING.

The intuitive statement that CANCEL is correct is that under any assignment the MEANINGS of the input and output of CANCEL are identical.

Theorem. CANCEL.PRESERVES.MEANING
 (EQUAL (MEANING X A)
 (MEANING (CANCEL X) A)).

This conjecture can be proved by our theorem-prover; the proof is discussed later in this paper. For the moment, suppose we have proved CANCEL.PRESERVES.MEANING.

4. Using CANCEL to Cancel

We now have a recursive function, CANCEL, that manipulates LISTP objects as though they were equations, and we can prove that the function is correct with respect to a particular definition of MEANING. But how can we use CANCEL to prove theorems?

Let us consider our example again. Suppose we are proving some conjecture and would like to cancel the common addends in the following equation, which we will call p:

(EQUAL (PLUS (PLUS A I) (PLUS B K))
 (PLUS J (PLUS I (PLUS K X)))).

* The prefix "meta" suggests something arcane, such as metaphysics. In fact, "meta" is Greek for "after." Metaphysics is so named not because it is subtly related to physics but because in the received order of Aristotle's works, the treatment of being, substance, cause, etc. comes after the treatises on physical matters.

That is, we would like to replace p by its cancelled form, which we will call q:

```
(EQUAL (PLUS A B)
      (PLUS J X)).
```

How can we use CANCEL, which operates on symbolic expressions, to derive the equation q from p and how to we know that q and p are provably equal?

Let c stand for the following term

```
(CONS "EQUAL"
      (CONS (CONS "PLUS"
                  (CONS (CONS "PLUS"
                              (CONS "A" (CONS "I" "NIL"))))
                        (CONS (CONS "PLUS"
                              (CONS "B" (CONS "K" "NIL"))))
                        "NIL"))))
      (CONS (CONS "PLUS"
                  (CONS "J"
                        (CONS (CONS "PLUS"
                              (CONS "I"
                                    (CONS (CONS "PLUS"
                                                  (CONS "K"
                                                        (CONS "X" "NIL"))))
                                                  "NIL"))))
                        "NIL"))))
      "NIL"))),
```

which may be abbreviated as

```
(LIST "EQUAL"
      (LIST "PLUS"
            (LIST "PLUS" "A" "I")
            (LIST "PLUS" "B" "K"))
      (LIST "PLUS"
            "J"
            (LIST "PLUS"
                  "I"
                  (LIST "PLUS" "K" "X")))).
```

Let alist stand for the term:

```
(LIST (CONS "A" A) (CONS "I" I) (CONS "B" B)
      (CONS "K" K) (CONS "J" J) (CONS "X" X)).
```

Then it is straightforward to confirm that the MEANING of c under alist is in fact p, the formula we wish to simplify. That is, the following is a theorem that may be proved by tediously expanding the definition of MEANING:

*1 (EQUAL p (MEANING c alist)).

But by CANCEL.PRESERVES.MEANING we have the theorem:

*2 (EQUAL (MEANING c alist)
 (MEANING (CANCEL c) alist)).

By expanding the definition of CANCEL we see that (CANCEL c) is equal to:

```
(LIST "EQUAL"
      (LIST "PLUS" "A" "B")
      (LIST "PLUS" "J" "X")),
```

which we will call d. Thus, we have the theorem:

*3 (EQUAL (MEANING (CANCEL c) alist)
 (MEANING d alist)).

But, by expanding the definition of (MEANING d alist) we have

*4 (EQUAL (MEANING d alist)
 (EQUAL (PLUS A B) (PLUS J X))),

or, equivalently

(EQUAL (MEANING d alist) q)

since the right-hand side of *4 is the equation we named q.

Thus, we can indeed use MEANING, CANCEL.PRESERVES.MEANING and CANCEL to derive q from p; furthermore, the chain of equalities *1 - *4 is a proof, within the theory, that p is equal to q, so we may replace p by q.

The paradigm for using CANCEL as a formula simplifier is to "lift" the formula to a symbolic expression with MEANING, "compute" CANCEL on that symbolic expression, and then use MEANING again to "drop" the symbolic expression back down to a formula. Of course, using the words "lift" and "drop" suggests that we are "ascending" to and "descending" from the metatheory, when in fact we are just translating the problem from one form to another.

It should be clear that we can use MEANING and CANCEL in this way to carry out an arbitrary cancellation, provided we can "lift" the formula into symbolic form and "drop" the output of CANCEL.

However, were we to implement the mechanical application of "metafunctions" along the lines just described, the implementation would sink into a swamp of PLUS-trees. Note for example that in lifting p we had to create a very large term, c. How do we even know such a term exists? Can we obtain it without a lot of work? Can we be sure that its MEANING is equal to p without the tedious expansion of MEANING required to justify *1? How can we quickly simplify (CANCEL c) to some new symbolic expression? Once that expression is obtained, do we know we can drop it back down to a formula -- that is, a formula not involving MEANING?

The remainder of this paper answers these and other questions. In particular, we carefully develop the logic behind the introduction and use of metafunctions, we describe an INTERLISP implementation that is very efficient, and we prove the correctness of our implementation.

III FORMALITIES

Here we lay some groundwork for the proof of the Metatheorem.

A. Alterations to A Computational Logic

The basic logic for which we prove the Metatheorem is the one described in Chapter III of A Computational Logic. To implement our approach to metafunctions, we found it desirable to make the following superficial changes to that logic.

1. Syntax

Here we alter the syntax of our language by increasing the set of characters that can be used in symbols.

A symbol is a nonempty sequence of characters $c_1, \dots, c_n$ such that (a) for each i greater than 0 and less than $n+1$, c_i is one of the following printing ASCII characters:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9
! # $ % + , - . / : ; < = > ? @ \ ^ _ `
```

and (b) c_1 is not a digit, the plus sign, the minus sign, or period.

We assume that associated with each symbol is a nonnegative integer called the arity of the symbol.

Intuitively, the arity of a symbol is the number of arguments the symbol takes when used as a function symbol. For example, the arities of TRUE, NOT, PLUS, and IF are, respectively 0, 1, 2, and 3. The arity of other symbols will become clear as time goes by.

A term is either a symbol or a finite sequence of length $n+1$ whose first member is a symbol of arity n , and whose remaining members are terms.

Although we now formally permit lower case letters in our terms, in this document we adhere to our convention of using lower case words to denote "metavariables" standing for terms. When we say that a term p has the form q , we mean that p can be obtained by replacing the lower case symbols in q by symbols or terms. For example, $(\text{LESSP } (D \ X) \ X)$ has the form $(r \ (D \ x) \ x)$ since it may be obtained by replacing r by LESSP and x by X . $(\text{CONS } A \ (\text{CONS } B \ \text{"NIL"}))$ has the form $(\text{CONS } x \ y)$ and also the form $(\text{LIST } a \ b)$, since $(\text{LIST } a \ b)$ is an abbreviation for $(\text{CONS } a \ (\text{CONS } b \ \text{"NIL"}))$. Finally, $(\text{CONS } A \ B)$ does not have the form $(\text{CONS } X \ y)$.

When we enclose a lower case symbol in quotation marks it should be understood to denote the same thing denoted by enclosing in quotation marks the denotation of the symbol. For example, if wrđ is understood to denote ABC , then " wrđ " is understood to denote " ABC ".

2. Literal Atoms

The shell definition for LITATOMs is modified: there is no bottom object.

We abandon the conventions in [1] specifying the interpretation of symbols in quotation marks (including the convention that " NIL " was an abbreviation of the (now absent) bottom object). We define our new abbreviation conventions below.

3. Ordered Pairs

The default values for the CONS shell are 0 and 0 (instead of " NIL " and " NIL ").

4. Negative Integers

Shell Definition

Add the shell MINUS of one argument
with recognizer NEGATIVEP,
accessor NEGATIVE.GUTS,
type restriction (NUMBERP X1),
default value 0, and
well-founded relation NEGATIVE.GUTSP.

5. Abbreviations

We continue to use the abbreviation conventions introduced in [1], except that we modify the conventions concerning the abbreviation of LITATOMs: if wrd is a symbol, and the ASCII character codes for the characters in wrd are, in order, $i_1, \dots, i_n$, then "wrđ" is an abbreviation of the term (PACK (CONS i_1 (CONS $i_2 \dots$ (CONS i_n 0)...))).

In addition, we add two new abbreviations.

If n is a positive integer and $t_1, t_2, \dots, t_n$ are terms, then (LIST $t_1 \dots t_n$) is an abbreviation of (CONS t_1 (LIST $t_2 \dots t_n$)). (LIST) is an abbreviation of "NIL".

If n is a positive integer, then $-n$ is an abbreviation of (MINUS n).

6. SYMBOLP

In preparation for defining FORMP, we add definitions for the functions LEGAL.CHAR.CODES, ILLEGAL.FIRST.CHAR.CODES, LEGAL.CHAR.CODE.SEQ, SYMBOLP, and LOOKUP to our basic theory.

(LEGAL.CHAR.CODES) has as its value the list of ASCII codes of those characters we permit to occur in symbol names (A-Z, a-z, 0-9, and a certain set of signs). (ILLEGAL.FIRST.CHAR.CODES) has as its value the list of those characters we do not permit as the first character of a symbol name (0-9, +, -, and .):

Definitions.

(LEGAL.CHAR.CODES)

=
(LIST 65 66 67 ... 88 89 90
97 98 99 ... 120 121 122
48 49 50 ... 55 56 57
33 35 36 38 43 44 45 46 47 58
59 60 61 62 63 64 92 94 95 126),

(ILLEGAL.FIRST.CHAR.CODES)

=
(LIST 48 49 50 ... 55 56 57 43 45 46),

(LEGAL.CHAR.CODE.SEQ L)

=
(AND (LISTP L)
(SUBSETP L (LEGAL.CHAR.CODES))
(NOT (MEMBER (CAR L) (ILLEGAL.FIRST.CHAR.CODES)))
(EQUAL (CDR (LAST L)) 0)),

(SYMBOLP X)

=
(AND (LITATOM X)
(LEGAL.CHAR.CODE.SEQ (UNPACK X))),

(LOOKUP X ALIST)

=
(IF (NLISTP ALIST)
"NIL"
(IF (AND (LISTP (CAR ALIST))
(EQUAL X (CAAR ALIST)))
(CDAR ALIST)
(LOOKUP X (CDR ALIST)))).

Functions used but not defined in this paper (e.g., NLISTP, LAST, MEMBER, and SUBSETP) are defined in [1] and are to be considered part of the basic theory. Informally, (NLISTP X) is (NOT (LISTP X)), (LAST X) is the last CONS in the CDR chain of X, (MEMBER X L) is (TRUE) or (FALSE) according to whether X is a member of the list L, and (SUBSET L1 L2) is (TRUE) or (FALSE) according to whether every member of L1 is a member of L2.

7. Miscellaneous

We prohibit the introduction of QUOTE and NIL as function symbols in axioms (including definitions and invocations of the shell principle).

B. Histories and Theories

The basic axioms are the axioms and definitions of Chapter III of A Computational Logic, as amended above.

t can be proved directly from a set of axioms A if and only if t may be derived from the axioms in A and the basic axioms by applying the following rules of inference:

- * The propositional calculus with equality and function symbols
- * The rule of inference that any instance of a theorem is a theorem
- * Our principle of induction

There are three kinds of axiomatic acts: (a) an application of the shell principle, (b) an application of the principle of definition, and (c) the arbitrary addition of an axiom. Each such act adds one or (in the case of the shell principle) more axioms.

A history is a finite sequence of axiomatic acts such that for each application of the principle of definition in the sequence, the theorems required by the principle of definition can be proved directly from the axioms added by the previous acts in the history.

If for some m , T_1 is the sequence of axiomatic acts $a_1, \dots, a_m$ and for some n , T_2 is the sequence of axiomatic acts $a_1, \dots, a_m, \dots, a_n$, where n is greater than or equal to m , then T_2 is an extension of T_1 .

If a history T_2 may be obtained from a history T_1 merely by adding definition acts to the end of T_1 , then T_2 is a definitional extension of T_1 .

A conjecture t can be proved in a history T if and only if t can be proved directly from the axioms of some definitional extension of T .

(For example, even though the proof of the correctness of the tautology-checker presented in [1] involves the introduction of the new, auxiliary concept of "IF-normal form," we say that the correctness theorem "can be proved" in the history before the addition of the definition of IF-normal.)

A history is constructive if it contains no arbitrary axioms.

A history is ordinary if there are no axioms of the history that mention APPLY, MEANING, MEANING.LST, ARITY, FORM.LSTP, or FORMP as function symbols.

t is a term of T if and only if t is a term, T is a history, and every symbol mentioned as a function symbol in t is used as a function symbol in some axiom of T .

C. Assumption of Consistency

We assume that if T is a constructive history, then T is consistent, i.e., (EQUAL (FALSE) (TRUE)) cannot be proved in T . This assumption plays an interesting role in the proof and implementation of the Metatheorem; we comment further upon that role in Section VI.

If any constructive history is inconsistent, then elementary number theory, at least, is inconsistent, since the constructive history can be embedded in elementary number theory.

While we might offer a "proof" that every constructive history is consistent, the only proof that we imagine requires at least the power of elementary number theory. We find it difficult to imagine a proof of the consistency of a constructive history within a mathematical theory that was less powerful than a constructive history because all logical theories of which we are aware require the power of inductive definition merely to define the language of any another logical theory.

D. Explicit Value Terms

In A Computational Logic we define the notion of an "explicit value term." We make extensive use of the properties of such terms in this paper, so we here summarize their properties.

Suppose T is a history.

A term in T is said to be an explicit value term with respect to T provided (i) it contains no variables, (ii) every function symbol in it is either TRUE, FALSE, or the bottom object or constructor function symbol of some shell class in T , and (iii) for each subterm of t of the form $(\text{const } t_1 \dots t_n)$, where const is the constructor function symbol of some shell class in T , each t_i satisfies the type restriction on the i th argument position of const .

Examples of explicit value terms are (ZERO), (ADD1 (ADD1 (ADD1 (ZERO)))), and (CONS (ADD1 (ZERO)) (ZERO)). (ADD1 (TRUE)) is not an explicit value because (TRUE) violates the numeric type restriction for ADD1.

Theorem. If t_1 and t_2 are two distinct explicit value terms with respect to T , then $(\text{NOT } (\text{EQUAL } t_1 t_2))$ is a theorem.

Proof. We induct on the structure of t_1 and t_2 .

Base Case. If either t_1 or t_2 is a variable, then the theorem is vacuously true because variables are not explicit values.

Induction step. If t_1 and t_2 are both function applications, say $(f s_1 \dots s_m)$ and $(g r_1 \dots r_n)$, then either f and g are the same function symbol or not. If f is not g , then the theorem follows, without appeal to the inductive hypotheses, merely by considering the shell axioms and the axiom that (TRUE) is not equal to (FALSE). If f and g are the same function symbol, then m is n and there is some i such that s_i is not the same term as r_i . By inductive hypothesis we can prove that $(\text{NOT } (\text{EQUAL } s_i r_i))$, and the desired conclusion follows from the shell axiom for f that $(\text{EQUAL } (f x_1 \dots x_n) (f x_1' \dots x_n'))$ is equivalent to the conjunction of $(\text{EQUAL } x_1 x_1')$... and $(\text{EQUAL } x_n x_n')$,

provided each x_i and x_i' satisfies the i th type restriction on const .
Q.E.D.

A function symbol fn is explicit value preserving with respect to T if it is TRUE , FALSE , IF , EQUAL , a function symbol axiomatized in T with an application of the shell principle, or a function symbol defined in T such that (a) every other function symbol used in the body of the definition is explicit value preserving with respect to T and (b) the theorems that must be proved under the principle of definition before fn is admitted can be proved directly from the shell axioms of T and the definitions of explicit value preserving functions with respect to T defined before fn .

For example, APPEND , as defined as follows:

```
Definition.  
(APPEND X Y)  
=  
(IF (LISTP X)  
    (CONS (CAR X) (APPEND (CDR X) Y))  
    Y)
```

is explicit value preserving.

The name "explicit value preserving" is derived from the observation that if some term t is the application of such a function to explicit values then it is possible to use the shell axioms and function definitions to derive an explicit value v such that $(\text{EQUAL } t \ v)$ is a theorem. For example, using shell axioms and the definition of APPEND it is easy to reduce $(\text{APPEND } (\text{CONS } 1 \ (\text{CONS } 2 \ \text{"NIL"})) \ (\text{CONS } 3 \ \text{"NIL"}))$ to the equivalent explicit value $(\text{CONS } 1 \ (\text{CONS } 2 \ (\text{CONS } 3 \ \text{"NIL"})))$. We now make this observation more formal.

A term t is reducible with respect to T if and only if t mentions no variable and every function symbol mentioned in t is explicit value preserving with respect to T .

We define recursively the reduction of a reducible term t of T . If t is an explicit value, then the reduction of t is t . If t is not an explicit value, then let s be the leftmost nonexplicit value subterm of

t , $(fn\ t_1 \dots t_n)$, such that either each t_i is an explicit value or fn is IF and t_1 is an explicit value. The reduction of t is the reduction of the term that results from replacing the leftmost occurrence of s in t with the term ans defined as follows:

- (1) If fn is EQUAL, ans is (TRUE) or (FALSE) according to whether t_1 is identical to t_2 .
- (2) If fn is IF, ans is t_3 or t_2 according to whether t_1 is (FALSE) or not.
- (3) If fn is a recognizer for a shell class with constructor function symbol c (and optionally, bottom object (btm)), then if the function symbol of t_1 is c (or btm) then ans is (TRUE), and otherwise ans is (FALSE).
- (4) If fn is the constructor function symbol for a shell class of T , ans is the result of replacing in s each argument t_i that does not satisfy the type restriction of the i th argument of fn with the i th default value.
- (5) If fn is the i th accessor for some shell class of T with constructor function symbol c , then if t_1 has the form $(c\ v_1 \dots v_n)$ for some $v_1, \dots, v_n$, then ans is v_i and otherwise ans is the i th default value for c .
- (6) If fn is a defined function in T , ans is the result of substituting each t_i for the corresponding formal parameter of the definition of fn in the body of fn .

That "the reduction of a reducible term" is well defined can be proved by induction because the definition of every defined function satisfies our principle of definition. That the reduction of a reducible term t is an explicit value that is provably equal to t follows from the fact that each step in the computation is justified by an axiom.

Theorem. If c is an explicit value with respect to T , then the reduction of (SYMBOLP c) in T is (TRUE) if and only if for some symbol w , c is " w ".

Proof. Suppose (SYMBOLP c) reduces to (TRUE). Then, by definition, the reduction of (AND (LITATOM c) (LEGAL.CHAR.CODE.SEQ (UNPACK c))) is (TRUE). Thus (LITATOM c) reduces to (TRUE) and c must have the form (PACK lst) and (UNPACK c) reduces to lst . Continuing the argument through LEGAL.CHAR.CODE.SEQ we finally conclude that c must

have the form (PACK (CONS t_1 (CONS t_2 ... (CONS t_n 0)...))), where $t_1, \dots, t_n$ are MEMBERS of (LEGAL.CHAR.CODES) and t_1 is not a MEMBER of (ILLEGAL.FIRST.CHAR.CODES). Thus, $t_1, \dots, t_n$ are NUMBERPs that may be abbreviated with the integers $i_1, \dots, i_n$. Furthermore, by the definitions of LEGAL.CHAR.CODES and ILLEGAL.FIRST.CHAR.CODES, $i_1, \dots, i_n$ are ASCII codes satisfying the restrictions we place on symbols. Let w be the symbol obtained by concatenating the ASCII characters for $i_1, \dots, i_n$. It is easy to confirm that c is " w ". The proof in the other direction is similar. Q.E.D.

E. Quotations

We now define the "correspondence" between terms.

Suppose that T is any history.

c is a quotation of t with respect to T if and only if c and t are terms and either (i) t is a symbol and c is " t " or (ii) t has the form (fn $a_1 \dots a_n$) and either (a) t is an explicit value with respect to T and c is (LIST "QUOTE" t) or (b) for some $q_1, \dots, q_n$ that are quotations, respectively, of $a_1, \dots, a_n$ with respect to T , c is (LIST "fn" $q_1 \dots q_n$). If c is a quotation of t with respect to T , then we say t is the dequotation of c with respect to T .

Thus, a quotation of the variable symbol X is (PACK (CONS 88 0)), which may be abbreviated " X ". A quotation of (ADD1 (ZERO)), which is the explicit value we abbreviate as 1, is (LIST "QUOTE" 1). But another quotation of 1 is (LIST "ADD1" (LIST "ZERO")). A quotation of (NOT X) is (LIST "NOT" " X "). Readers who feel uneasy about the use of such expressions as "QUOTE" and "ADD1" in terms should recall that they are mere abbreviations:

"QUOTE" (PACK (CONS 81 (CONS 85 (CONS 79 (CONS 84 (CONS 69 0))))))
 "ADD1" (PACK (CONS 65 (CONS 68 (CONS 68 (CONS 49 0)))))
 "ZERO" (PACK (CONS 90 (CONS 69 (CONS 82 (CONS 79 0)))))
 "NOT" (PACK (CONS 78 (CONS 79 (CONS 84 0))))
 "X" (PACK (CONS 88 0))

Theorem. If c is a quotation of t with respect to T , then c is an explicit value term.

Proof. We induct on t . If t is a variable, c is " t ", which is an explicit value. Otherwise, t has the form $(\text{fn } a_1 \dots a_n)$. If c has the form $(\text{LIST "QUOTE" } d)$ then, since QUOTE is not a function symbol, d is an explicit value and, thus, so is c . Otherwise, c has the form $(\text{LIST "fn" } q_1 \dots q_n)$, where each q_i is a quotation the corresponding a_i . By induction hypothesis, each q_i is an explicit value. Thus c is an explicit value. Q.E.D.

Theorem. If c is a quotation of both t_1 and t_2 with respect to T , then t_1 is the same term as t_2 .

Proof. We induct on the structure of c .

Base case 1. If the top function symbol of c is not CONS, then c is not the quotation of any term except a symbol. c cannot be a quotation of two distinct symbols.

Base case 2. If c has the form $(\text{LIST "QUOTE" } d)$, then since QUOTE is not a function symbol, c is a quotation only of the explicit value d .

Induction Step. If the top function symbol of c is CONS but c does not have the form $(\text{LIST "QUOTE" } d)$, then for some fn , $q_1, \dots, q_n$, c has the form $(\text{LIST "fn" } q_1 \dots q_n)$ where, by inductive hypothesis, each q_i is a quotation of some unique t_i . But then c can only be a quotation of the term $(\text{fn } t_1 \dots t_n)$. Q.E.D.

IV THE METATHEOREM

A. The Metaaxioms and Metadefinitions

When we are trying to prove the correctness of a metafunction, we have some axioms about MEANING, FORMP, and some auxiliary functions. These axioms, called the "metaaxioms," specify the values of MEANING and FORMP on symbolic expressions corresponding to terms in the current theory. The axioms do not specify the values of MEANING and FORMP on objects that "look like" symbolic expressions but that have unrecognized function symbols.

Once a metafunction has been proved correct, we may apply it as a new proof procedure -- even if new function symbols have been added to the theory. Formally speaking, its application involves the introduction of definitions of MEANING, FORMP, and some auxiliary functions. These definitions are called the "metadefinitions." The definitions not only specify the values of MEANING and FORMP on symbolic expressions corresponding to terms in the new theory, but on all explicit values. For example, FORMP is (FALSE) on any object that "looks like" a symbolic expression but has an unrecognized function symbol. But because the metaaxioms are easy consequences of the metadefinitions, we can prove the correctness of the metafunction -- and use it -- in the new theory.

By not completely specifying MEANING and FORMP during the correctness proof for a metafunction, we permit the application of the metafunction in extensions containing new function symbols. By introducing MEANING and FORMP under the principle of definition when the metafunction is used in the proof of a new conjecture, the proof of the conjecture does not depend upon nondefinitional axioms about MEANING and FORMP.

Assume we have a standard ordering of all symbols and that TRUE, NOT, IF, and PLUS come first, in that order.

Suppose that (a) T is a history, (b) TRUE, NOT, IF, PLUS, f_5 , ..., f_m is the sequence of symbols mentioned as function symbols in axioms of T in the standard order, and (c) 0, 1, 3, 2, a_5 , ..., a_m is the sequence of the arities of the symbols TRUE, NOT, IF, PLUS, f_5 , ..., f_m .

1. The Metaaxioms

The metaaxioms for T are as follows:

(MEANING.LST X A)

=

(IF (NLISTP X)

"NIL"

(CONS (MEANING (CAR X) A)

(MEANING.LST (CDR X) A))),

(IMPLIES (NLISTP X)

(EQUAL (MEANING X A) (LOOKUP X A))),

(EQUAL (MEANING (LIST "QUOTE" X) A)

X),

(IMPLIES (NOT (EQUAL FN "QUOTE"))

(EQUAL (MEANING (CONS FN X) A)

(APPLY FN (MEANING.LST X A)))),

(EQUAL (APPLY "TRUE" X)

(TRUE)),

(EQUAL (APPLY "NOT" X)

(NOT (CAR X))),

(EQUAL (APPLY "IF" X)

(IF (CAR X) (CADR X) (CADDR X))),

(EQUAL (APPLY "PLUS" X)

(PLUS (CAR X) (CADR X))),

... and so on for all of the functions f_5 , ..., f_m ,

(EQUAL (ARITY "TRUE") 0),

```

(EQUAL (ARITY "NOT") 1),

(EQUAL (ARITY "IF") 3),

(EQUAL (ARITY "PLUS") 2),

... and so on for all of the symbols "f5", ... , "fm",

(FORM.LSTP X)
=
(IF (NLISTP X)
    (EQUAL X "NIL")
    (AND (FORMP (CAR X))
         (FORM.LSTP (CDR X))))),

(FORMP X)
=
(IF (NLISTP X)
    (SYMBOLP X)
    (IF (EQUAL (CAR X) "QUOTE")
        (AND (LISTP (CDR X))
              (EQUAL (CDDR X) "NIL"))
        (AND (EQUAL (ARITY (CAR X)) (LENGTH (CDR X)))
              (FORM.LSTP (CDR X)))))).

```

Note that the value of (ARITY X) is unspecified if X is not one of the LITATOMS "TRUE", "NOT", "IF", "PLUS", "f₅", ..., "f_m". Further, (FORMP X) is unspecified if X is a LISTP and (ARITY (CAR X)) is unspecified.

Note also that FORMP is more elaborate than we sketched it in the discussion of CANCEL. In particular, we require that function and variable symbols be SYMBOLPs and that objects used as function symbols have numeric arity and be provided with the proper number of arguments; in addition, we allow the symbolic expression whose CAR is the LITATOM "QUOTE" and whose MEANING is defined to be the CADR of the expression. Note that we are not elevating QUOTE to a "function symbol" even at the meta level of FORMP. We are merely axiomatizing the recursive functions FORMP and MEANING to behave in a certain way when they encounter a LISTP object whose CAR is (PACK (CONS 81 (CONS 85 (CONS 79 (CONS 84 (CONS 69 0)))))).

2. The Metadeclarations

The metadeclarations for T are as follows:

```
(APPLY X L)
=
(IF (EQUAL X "TRUE")
    (TRUE)
    (IF (EQUAL X "NOT")
        (NOT (CAR L))
        (IF (EQUAL X "IF")
            (IF (CAR L) (CADR L) (CADDR L))
            (IF (EQUAL X "PLUS")
                (PLUS (CAR L) (CADR L))
                (IF (EQUAL X "f5")
                    (f5 (CAR L) ... (CAD...R L))
                    ...

(IF (EQUAL X "fm")
    (fm (CAR L) ... (CAD...R L))
    (TRUE))))))

(MEANING.LST X A)
=
(IF (NLISTP X)
    "NIL"
    (CONS (IF (NLISTP (CAR X))
              (LOOKUP (CAR X) A)
              (IF (EQUAL (CAAR X) "QUOTE")
                  (CADR (CAR X))
                  (APPLY (CAAR X)
                        (MEANING.LST (CDAR X) A))))
          (MEANING.LST (CDR X) A))),

(MEANING X A)
=
(CAR (MEANING.LST (LIST X) A)),
```

```

(ARITY X)
=
(IF (EQUAL X "TRUE")
  0
  (IF (EQUAL X "NOT")
    1
    (IF (EQUAL X "IF")
      3
      (IF (EQUAL X "PLUS")
        2
        (IF (EQUAL X "f5")
          a5
          ...
          (IF (EQUAL X "fm")
            am
            "NIL"))))))),

```

```

(FORM.LSTP X)
=
(IF (NLISTP X)
  (EQUAL X "NIL")
  (AND (IF (NLISTP (CAR X))
    (SYMBOLP (CAR X))
    (IF (EQUAL (CAAR X) "QUOTE")
      (AND (LISTP (CDAR X))
        (EQUAL (CDDR (CAR X)) "NIL"))
      (AND (EQUAL (ARITY (CAAR X))
        (LENGTH (CDAR X)))
        (FORM.LSTP (CDAR X))))))
  (FORM.LSTP (CDR X))),

```

```

(FORMP X)
=
(FORM.LSTP (LIST X)).

```

If T is an ordinary history, let $\underline{MA}[T]$ be the history that results from adding the metaaxioms for T to T as arbitrary axioms and let $\underline{MD}[T]$ be the history that results from adding the metadefinitions for T to T , in order, as (explicit value preserving) definitions.

B. Statement and Proof of the Metatheorem

For the remainder of this section, let us make the following suppositions.

- (1) T_1 is a constructive, ordinary history,
- (2) simp is an explicit value preserving function defined in T_1 with arity 1,
- (3) in $MA[T_1]$ we can prove the formula

*META
(IMPLIES (FORMP X)
 (AND (EQUAL (MEANING X ALIST)
 (MEANING (simp X) ALIST))
 (FORMP (simp X))))), and

- (4) T_2 is an ordinary extension of T_1 .

It is our objective to show that if p is a term of T_2 , c is a quotation of p with respect to T_2 , and d is the reduction of $(\text{simp } c)$, then d is a quotation of some term q of T_2 with respect to T_2 and $(\text{EQUAL } p \ q)$ is a theorem of T_2 . Thus, while proving theorems in T_2 , we may at anytime replace a term p of T_2 with the dequotation of the reduction of the application of simp to a quotation of p . First we note a few lemmas.

Let $T_{1.5}$ be the extension of T_1 that results from adding (a) the applications of the shell principle made while extending T_1 to T_2 and (b) the metadefinitions for T_2 as definitions.

We now make a few trivial observations about $T_{1.5}$:

- (1) In $T_{1.5}$, APPLY may mention functions that are undefined. Nevertheless, $T_{1.5}$ is constructive since T_1 is constructive and in producing the extension we added no arbitrary axioms.
- (2) In $T_{1.5}$, ARITY, FORM.LSTP, and FORMP are explicit value preserving.
- (3) The explicit values of T_2 are just the explicit values of $T_{1.5}$.
- (4) If t is reducible with respect to $T_{1.5}$, then t is reducible with respect to $MD[T_2]$ and the reduction of t in $T_{1.5}$ is the reduction of t in $MD[T_2]$.

- (5) Finally, c is a quotation of t with respect to T_2 if and only if c is a quotation of t with respect to $T_{1.5}$.

We are interested in $T_{1.5}$ because, being constructive, it is consistent and yet has the property proved below. Our interest in consistency is explained in Section VI, after we have proved the Metatheorem and discussed its use.

Theorem A. If c is an explicit value with respect to $T_{1.5}$, then the reduction of $(FORMP\ c)$ in $T_{1.5}$ is $(TRUE)$ if and only if for some term t of T_2 , c is a quotation of t with respect to $T_{1.5}$.

The proof is by induction on the structure of c .

Base case 1. If the top function symbol of c is not $CONS$, then the reduction of $(FORMP\ c)$ is the reduction of $(SYMBOLP\ c)$. But the reduction of $(SYMBOLP\ c)$ is $(TRUE)$ if and only if for some symbol w , c is " w ". But " w " is a quotation of w .

Base case 2. If c is a term of the form $(LIST\ "QUOTE"\ d)$, then the reduction of $(FORMP\ c)$ is $(TRUE)$, d is an explicit value, and c is a quotation of d .

Induction step. Suppose the function symbol of c is $CONS$ but c does not have the form $(LIST\ "QUOTE"\ d)$. Suppose that the reduction of $(FORMP\ c)$ is $(TRUE)$. Then for some symbol fn and for some explicit values $c_1, \dots, c_n$, c has the form $(LIST\ "fn"\ c_1 \dots c_n)$, the reduction of $(ARITY\ "fn")$ is n , and the reduction of each $(FORMP\ c_i)$ is $(TRUE)$. By inductive hypothesis, there exist terms $t_1, \dots, t_n$ of T_2 such that c_i is a quotation of t_i with respect to $T_{1.5}$. By the construction of the definition of $ARITY$, the arity of fn is n and c is a quotation of the term $(fn\ t_1 \dots t_n)$. On the other hand, suppose that for some term t , c is a quotation of t . t must have the form $(fn\ t_1 \dots t_n)$ and c must have the form $(LIST\ "fn"\ a_1 \dots a_n)$ for some quotations $a_1, \dots, a_n$ of $t_1, \dots, t_n$. Hence the reduction of $(FORMP\ c)$ is $(TRUE)$. Q.E.D.

If $v_1, \dots, v_n$ is a sequence of symbols, then the standard alist for $v_1, \dots, v_n$ is the term:

$(\text{LIST } (\text{CONS } "v_1" v_1) \dots (\text{CONS } "v_n" v_n)).$

Theorem B. If c is a quotation of t with respect to T_2 , t is a term of T_2 , and a is the standard alist for any sequence of variables that contains all of the symbols that are used as variables in t , then the following can be proved in $\text{MD}[T_2]$:

$(\text{EQUAL } (\text{MEANING } c \ a) \ t).$

Proof. We prove this theorem by induction on the structure of the term t .

Base Case. If t is a symbol, then c is $"t"$ and $(\text{MEANING } "t" \ a)$ is $(\text{LOOKUP } "t" \ a)$ which is $(\text{CDR } (\text{CONS } "t" \ t))$ which is t .

Induction step. Suppose t has the form $(\text{fn } t_1 \dots t_n)$. If t is an explicit value and c is $(\text{LIST } \text{"QUOTE"} \ t)$, then by the definition of MEANING , $(\text{MEANING } c \ a)$ is t . If c does not have the form $(\text{LIST } \text{"QUOTE"} \ d)$, then c has the form $(\text{LIST } \text{"fn"} \ q_1 \dots q_n)$, where each q_i is a quotation of t_i . Since every variable of any t_i is a variable of t , we have, by inductive hypothesis that for each i , $(\text{EQUAL } (\text{MEANING } q_i \ a) \ t_i)$. Because fn is a function symbol used in an axiom of T_2 and is not QUOTE , we have by the definition of MEANING that

$(\text{EQUAL } (\text{MEANING } (\text{LIST } \text{"fn"} \ q_1 \dots q_n) \ a) \ (\text{fn } (\text{MEANING } q_1 \ a) \dots (\text{MEANING } q_n \ a))).$

Thus we derive

$(\text{EQUAL } (\text{MEANING } (\text{LIST } \text{"fn"} \ q_1 \dots q_n) \ a) \ (\text{fn } t_1 \dots t_n)).$

Q.E.D.

The Metatheorem.

Suppose that

- (1) p is a term of T_2 ,
- (2) c is a quotation of p with respect to T_2 , and
- (3) d is the reduction of $(\text{simp } c)$ in T_2 .

Then, the reduction of $(\text{FORMP } d)$ in T_2 is (TRUE) , d is the quotation of some term q of T_2 , and in T_2 we can prove

$(\text{EQUAL } p \ q)$.

Proof. Since c is a quotation with respect to T_2 of a term of T_2 , $(\text{FORMP } c)$ reduces to (TRUE) in $T_{1.5}$ by Theorem A. Because we can prove $*\text{META}$ in $\text{MA}[T_1]$ and because the metaaxioms of T_1 are each theorems of $T_{1.5}$, we can prove $*\text{META}$ in $T_{1.5}$. Detaching the hypothesis of $*\text{META}$, we can prove in $T_{1.5}$ that $(\text{FORMP } d)$. The reduction of $(\text{FORMP } d)$ in $T_{1.5}$ is either (TRUE) or (FALSE) . If it is (FALSE) , then $T_{1.5}$ is inconsistent. But we have assumed that $T_{1.5}$ is consistent since it is constructive. Thus the reduction of $(\text{FORMP } d)$ in $T_{1.5}$ is (TRUE) and its reduction in T_2 is (TRUE) also.

By Theorem A there exists a term q of T_2 such that d is a quotation of q . Let q be the dequotation of d . Let a be the standard alist for a sequence containing all of the variables in p and q . Since every axiom (including the definitions) of $\text{MA}[T_1]$ can be proved in $\text{MD}[T_2]$, both $*\text{META}$ and $(\text{FORMP } c)$ can be proved in $\text{MD}[T_2]$. Detaching the hypothesis of $*\text{META}$ in $\text{MD}[T_2]$, we derive that $(\text{EQUAL } (\text{MEANING } c \ a) \ (\text{MEANING } d \ a))$. But since $(\text{EQUAL } p \ (\text{MEANING } c \ a))$ and $(\text{EQUAL } q \ (\text{MEANING } d \ a))$ by Theorem B we obtain $(\text{EQUAL } p \ q)$ in $\text{MD}[T_2]$. Since $\text{MD}[T_2]$ is a definitional extension of T_2 , we can prove $(\text{EQUAL } p \ q)$ in T_2 . Q.E.D.

V OUR IMPLEMENTATION OF METAFUNCTIONS

In the next three sections of this paper we describe our efficient implementation of metafunctions in INTERLISP [6]. Here are the steps in our description and the proof of the implementation's correctness:

- (1) We describe in Section VI how in our theorem-proving program we represent the terms of our theories with INTERLISP objects.
- (2) Let $(\text{list } \text{'QUOTE } \text{obj})$ denote an INTERLISP list of length two with the INTERLISP atom QUOTE as its first element and the INTERLISP object obj as its second. In Section VII, Lemma 18, we demonstrate, under the suppositions and hypothesis of the Metatheorem, that if $(\text{list } \text{'QUOTE } \text{obj})$ represents some explicit value w of T_2 , $(\text{FORMP } w)$ reduces to (TRUE) in $\text{MD}[T_2]$ (or, equivalently, in $T_{1.5}$), and the INTERLISP machine state corresponds to the history T_2 , then obj represents a term of T_2 .
- (3) We then demonstrate in Section VII, Lemma 19, that if the INTERLISP machine state corresponds to any history T and in that state obj is an INTERLISP object that represents a term p of T , then $(\text{list } \text{'QUOTE } \text{obj})$ represents a term in T that is a quotation of p with respect to T .
- (4) Finally, in Section VIII we describe how we have arranged so that if the INTERLISP machine state corresponds to any theory T and fn is an explicit value preserving function with respect to T , then stored in the definition cell of the INTERLISP literal atom lfn is a routine such that if $c_1, \dots, c_n$ are explicit values of T represented by the INTERLISP objects $(\text{list } \text{'QUOTE } \text{obj}_1), \dots, (\text{list } \text{'QUOTE } \text{obj}_n)$, and val is the INTERLISP object computed by applying lfn to $\text{obj}_1, \dots, \text{obj}_n$, then $(\text{list } \text{'QUOTE } \text{val})$ represents the reduction of $(\text{fn } c_1 \dots c_n)$.

We are then free to utilize the Metatheorem in the following way. Suppose that during a proof in T_2 we have in hand an INTERLISP object objc representing some term p of T_2 . By Lemma 19, $(\text{list } \text{'QUOTE } \text{objc})$ represents some term c that is a quotation of p . Let objd be the result of applying lsimp to objc . By our implementation of lfunctions , $(\text{list}$

'QUOTE objd) represents the term d that is the reduction of (simp c). From the Metatheorem, we know that (FORMP d) reduces to (TRUE). From Lemma 18, we learn, then, that objd represents some term q of T_2 . From Lemma 19, again, we learn that d is a quotation of q. Finally, from the Metatheorem, we learn that (EQUAL p q) is a theorem of T_2 . Consequently, we may engage in the typical theorem-prover activities justified by "substitution of equals for equals," replacing objc with objd.

The place in our theorem-prover where metafunctions are thus utilized is described in Section X.

VI INTERLISP REPRESENTATION OF TERMS

In this section, we explain how we represent terms in our theorem-prover.

A. The Role of Consistency

Before describing our representation, let us first anticipate some problems we face and explain why we are interested in consistency.

Recall how we use the metafunction `simp` to simplify a term represented by the INTERLISP object `objc`: the theorem-prover executes the routine `lsimp` on `objc`, obtains some INTERLISP object `objd` as a result, and uses `objd` in place of `objc`.

We find the fact that `objd` represents a term to be remarkable in light of all the invariants a data object must satisfy to represent a term in an efficiently implemented theorem-prover. To appreciate the subtlety of the situation, consider what might happen when the compiled INTERLISP code for the theorem-prover begins to operate on `objd`. If `objd` is an INTERLISP list cell, our theorem-proving code will assume that the car of `objd`, `x`, is an INTERLISP literal atom representing a function symbol and may fetch `x`'s property list (the left half-word of the location addressed by `x`) where information about the function is stored. But what would happen if `x` were not a literal atom -- for example, what would happen if it were an INTERLISP number? Then the machine instruction used to obtain the property list might return an illegal object whose use could lead the theorem-prover to random, unpredictable behavior.

For efficiency, we do not check that `objd` actually satisfies all the properties the theorem-prover requires of an object representing a term; so what ensures us that it does? The answer is that we know that

the INTERLISP object obtained by embedding `objd` in a `QUOTE` represents a term, `d`, and that `(FORMP d)` reduces to `(TRUE)`. We will prove, in Lemma 18, that `objd` must therefore represent a term.

How do we know that `(FORMP d)` reduces to `(TRUE)`? One's first reaction is: `d` is `(simp c)`, `(FORMP c)` is `(TRUE)`, and `*META` establishes `(IMPLIES (FORMP c) (FORMP (simp c)))`. But wait. That argument only implies that `(FORMP d)` is provably `(TRUE)`. But our Lemma 18 requires that it reduce to `(TRUE)`. However, as we argued in the proof of the Metatheorem, `(FORMP d)` must reduce to `(TRUE)` or `(FALSE)` and were it to reduce to `(FALSE)` a constructive theory (namely $T_{1.5}$) would be inconsistent.

Let us consider the role of consistency from another point of view. Recall that we require that `*META` be proved in a constructive (consistent) theory, T_1 . Suppose we weakened that and permitted it to be proved in any theory. What would happen if the theory were inconsistent? One consequence of an inconsistency in T_1 is that one admits a proof procedure that might prove falsehoods. But nothing is wrong with that state of affairs, for if T_1 is inconsistent, one may indeed prove anything in it. However, something worse happens. Suppose the inconsistency permits `(FORMP (simp c))` to be proved when in fact it reduces to `(FALSE)`. Then `objd` will not in fact be an object satisfying the theorem-prover's restrictions on terms. Consequently, the application of `simp` may cause totally unpredictable behavior by the theorem-prover (e.g., the smashing of disk files, illegal memory fetches, loss of the day's work, and so on).

Such catastrophic behavior is a far cry from the expectation that an inconsistent T_1 leads to well-behaved proofs of falsehoods. Some readers may feel that the user of an inconsistent theory deserves even catastrophic failures. This is an ill-considered position. Mechanical theorem-provers often deal with inconsistent theories because a standard proof strategy is to assume the negation of what one desires to prove and then seek to prove `(FALSE)`. The theory T_2 in which one may apply `simp` may be such a theory and cause no catastrophic effects. The moral

however is that one should not prove the soundness of one's new proof procedures while in an inconsistent theory.

Finally, we should observe that we could have stated *META so that (FORMP (simp c)) did not have to be proved in T_1 and then could have implemented a run-time check that objd indeed represents a term. We then could have permitted T_1 to be inconsistent without catastrophic consequences. We did not adopt this approach because in most cases the proof of the FORMP part of *META is straightforward (see Section IX) and buys efficiency at the mere expense of complicating this paper.

B. Our Subset of INTERLISP

Our objective in this section is to describe how we represent terms in our theorem-prover in a way that permits the efficient implementation of the Metatheorem without sacrificing efficiency in more routine activities. We describe our representation by exhibiting two INTERLISP programs. The first determines whether its argument represents a term. The second returns a conventional representation of the term represented. We chose to describe our representation with such programs because INTERLISP provides a very succinct way to describe complicated INTERLISP data structures.

The INTERLISP definitions we present in this paper and our proofs about those definitions are made in a vastly simplified version of INTERLISP akin to Pure Lisp. We do not specify the subset precisely. However, the subset does have the following properties.

- * We make no use of "destructive" operations such as SETQ, SET, and RPLACA.
- * We restrict our attention to INTERLISP structures that are not "circular."
- * In establishing the correctness of our mapping between terms in the theory and INTERLISP objects, we assume we have an INTERLISP machine with unlimited resources.

The latter assumption permits us to ignore such problems as running out of list space or exhausting the machine's stack while proving, for example, that we can represent every explicit value. Of course, we did

not make this assumption while designing the representation, since the economical representation of terms is one of our objectives, and our theorem-prover actually causes errors and aborts the proof attempt when resources are exhausted. But at the moment we are engaged in the mathematical exercise of establishing the correctness of a mapping between terms in our theory and INTERLISP objects and we are using INTERLISP as a mathematical language to describe those objects.

We assume the reader is familiar with the standard, primitive LISP routines such as cond, cons, car, cdr, and listp. (MACLISP users: read "consp" for "listp".)

C. Conventions for Mixing INTERLISP and the Theory

The syntax of INTERLISP expressions is very similar to that of our theory. Because we will often be referring to functions in our theory and to INTERLISP functions (henceforth called "routines") in close proximity, we adopt the following three conventions to demark clearly the boundary between the two.

First, despite the fact that most INTERLISP routines are spelled in upper case, we spell them in lower case here. We will use upper case words to denote functions in our theory. Thus (LENGTH X) is a term in our theory, while (length x) refers to the value of the INTERLISP routine length applied to the value of the variable x.

Second, we shall adopt the syntactic convention of writing 'w for (QUOTE w) when w is an INTERLISP literal atom. Thus, the INTERLISP form that might be written as:

```
(COND ((EQ I 0) (LIST (QUOTE ZERO)))  
      (T (LIST (QUOTE ADD1) (FN (SUB1 I)))))
```

will here be displayed as

```
(cond ((eq i 0) (list 'ZERO))  
      (t (list 'ADD1 (fn (sub1 i)))).
```

Third, it is often necessary in this paper to refer to characters obtained by printing certain INTERLISP objects. To indicate the result of printing the value of an INTERLISP form, we surround the form with vertical bars. Such an expression is to be understood as denoting the sequence of characters obtained by printing the value of the enclosed INTERLISP form (with `prin4`, using the original read table and decimal radix). Thus, if we say "We can prove (EQUAL |(cons 'ZERO NIL)|(REMAINDER X X))", then we mean "We can prove (EQUAL (ZERO) (REMAINDER X X))". Of course, to use the vertical bar notation in a context where a term is expected, we will have to establish that the result of printing the value of the form denotes a term.

D. Basic INTERLISP Routines

Our representation of terms will involve the following defined auxiliary INTERLISP routines:

- * The routine `legal.char.codes` takes no arguments and returns a list, in ascending numerical order, of the integers mentioned in the definition of `LEGAL.CHAR.CODES` in Section III, which are the ASCII codes for the characters that we permit in symbols.
- * The routine `illegal.first.char.codes` takes no arguments and returns a list, in ascending numerical order, of the integers mentioned in the definition of `ILLEGAL.FIRST.CHAR.CODES` in Section III, which are the ASCII codes for the characters that may appear in symbols, but not first.
- * The routine `legal.char.code.seq` returns T or NIL according to whether its argument `x` has or does not have all of the following properties: (i) (`listp x`), (ii) for every `c`, if (`member c x`), then (`member c (legal.char.codes)`), (iii) it is not the case that (`car x`) is a member of (`illegal.first.char.codes`), and (iv) the `cdr` of the last list cell in `x` is 0.
- * The routine `unpack0`, when given a literal atom `x`, returns a list of the ASCII codes of the characters in the "print name" of `x`, in the order in which the characters occur in the print name, and terminating in a 0 instead of a NIL. (The "print name" of a literal atom is the sequence of characters produced when the atom is printed. Thus, (`unpack0 'ABC`) is a list that prints as (65 66 67 . 0).)

- * The routine pack0, when given an object x satisfying (legal.char.code.seq x), returns the unique INTERLISP literal atom atm such that (unpack0 atm) is x.
- * The routine symbolp returns T or NIL according to whether its argument represents a symbol in our logic. The definition of symbolp is:

```
(symbolp (lambda (x)
            (and (litatom x)
                 (legal.char.code.seq
                  (unpack0 x)))))).
```

Note that if (symbolp x) holds, then x is a literal atom and its print name is a legal.char.code.seq. If (symbolp x) holds, (pack0 (unpack0 x)) is x. Furthermore, if (legal.char.code.seq seq) holds, then (unpack0 (pack0 seq)) is equal to seq and (symbolp (pack0 seq)) holds. These are the basic properties required of symbolp, pack0, unpack0, and legal.char.code.seq. INTERLISP contains literal atoms for which pack0 and unpack0 are inverses but which we do not use as symbols. For example, there is one whose print name is 1A2. We could have defined legal.char.code.seq to check for precisely the syntax of those objects for which pack0 and unpack0 are inverses, but that would have made its definition far more complicated, for while 1A2 is such an object, 1E2 is not (it is $1.0 \times 10^2 = 100.0$).

As the user of our theorem-prover adds definitions, shells, and other kinds of axioms, our theorem-prover naturally changes the state of the INTERLISP machine.

- * The routine arity, of one argument x, is defined so that if x is a symbol which is used as a function symbol in some axiom of the history represented by the current state of INTERLISP, then arity returns an INTERLISP integer representing the number of arguments that x takes. Otherwise, (arity x) is NIL.
- * The routine shell.state, of no arguments, returns an alist which encapsulates information about the uses of the shell principle in the construction of the history represented by the current state of INTERLISP. Each member of the list has a shell constructor function symbol or a bottom object function symbol as its car. The cdr is a list whose length is the number of arguments of the function symbol. Each element of the cdr encodes the type restrictions placed on

the corresponding argument to the constructor function. Recall that each type restriction for a shell can be expressed as a requirement that the corresponding argument either be recognized by one of a finite collection of shell recognizers or else be recognized by none of a finite number of shell recognizers. Thus, it would be sufficient if each element of the cdr were either of the form (ONE.OF . r) or (NONE.OF . r), where r was a list of recognizers. However, for convenience we define r to be the list of all constructor and bottom object function symbols recognized by the recognizers in question.

- * The routine addl.nest takes a nonnegative integer x as its argument and it returns an object that prints as (ZERO) for 0, (ADD1 (ZERO)) for 1, (ADD1 (ADD1 (ZERO))) for 2, and so on. Its definition is

```
(addl.nest
  (lambda (i)
    (cond ((equal i 0) (list 'ZERO))
          (T (list 'ADD1 (addl.nest (sub1 i)))))).
```

- * The routine bminus, if given an argument representing an integer x, returns a INTERLISP representation of the negative of x.
- * The routine baddl, if given an argument representing an integer x, returns a INTERLISP representation of x+1.
- * The routine plistp returns T or NIL according to whether or not its argument is a (possibly empty) list whose final cdr is NIL. Its definition is

```
(plistp (lambda (x)
  (cond ((nlistp x) (eq x NIL))
        (T (plistp (cdr x)))))).
```

E. Global Variables

For our representation of terms, we have assigned distinct values to three INTERLISP global variables lt, lf, and lsqm. Each value is an INTERLISP literal atom, and none of the values represents a symbol in the logic (i.e., (symbolp lt), (symbolp lf) and (symbolp lsqm) are all NIL). The role of these variables is explained below. By choosing names that begin with digits we are guaranteed that these INTERLISP

variables never have the same names as variables in our logic. This plays a minor role in the efficient compilation of explicit value preserving functions.

F. The Definition of Terms

Roughly speaking we shall represent variables as symbols and function applications as lists in which the car is the function symbol and the cdr is the list of the appropriate number of argument terms. However, we wish to encode explicit value terms efficiently. For example, we prefer to represent the explicit value term

```
(CONS (PACK (CONS 78 (CONS 79 (CONS 84 0))))  
      (CONS (PACK (CONS 80 0))  
            (PACK (CONS 78 (CONS 73 (CONS 76 0)))))),
```

which may be abbreviated by:

```
(LIST "NOT" "P")
```

with the INTERLISP list constant that prints as (QUOTE (NOT P)). There are two reasons: we consume much less space, and if constants in the theory are represented efficiently by INTERLISP constants then we can choose to represent terms in our program by INTERLISP constants which simultaneously represent constants in our theory and facilitate the efficient application of metafunctions to formulas.

For example, we can represent some NUMBERPs and NEGATIVEPs by INTERLISP integers, some LITATOMs by INTERLISP literal atoms, and some LISTPs by INTERLISP lists. Of course, we cannot use the INTERLISP literal atom 'P to represent both the variable P and the explicit value (PACK (CONS 80 0)). So we use 'P to represent the variable P and the value of (list 'QUOTE 'P) to represent (PACK (CONS 80 0)).

Similarly, if the value of (list 'QUOTE obj₁) represents some explicit value term t₁ and the value of (list 'QUOTE obj₂) represents some explicit value term t₂, then the value of (list 'QUOTE (cons obj₁ obj₂)) represents the explicit value term (CONS t₁ t₂). To obtain the

CAR of (CONS t_1 t_2) from its representation, we apply car to (cons obj₁ obj₂). To obtain the UNPACK of (PACK (CONS 80 0)) from its representation, we apply unpack0 to to 'P. However, we must address three problems.

The first problem concerns the precise choice of our representation of LITATOMs. The reason LITATOMs must be represented efficiently is that they are used by FORMP to stand for function and variable symbols. Thus, the internal representation of a LITATOM satisfying SYMBOLP must be an INTERLISP object the theorem-prover can use as a function or variable symbol. But to implement a theorem-prover efficiently one's function and variable symbols should be distinguishable by eq (in one machine instruction) and have property lists. The obvious candidates are literal atoms. So certain LITATOMs are represented by INTERLISP literal atoms. But for theoretical simplicity we allow a LITATOM to be constructed from any object (e.g., (PACK 1200) is a LITATOM in the theory), while INTERLISP requires that literal atoms be constructed only from lists of ASCII codes so that they are "printable." To represent the theory's "unprintable" LITATOMs we will use the structures described below for user-defined shells. Thus, there are two distinct ways LITATOMs are represented, but any given LITATOM will be represented in only one of the ways, depending on whether it is a SYMBOLP.

The second problem is that while certain shell constants in the theory (e.g., some NUMBERPs, LITATOMs, and LISTPs) have obvious INTERLISP representatives, others (e.g., (TRUE), (FALSE), and user-defined shells such as stacks or triples) do not. We could use the INTERLISP "user data type" facility to declare a new INTERLISP type for each of these unusual types in the theory. But this is unacceptable because (a) every user data type is initially allocated 512 words of storage, regardless of how many items of that type are required, (b) having additional data types in use slows down garbage collections, (c) the efficiently compiled and widely used INTERLISP routine equal does not work on user data types, and (d) INTERLISP user data types do not print out or read in conveniently.

We shall therefore encode user-defined shell constants as INTERLISP list structures containing the name of the constructor (or bottom object) and the n-tuple of objects representing the explicit value arguments. But such a list structure could be confused with the representation of a LISTP containing n+1 objects. To avoid ambiguity, we cons the value of lsqm (which stands for "shell quote mark") onto the front of the structure. This marking scheme avoids ambiguity because lsqm is not the internal representation of any explicit value -- in particular it does not satisfy symbolp and so does not represent a LITATOM -- so a list with lsqm as its car could not possibly represent a LISTP whose CAR was represented by lsqm. For example, if TRIPLE is a user-defined shell constructor, then the explicit value (TRIPLE 1 (PACK (CONS 80 0)) 2) is represented by the value of (list lsqm 'TRIPLE 1 'P 2), embedded in a QUOTE form.

We could represent (TRUE) and (FALSE) similarly -- for example, (TRUE) could be represented by the value of (list lsqm 'TRUE), embedded in a QUOTE form -- but that would be very inefficient because (TRUE) and (FALSE) are constantly tested against in tight loops in the theorem-prover. Instead, we represent (TRUE) and (FALSE) with (the values of) the variables lt and lf, embedded in QUOTE forms. These values cannot be mistaken as representing LITATOMs in the theory even though they are literal atoms in INTERLISP.

The third problem is the finite limitations imposed by INTERLISP (and all programming languages). For example, no INTERLISP literal atom can have more than 125 characters, nor can any integer require more than 36 bits to represent it. In this paper we pretend INTERLISP imposed no such limits. To ensure the correctness of our program, we have designed it to cause errors (which result in the abortion of any proof attempt) when the finite limitations of INTERLISP are reached. Thus, for example, we use our own badd1 routine for adding one to an integer -- and causing an error if the result is unrepresentable -- rather than use the built-in routine add1 which returns an inaccurate answer on overflow.

We now make the foregoing sketch precise. An INTERLISP object `obj` is called an INTERLISP term if `(term obj)` is non-NIL. Below we define `term` and its subroutine `evg` (for "explicit value guts") which recognizes the INTERLISP objects that may be embedded in QUOTES to represent explicit value terms.

```
(term
  (lambda (x)
    (cond
      ((nlistp x)
       (symbolp x))
      ((eq (car x)
           'QUOTE)
       (and (listp (cdr x))
            (null (cddr x))
            (evg (cadr x))))
      (T (and (plistp (cdr x))
              (equal (length (cdr x))
                    (arity (car x)))
              (for z in (cdr x) always (term z)))))).
```

We define `(evg y)` so that if `y` is an INTERLISP object that, when embedded in a QUOTE, represents some explicit value term `v`, then `(evg y)` is the top-level function symbol of `v`. Otherwise, `(evg y)` is NIL.

```

(evg
  (lambda (y)
    (cond ((nlistp y)
      (cond ((fixp y)
        (cond ((lessp y 0) (quote MINUS))
              ((equal y 0) (quote ZERO))
              (T (quote ADD1))))
        ((eq y lt)
          (quote TRUE))
        ((eq y lf)
          (quote FALSE))
        ((symbolp y)
          (quote PACK))
        (T NIL)))
      ((eq (car y) lsqm)
        (cond
          ((and (listp (cdr y))
            (plistp (cdr y))
            (equal (length (cddr y))
              (arity (cadr y)))
            (assoc (cadr y)
              (shell.state))
            (for z in (cddr y) always (evg z))
            (for restriction in (cdr (assoc (cadr y)
              (shell.state)))
              as arg in (cddr y) always
                (cond ((eq (car restriction) 'ONE.OF)
                  (member (evg arg)
                    (cdr restriction)))
                  (T (not (member (evg arg)
                    (cdr restriction)))))))
            (cond
              ((eq (cadr y) (quote PACK))
                (not (legal.char.code.seq (caddr y))))
              ((eq (cadr y) (quote MINUS))
                (equal (caddr y) 0))
              (T (not (member (cadr y)
                (quote (ADD1 ZERO CONS)))))))
          (cadr y))
          (T NIL)))
      ((and (evg (car y))
        (evg (cdr y)))
        (quote CONS))
      (T NIL))))

```

The puzzled reader should be reminded that `term` and `evg` are only used to say precisely how we represent terms. The theorem-prover only

calls `term` once when a term is submitted to it by the user. Internal subroutines know what terms look like -- indeed, it is to make these internal subroutines efficient that `term` is so complicated. As for the correctness of metafunctions, all we have to prove is that when given FORMPs they return FORMPs. The careful reader will note that FORMP is considerably simpler than `term` -- in particular there is nothing corresponding to the ghastly `evg`. The fact that a QUOTEd `evg` can be proved to be a FORMP if and only if the `evg` itself is a `term` is what we have to prove once and for all as Lemma 18.

G. Solidification

We now specify what term in the theory is represented by a given INTERLISP term. Given an INTERLISP term `x`, the routine `s` (for "solidify") returns an INTERLISP object that when printed is the term represented by `x`, displayed without any abbreviations. The subroutine `sevg` ("solidify explicit value guts") computes the explicit value term represented by an `evg` object. These two routines are never used by the theorem-prover. They are defined only to make precise the map from INTERLISP terms to terms in the theory.

```
(s
  (lambda (x)
    (cond
      ((nlistp x)
       x)
      ((eq (car x) 'QUOTE)
       (sevg (cadr x)))
      (T (cons (car x)
                (for z in (cdr x) collect (s z)))))))
```

```

(sevg
  (lambda (y)
    (cond
      ((nlistp y)
        (cond
          ((litatom y)
            (cond
              ((eq y lt)
                (quote (TRUE)))
              ((eq y lf)
                (quote (FALSE)))
              (T (list (quote PACK)
                        (sevg (unpack0 y))))))
          ((lessp y 0)
            (list (quote MINUS)
                  (addl.nest (bminus y))))
          (T (addl.nest y))))
      ((eq (car y)
           lsqm)
        (cons (cadr y)
              (for z in (cddr y) collect (sevg z))))
      (T (list (quote CONS)
                (sevg (car y))
                (sevg (cdr y)))))))

```

H. Some Example INTERLISP Terms and Solidifications

Suppose that the value of `lsqm` is the INTERLISP literal atom `lsqm` (which could not represent a symbol because it has a digit as its first character). Below we exhibit, in the left-hand column, some sample INTERLISP objects (as printed by `prin4`) and, in the right-hand column, the corresponding term in our theory. In the table we have printed some of the `ADD1`-nests as integers even though `|(s x)|` never actually contains integers.

x	(s x)
(PLUS (QUOTE 1) X)	(PLUS (ADD1 (ZERO)) X)
(FN (QUOTE (A . B)))	(FN (CONS (PACK (CONS 65 0)) (PACK (CONS 66 0)))))
(QUOTE (ISQM PACK 2))	(PACK (ADD1 (ADD1 (ZERO))))
(QUOTE 0)	(ZERO)
(QUOTE (QUOTE 0))	(CONS (PACK (CONS 81 (CONS 85 (CONS 79 (CONS 84 (CONS 69 0))))) (CONS (ZERO) (PACK (CONS 78 (CONS 73 (CONS 76 0)))))
(ZERO)	(ZERO)
(QUOTE (ZERO))	(CONS (PACK (CONS 90 (CONS 69 (CONS 82 (CONS 79 0))))) (PACK (CONS 78 (CONS 73 (CONS 76 0)))))

Displayed with some abbreviations, the last four entries in the table are:

x	(s x)
(QUOTE 0)	(ZERO)
(QUOTE (QUOTE 0))	(LIST "QUOTE" (ZERO))
(ZERO)	(ZERO)
(QUOTE (ZERO))	(LIST "ZERO")

These examples are included to encourage the reader to think about our claim that if `obj` represents the term `t`, then the result of embedding `obj` in a `QUOTE` represents a term whose `MEANING`, under the standard alist for the variables in `t`, is `t`. Note that `(QUOTE 0)` represents the term `(ZERO)`; the result of embedding `(QUOTE 0)` in a `QUOTE` is `(QUOTE (QUOTE 0))`, which represents the term `(LIST "QUOTE" (ZERO))`. As claimed, the `MEANING` of `(LIST "QUOTE" (ZERO))` is `(ZERO)`. But the `INTERLISP` list that prints as `(ZERO)` is also a term that represents `(ZERO)`. The result of embedding `(ZERO)` in a `QUOTE` is `(QUOTE (ZERO))`, which represents the term `(LIST "ZERO")`. `(LIST "ZERO")` and `(LIST "QUOTE" (ZERO))` are two distinct explicit values and are thus not `EQUAL`. Nevertheless, the `MEANING` of `(LIST "ZERO")` is `(ZERO)`.

VII PROOFS OF THE LEMMAS

A. Lemmas 1 Through 7

The first important lemma is Lemma 4, which establishes that every INTERLISP term actually represents a term in the logic. Lemma 4 guarantees that $|(s\ obj)|$ is a term in our theory when $(term\ obj)$ is non-NIL. Lemma 5 states that if obj is an INTERLISP term, then $(list\ 'QUOTE\ obj)$ is an INTERLISP term.

We will first state and prove a very simple lemma as a warm-up exercise.

Lemma 1 ("addl.nest of an integer is a term"). If i is a nonnegative INTERLISP integer, then $|(addl.nest\ i)|$ is a term.

First consider an example. If i is the INTERLISP integer 3 then $|(addl.nest\ i)|$ is the explicit value term $(ADD1\ (ADD1\ (ADD1\ (ZERO))))$.

Proof. We prove Lemma 1 by induction on i .

Base case. If i is 0, $(addl.nest\ i)$ returns the value of $(list\ 'ZERO)$, which prints as $(ZERO)$.

Induction step. If i is an integer greater than 0, we may inductively assume that $|(addl.nest\ (sub1\ i))|$ is a term. Then $|(addl.nest\ i)|$ is $|(list\ 'ADD1\ (addl.nest\ (sub1\ i)))|$ which is $(ADD1\ |(addl.nest\ (sub1\ i))|)$, which is a well-formed term since $ADD1$ is a function symbol of one argument and the argument, $|(addl.nest\ (sub1\ i))|$, is a term by inductive hypothesis. Q.E.D.

Lemma 2 ("sevg of a list of integers is a term"). If obj is an INTERLISP list of nonnegative integers whose final CDR is 0, then $|(sevg\ obj)|$ is a term.

Consider another example. If obj is an INTERLISP list which prints as

$(1\ 2\ .\ 0)$,

then $|(sevg\ obj)|$ is the term

$$(CONS\ (ADD1\ (ZERO))\ (CONS\ (ADD1\ (ADD1\ (ZERO)))\ (ZERO))),$$

or more succinctly, using the abbreviations of the theory,

$$(CONS\ 1\ (CONS\ 2\ 0)).$$

Proof. The proof is by induction on the size of obj .

Base case. If obj is not a cons, then it must be 0. But $|(sevg\ 0)|$ is $|(add1.nest\ 0)|$, which is a term by Lemma 1 ("add1.nest of an integer is a term").

Induction step. If obj is a cons, then $(car\ obj)$ is a nonnegative integer and $|(sevg\ (cdr\ obj))|$ is a term, by inductive hypothesis. Since $(car\ obj)$ is not $lsqm$ (because $lsqm$ is not an integer),

$$\begin{aligned} |(sevg\ obj)| &= |(list\ 'CONS\ (sevg\ (car\ obj))\ (sevg\ (cdr\ obj)))| \\ &= |(list\ 'CONS\ (add1.nest\ (car\ obj))\ (sevg\ (cdr\ obj)))| \\ &= (CONS\ |(add1.nest\ (car\ obj))|\ |(sevg\ (cdr\ obj))|), \end{aligned}$$

which is a term since $CONS$ is a function symbol of two arguments and both arguments in the $CONS$ -expression above are themselves terms by Lemma 1 ("add1.nest of an integer is a term") and our induction hypothesis. Q.E.D.

Lemma 3 ("sevg of an evg is a term"). If $(evg\ obj)$ is non-NIL, then $|(sevg\ obj)|$ is a term. (In fact, $|(sevg\ obj)|$ in this case is an explicit value term, but we will prove that later.)

Proof. We induct on the size of obj .

Base case. Suppose that obj is not a cons. By the definition of evg , obj must therefore be an integer (i.e., recognized by $fixp$), lt , lf , or a symbolp. (a) If obj is an integer, $|(sevg\ obj)|$ is either $|(add1.nest\ obj)|$ or $(MINUS\ |(add1.nest\ (bminus\ obj))|)$, both of which are terms by Lemma 1 ("add1.nest of a integer is a term"). (b) If obj

is `lt` or `lf`, then `|(sevg obj)|` is `(TRUE)` or `(FALSE)`, both of which are terms. (c) If `(symbolp obj)` then we also have `(litatom obj)` and thus `|(sevg obj)|` is `(PACK |(sevg (unpack0 obj))|)`. But since `(unpack0 obj)` is an INTERLISP list of nonnegative integers whose final `cdr` is 0, Lemma 2 ("sevg of a list of integers is a term") tells us `|(sevg (unpack0 obj))|` is a term. Hence `(PACK |(sevg (unpack0 obj))|)` is a term.

Induction step. Suppose `obj` is a cons. We inductively assume that `|(sevg obj')|` is a term whenever `obj'` is an INTERLISP object such that `(evg obj')` holds and `obj'` is smaller than `obj` (as measured by the INTERLISP routine count). (a) if the `car` of `obj` is `lsqm`, then, by our `(evg obj)` hypothesis, `obj` must have the form `(lsqm fn arg1 ... argn)`, where `fn` is a constructor function symbol or bottom object function symbol and `n` is the arity of `fn` and `(evg argi)` holds for each `argi`. `|(sevg obj)|` is `(fn |(sevg arg1)| ... |(sevg argn)|)`, which is a term by the induction hypothesis. (b) If `car` of `obj` is not `lsqm`, then we have `(evg (car obj))` and `(evg (cdr obj))` and therefore, by our induction hypotheses, `|(sevg (car obj))|` and `|(sevg (cdr obj))|` are terms. But `|(sevg obj)|` is `(CONS |(sevg (car obj))| |(sevg (cdr obj))|)`, which is also a term. Q.E.D.

Lemma 4 ("s of a term is a term"). If `(term p obj)`, then `|(s obj)|` is a term.

Proof. We induct on the size of `obj`.

Base case. Given that `obj` is not a cons and `(term p obj)` is non-NIL, we know `(symbolp obj)`. Thus, `|obj|` (i.e., the print name of `obj`) is a character sequence satisfying the restrictions on variable symbols in our logic. But `|(s obj)|` is `|obj|` and thus a term (in particular, a variable).

Induction step. Suppose `obj` is a cons. (a) If `(car obj)` is `'QUOTE`, then `(term p obj)` implies that `obj` must have the form `(QUOTE obj')` where `(evg obj')`. `|(s obj)|` is then `|(sevg obj')|`, which is a term by Lemma 3 ("sevg of an evg is a term"). (b) If `(car obj)` is not `'QUOTE`, then `obj` has the form `(fn arg1 ... argn)` where `n` is the

nonnegative arity of the function symbol fn and $(term\ arg_i)$ for each i . $| (s\ obj) |$ is $(fn\ |(s\ arg_1)|\ \dots\ |(s\ arg_n)|)$, which is a term since each $| (s\ arg_i) |$ is inductively a term. Q.E.D.

Lemma 5 ("QUOTEd term is a term"). If $(term\ obj)$, then $(term\ (list\ 'QUOTE\ obj))$.

Proof. By the definition of $term$, $(term\ (list\ 'QUOTE\ obj))$ is equivalent to $(evg\ obj)$. Thus it suffices to show that $(term\ obj)$ implies $(evg\ obj)$.

The proof is by induction on the size of obj .

Base case. Suppose obj is not a cons. Then from $(term\ obj)$, we have $(symbolp\ obj)$, which guarantees $(evg\ obj)$.

Induction step. Suppose obj is a cons.

(a) If $(car\ obj)$ is $'QUOTE$, then, by $(term\ obj)$, we have $(listp\ (cdr\ obj))$, $(evg\ (cadr\ obj))$ and that $(caddr\ obj)$ is NIL. Since $(car\ obj)$ is not $lsqm$, $(evg\ obj)$ is equivalent to the conjunction of $(evg\ (car\ obj))$ and $(evg\ (cdr\ obj))$. The first is immediate. The second is equivalent to the conjunction of $(evg\ (cadr\ obj))$ and $(evg\ (caddr\ obj))$ (both of which are also immediate) provided $(cadr\ obj)$ is not $lsqm$. But $(cadr\ obj)$ cannot be $lsqm$ because $(evg\ (cadr\ obj))$ is non-NIL, while $(evg\ lsqm)$ is NIL (because $lsqm$ is a literal atom, distinct from lt and lf , and not a symbolp).

(b) If $(car\ obj)$ is not $'QUOTE$, then obj has the form $(fn\ arg_1\ \dots\ arg_n)$, where n is the length of $(cdr\ obj)$, $(arity\ fn)$ is n , and $(term\ arg_i)$ for each i . Provided neither fn nor any arg_i is $lsqm$, $(evg\ obj)$ is equivalent to the conjunction of $(evg\ fn)$, $(evg\ arg_1)$, ..., $(evg\ arg_n)$, and $(evg\ NIL)$. But $(evg\ fn)$ follows from the fact that $(arity\ fn)$ is $(length\ (cdr\ obj))$, which is a nonnegative integer, and $(arity\ fn)$ is a nonnegative integer only if $(symbolp\ fn)$. Each $(evg\ arg_i)$ follows from our inductive hypotheses and $(term\ arg_i)$. $(evg\ NIL)$ is immediate. Thus, we must show that neither fn nor any arg_i is $lsqm$. But since $(evg\ lsqm)$ is NIL, this must be the case. Q.E.D.

Lemma 6 ("unique representation of explicit values"). For each explicit value term t , there exists (modulo INTERLISP equality) exactly one INTERLISP object v such that $(\text{evg } v)$ is non-NIL and $|(\text{sevg } v)|$ is t .

We do not prove this lemma here. The proof, by induction on the structure of explicit values, is tedious but straightforward. We indicate how the proof goes by considering the case for an explicit value of the form $(\text{CONS } t_1 t_2)$. By induction hypothesis, the explicit values t_1 and t_2 are uniquely represented, say by v_1 and v_2 . The existence part of the proof is easy. The $\text{evg } (\text{cons } v_1 v_2)$ represents $(\text{CONS } t_1 t_2)$: since v_1 is an evg , it is not lsqm and so $|(\text{sevg } (\text{cons } v_1 v_2))|$ is $(\text{CONS } |(\text{sevg } v_1)| |(\text{sevg } v_2)|)$ which is $(\text{CONS } t_1 t_2)$. The uniqueness argument is more tedious. Suppose that for some $\text{evg } v$ not equal to $(\text{cons } v_1 v_2)$, $|(\text{sevg } v)|$ is $(\text{CONS } t_1 t_2)$. Consider the structure of v . Suppose v is not a list. Then the function symbol of $|(\text{sevg } v)|$ is either TRUE , FALSE , PACK , MINUS , ADD1 , or ZERO , contradicting the assumption that it is CONS . Suppose v is a list whose car is lsqm . Then the cadr of v must be CONS since the function symbol of $|(\text{sevg } v)|$ is CONS . But $(\text{evg } v)$ requires that the cadr of such a v not be CONS , so such a v is not an evg . Thus, v must be a list whose car is not lsqm . But then its car must be an evg representing t_1 and its cdr an evg representing t_2 . v_1 and v_2 are the only evgs with that property. Thus, v is equal to $(\text{cons } v_1 v_2)$.

In general, the key to the uniqueness argument is that evg checks that lsqm is not used to "counterfeit" terms that have more efficient representations. Thus, $(\text{list } \text{lsqm } \text{'CONS } v_1 v_2)$, the counterfeit representation of $(\text{CONS } t_1 t_2)$, fails to be an evg . Similarly, evg checks that lsqm is not used to represent ADD1 terms, (ZERO) , PACK terms of $\text{LEGAL.CHAR.CODE.SEQs}$, or MINUS terms other than $(\text{MINUS } 0)$.

We next prove a result similar to but stronger than Lemma 3 ("sevg of an evg is a term").

Lemma 7 ("sevg of an evg is an explicit value"). If $(\text{evg } v)$ then $|(\text{sevg } v)|$ is an explicit value term.

Proof. By Lemma 3 we know $|(sevg\ v)|$ is a term. To prove that it is an explicit value term we must prove that (i) there are no variables in it, (ii) there are no function symbols other than TRUE, FALSE, and shell constructor and bottom object function symbols, and (iii) if the term t occurs as the i th argument to some constructor function $const$ in $|(sevg\ v)|$, then the function symbol of t must be one of those recognized (or, depending on the type restriction, not recognized) by the finite set of recognizers specified for the i th component of $const$.

The proof of each of these facts is by induction on the size of v . Proving that $|(sevg\ v)|$ contains no variables is immediate from inspection of $sevg$ and one's inductive hypotheses. Proving that all the function symbols are as specified by (ii) is immediate from inspection of $sevg$ and induction except for the case where v is a list whose car is $lsqm$. In this case $(sevg\ v)$ is $(cons\ (cadr\ v)\ \dots)$ and so might appear to have an arbitrary function symbol when printed. But $(evg\ v)$ ensures us that $(cadr\ v)$ is a shell constructor or bottom object, since it must be found on $(shell.state)$. As for (iii), there are three interesting cases: $|(sevg\ v)|$ is an $(ADD1\ \dots)$, a $(MINUS\ \dots)$, or a user-defined shell constructor term. No other primitive shells have type restrictions on their components. A trivial case analysis shows that $add1.nest$ produces only terms satisfying $NUMBERP$, so the first two cases are immediate. When the third case obtains, the car of v is $lsqm$ and we must prove that the function symbol of each of the arguments satisfies the corresponding type restriction. But $(evg\ v)$ checks precisely that by insuring that the function symbol of each argument is a member of (or, depending on the type restriction, not a member of) the finite set specified by $(shell.state)$. Q.E.D.

B. Lemmas 8 Through 18

We now prove a series of lemmas that let us move from reductions in a history to computations in INTERLISP. Our main goal in this section is Lemma 18.

Lemma 8 ("LISTP iff listp and not lsqm"). If (evg x), then the reduction of (LISTP |(sevg x)|) is (TRUE) if and only if (listp x) is non-NIL and (car x) is not lsqm.

Proof. If (listp x) is NIL or (car x) is lsqm, then the function symbol of |(sevg x)| is either TRUE, FALSE, a bottom object, or a shell constructor other than CONS. Thus, the reduction of (LISTP |(sevg x)|) is (FALSE). To prove the lemma in the other direction, suppose (listp x) is non-NIL and (car x) is not lsqm. Then (LISTP |(sevg x)|) is (LISTP (CONS |(sevg (car x))| |(sevg (cdr x))|))), whose reduction is (TRUE). Q.E.D.

Lemma 9 ("CDR is cdr when not lsqm"). If (evg x) and (listp x) and the car of x is not lsqm, then the reduction of (CDR |(sevg x)|) is |(sevg (cdr x))|.

Proof. The proof is trivial. Under the conditions given,

(CDR |(sevg x)|)

is

(CDR (CONS |(sevg (car x))| |(sevg (cdr x))|))),

whose reduction is |(sevg (cdr x))|. Q.E.D.

We state, without proof, the analogous lemma for CAR and car.

Lemma 10 ("CAR is car when not lsqm"). If (evg x) and (listp x) and the car of x is not lsqm, then the reduction of (CAR |(sevg x)|) is |(sevg (car x))|.

Lemma 11 ("EQUAL iff identical") If (evg x) and (evg y) then the reduction of (EQUAL |(sevg x)| |(sevg y)|) is (TRUE) if and only if x and y are equal.

Proof. Recall that the reduction of the equation of two explicit values is (TRUE) if and only if the two terms are identical. In addition, by Lemma 7 ("sevg of an evg is an explicit value"), |(sevg x)| and |(sevg y)| are both explicit values.

Suppose the reduction of (EQUAL |(sevg x)| |(sevg y)|) is (TRUE). Then |(sevg x)| and |(sevg y)| are identical. So x and y are equal by Lemma 6 ("unique representation of explicit values"). In the other direction, the reduction is immediate. Q.E.D.

Lemma 12 ("LEGAL.CHAR.CODE.SEQ iff legal.char.code.seq"). If (evg v), then the reduction of (LEGAL.CHAR.CODE.SEQ |(sevg v)|) is (TRUE) if and only if (legal.char.code.seq v) is non-NIL.

Recall that LEGAL.CHAR.CODE.SEQ checks that its argument is a LISTP whose first member is not in (ILLEGAL.FIRST.CHAR.CODES), is a subset of (LEGAL.CHAR.CODES). and terminates in a 0. legal.char.code.seq checks the same things at the level of INTERLISP.

We do not prove this lemma here. However, we will indicate how the proof goes. Our proof involves the following two lemmas:

- (1) If (evg c), (evg x), and every element of x is an evg, then the reduction of (MEMBER |(sevg c)| |(sevg x)|) is (TRUE) if and only if (member c x) is non-NIL.
- (2) If (evg x), (evg y), and y is a lists of evgs, then the reduction of

```
*a (AND (SUBSETP |(sevg x)| |(sevg y)|)
      (EQUAL (ZERO)
              (IF (LISTP |(sevg x)|)
                  (CDR (LAST |(sevg x)|))
                  |(sevg x)|)))
```

is (TRUE) if and only if

```
*b (and (for c in x always (member c y))
      (equal 0
              (cond ((listp x) (cdr (last x)))
                    (t x)))).
```

To use these two lemmas in the proof of Lemma 12 ("LEGAL.CHAR.CODE.SEQ iff legal.char.code.seq") it is only necessary to observe that (LEGAL.CHAR.CODES) and (ILLEGAL.FIRST.CHAR.CODES) reduce to |(sevg (legal.char.codes))| and |(sevg (illegal.char.codes))|. Furthermore, since both (legal.char.codes) and (illegal.first.char.codes) are lists of integers, they are also lists of evgs.

Lemma 13 ("SYMBOLP iff symbolp"). If (evg v) then the reduction of (SYMBOLP |(sevg v)|) is (TRUE) if and only if (symbolp v) is non-NIL.

Proof. If the reduction of the SYMBOLP expression is (TRUE), we know the reduction of (LITATOM |(sevg v)|) is (TRUE) and that the reduction of (LEGAL.CHAR.CODE.SEQ (UNPACK |(sevg v)|)) is (TRUE). But by sevg and evg, if the reduction of the LITATOM expression is (TRUE) then either v is a literal atom and (legal.char.code.seq (unpack0 v)) holds so (symbolp v) holds, or else v is a list whose car is lsqm, whose cadr is PACK, and whose caddr is rejected by legal.char.code.seq. We can prove the latter cannot happen, because for such a v (UNPACK |(sevg v)|) is |(sevg (caddr v))| and hence the reduction of (LEGAL.CHAR.CODE.SEQ |(sevg (caddr v))|) is (TRUE), and so Lemma 12 ("LEGAL.CHAR.CODE.SEQ iff legal.char.code.seq") assures us that (legal.char.code.seq (caddr v)) is non-NIL, contradicting the hypothesis that (caddr v) was rejected by legal.char.code.seq. The argument in the other direction is similar. Q.E.D.

Lemma 14 ("if PLISTP, then plistp and list of evgs"). If (evg x) and the reduction of (PLISTP |(sevg x)|) is (TRUE), then (plistp x) is non-NIL and every element of x is an evg.

The definition of the function PLISTP, from [1], is:

```
Definition.  
(PLISTP L)  
=  
(IF (LISTP L)  
    (PLISTP (CDR L))  
    (EQUAL L "NIL")).
```

Observe that if c is reducible and (FORM.LSTP c) reduces to (TRUE), then so does (PLISTP c). We introduce PLISTP because only to make it easier to establish later that if (FORM.LSTP |(sevg x)|) reduces to (TRUE) then x is a proper list of evgs. We now prove Lemma 14.

Proof. We induct on x.

Base case. If x is not a cons, then the reduction of (LISTP |(sevg x)|) is (FALSE) by Lemma 8 ("LISTP iff listp and not lsqm"). Since the

reduction of (PLISTP |(sevg x)|) is (TRUE), (EQUAL "NIL" |(sevg x)|) must reduce to (TRUE), which implies x is NIL by Lemma 11 ("EQUAL iff identical"). But if x is NIL, then our conclusion holds.

Induction step. If x is a cons, inductively assume that if the reduction of (PLISTP |(sevg (cdr x))|) is (TRUE), then (plistp (cdr x)) is non-NIL and every element of (cdr x) is an evg. We must show (plistp x) and that every element of x is an evg. We first observe that (car x) cannot be lsqm, for if it were, (LISTP |(sevg x)|) would reduce to (FALSE) by Lemma 8 ("LISTP iff listp and not lsqm") and so the reduction of (EQUAL "NIL" |(sevg x)|) would have to be (TRUE), but is not, by Lemma 11 ("EQUAL iff identical") and the observation that NIL is not identical to x. So, we have that the reduction of (LISTP |(sevg x)|) is (TRUE) and thus the reduction of (PLISTP (CDR |(sevg x)|)) is also. But then the reduction of (PLISTP |(sevg (cdr x))|) is (TRUE) by Lemma 9 ("CDR is cdr when not lsqm"), so we get, from our induction hypothesis, that (plistp (cdr x)) is non-NIL and every element of (cdr x) is an evg. The former guarantees that (plistp x) is non-NIL, and the latter guarantees that every element of x is an evg if we can establish that (car x) is an evg. But this follows from (evg x), given that (car x) is not lsqm. Q.E.D.

Lemma 15 ("LENGTH is length when list of evgs"). If (evg x) and x is a list of evgs, then the reduction of (LENGTH |(sevg x)|) is |(sevg (length x))|.

The proof, by induction on x, is omitted because it is so similar to the proof of the preceding Lemma 14.

For the remainder of this section, let us assume that the state of the INTERLISP machine (in particular, the definitions of arity and shell.state) reflect the history T_2 of the Metatheorem.

Lemma 16 ("ARITY is arity"). If (symbolp x), then the reduction in both $T_{1.5}$ and $MD[T_2]$ of (ARITY |(sevg x)|) is |(sevg (arity x))|.

Proof. The function symbols TRUE, NOT, IMPLIES, PLUS, f_5 , ..., f_m are, by definition, the only functions mentioned in the axioms of T_2 .

If x is one of these symbols, the theorem holds by the definitions of arity and ARITY. If x is not one of these symbols, both are "NIL".
Q.E.D.

Lemma 17 ("if FORMP.LSTP, then list of FORMPs"). If x is a proper list of evgs and the reduction of $(\text{FORMP.LSTP } |(\text{sevg } x)|)$ in T_2 is (TRUE), then for each element arg in x , the reduction of $(\text{FORMP } |(\text{sevg } \text{arg})|)$ is (TRUE).

Proof. The proof is by induction on x .

Base case. If x is not a cons, then the reduction of $(\text{LISTP } |(\text{sevg } x)|)$ is (FALSE) by Lemma 8 ("LISTP iff listp and not lsqm") so by our FORMP.LSTP hypothesis we know the reduction of $(\text{EQUAL } |(\text{sevg } x)| \text{ "NIL"})$ is (TRUE), so x must be NIL by Lemma 6 ("unique representation of explicit values") and our conclusion is vacuously true.

Induction step. If x is a cons, we can inductively assume that if $(\text{cdr } x)$ is a proper list of evgs and the reduction of $(\text{FORMP.LSTP } |(\text{sevg } (\text{cdr } x))|)$ is (TRUE), then for every arg in $(\text{cdr } x)$, the reduction of $(\text{FORMP } |(\text{sevg } \text{arg})|)$ is (TRUE). We must prove that if x is a proper list of evgs and the reduction of $(\text{FORMP.LSTP } |(\text{sevg } x)|)$ is (TRUE), then for each element arg of x , the reduction of $(\text{FORMP } |(\text{sevg } \text{arg})|)$ is (TRUE). Observe that $(\text{car } x)$ is not lsqm, for if it were, $(\text{LISTP } |(\text{sevg } x)|)$ would reduce to (FALSE) by Lemma 8 but the reduction of $(\text{EQUAL } |(\text{sevg } x)| \text{ "NIL"})$ is (FALSE) by a unique representation of explicit values argument. So the reduction of $(\text{LISTP } |(\text{sevg } x)|)$ is (TRUE) and we infer that the reduction of both $(\text{FORMP } (\text{CAR } |(\text{sevg } x)|))$ and $(\text{FORMP.LSTP } (\text{CDR } |(\text{sevg } x)|))$ is (TRUE). Moving the CAR and CDR inside, using Lemmas 9 and 10, we determine that the reduction of both $(\text{FORMP } |(\text{sevg } (\text{car } x))|)$ and $(\text{FORMP.LSTP } |(\text{sevg } (\text{cdr } x))|)$ is (TRUE), and by using our induction hypothesis, we establish that the reduction of $(\text{FORMP } |(\text{sevg } \text{arg})|)$ is (TRUE) when arg is $(\text{car } x)$ or an element of $(\text{cdr } x)$, which is to say, for each element arg of x . Q.E.D.

We now prove the first of the two lemmas used directly in the proof that our implementation of the Metatheorem is correct. Lemma 18

establishes that if (FORMP c) reduces to (TRUE) and c is represented by (list 'QUOTE obj), then obj itself represents a term. In fact, the lemma holds in the other direction too, but we do not need it or prove it in that direction.

Lemma 18 ("FORMP of a QUOTEd evg iff term"). If (term (list 'QUOTE obj)) and the reduction of (FORMP |(s (list 'QUOTE obj))|) in $T_{1.5}$ (equivalently, MD[T₂]) is (TRUE), then (term obj).

Proof. Observe that the first hypothesis is equivalent to (evg obj) and the second hypothesis is equivalent to the supposition that the reduction of (FORMP |(sevg obj)|) is (TRUE). The proof is by induction on the size of obj.

Base case. If obj is not a cons, then by Lemma 8 ("LISTP iff listp and not lsqm") we know (LISTP |(sevg obj)|) reduces to (FALSE). Thus, by our FORMP hypothesis, we know the reduction of (SYMBOLP |(sevg obj)|) is (TRUE). Hence, by Lemma 13 ("SYMBOLP iff symbolp") we know (symbolp obj), which guarantees (term obj).

Induction step. obj is a cons. Consider (car obj).

(a). If (car obj) is 'QUOTE, then we must show (i) (listp (cdr obj)), (ii) (null (cddr obj)), and (iii) (evg (cadr obj)). The reduction of (FORMP |(sevg obj)|) is the reduction of (FORMP (CONS "QUOTE" |(sevg (cdr obj))|)), which means that the reduction of both (LISTP |(sevg (cdr obj))|) and (EQUAL "NIL" (CDR |(sevg (cdr obj))|)) is (TRUE). Lemma 8 ("LISTP iff listp and not lsqm") is sufficient to ensure (i). In addition, Lemma 8 tells us (cadr obj) is not lsqm. Thus the reduction of (EQUAL "NIL" (CDR |(sevg (cdr obj))|)) is the reduction of (EQUAL "NIL" |(sevg (cddr obj))|) by Lemma 9 ("CDR is cdr when not lsqm"). But then (cddr obj) is NIL, by a unique representation of explicit values argument. So (ii) holds. As for (iii), note that since both obj and (cdr obj) are listps and neither (car obj) nor (cadr obj) is lsqm, (evg obj) establishes (evg (cdr obj)) which in turn gives us (evg (cadr obj)), which is (iii).

(b). If (car obj) is not 'QUOTE, then we need to show (i) (plistp (cdr obj)), (ii) (equal (length (cdr obj)) (arity (car obj))), and (iii) (for z in (cdr obj) always (term p z)). Our hypotheses are (evg obj) and that the reduction of (FORMP |(sevg obj)|) is (TRUE). First we establish that (car obj) is not lsqm. Suppose it were. Then the reduction of (LISTP |(sevg obj)|) would be (FALSE) by Lemma 8 ("LISTP iff listp and not lsqm") and thus the reduction of (FORMP |(sevg obj)|) would be (FALSE) since the reduction of (SYMBOLP |(sevg obj)|) is (FALSE) by Lemma 13 ("SYMBOLP iff symbolp"). Thus, (car obj) is not lsqm and the reduction of (LISTP |(sevg obj)|) is (TRUE).

Thus, our hypothesis that the reduction of (FORMP |(sevg obj)|) is (TRUE) gives us that the reductions of both

(EQUAL (ARITY (CAR |(sevg obj)|))
(LENGTH (CDR |(sevg obj)|)))

and

(FORM.LSTP (CDR |(sevg obj)|))

are (TRUE). Hence the reduction of

(PLISTP (CDR |(sevg obj)|))

is (TRUE). By lemmas 9 and 10 ("CDR is cdr when not lsqm" and "CAR is car when not lsqm"), and Lemma 16 ("ARITY is arity"), the reduction of each of the following is (TRUE):

(EQUAL |(sevg (arity (car obj)))|
(LENGTH |(sevg (cdr obj))|)),

(FORM.LSTP |(sevg (cdr obj))|),

and

(PLISTP |(sevg (cdr obj))|).

Thus, by Lemma 14 ("if PLISTP, then plistp and list of evgs") we know (plistp (cdr obj)) is non-NIL (which establishes (i)) and that every element of (cdr obj) is an evg. But now we can apply Lemma 15 ("LENGTH

is length when list of evgs") and Lemma 17 ("if FORM.LSTP, then list of FORMPs") to get that the reduction of

$$\text{(EQUAL |(sevg (arity (car obj)))|} \\ \text{|(sevg (length (cdr obj)))|})$$

is (TRUE) and that for every arg in (cdr obj), the reduction of (FORMP |(sevg arg)|) is (TRUE). The former is sufficient to guarantee (ii), by a unique representation of explicit values argument. The latter guarantees (iii) since, by induction hypothesis, when the reduction of (FORMP |(sevg arg)|) is (TRUE) for an arg whose count is smaller than obj, then (termp arg) is non-NIL. Q.E.D.

C. Lemma 19

We now prove the final lemma used in the argument that our implementation of the Metatheorem is correct. Lemma 19 establishes that if obj represents term t, then (list 'QUOTE obj) represents a quotation of t.

Lemma 19 ("QUOTED term is a quotation"). If (termp obj), then |(s (list 'QUOTE obj))| is a quotation of |(s obj)|.

Proof. By the definition of s, (s (list 'QUOTE obj)) is (sevg obj).

The proof is by induction on the size of obj.

Base Case. If obj is not a cons, then from (termp obj), we have that (symbolp obj). But |(sevg obj)| is then "obj" and |(s obj)| is obj.

Induction step. If obj is a cons, consider (car obj).

(a). Suppose the car of obj is 'QUOTE. |(s obj)| is |(sevg (cadr obj))|. From (termp obj), we infer (evg (cadr obj)). From Lemma 7 ("sevg of an evg is an explicit value"), we infer that |(sevg (cadr obj))| is an explicit value. Hence one quotation of |(sevg (cadr obj))| is (LIST "QUOTE" |(sevg (cadr obj)))|, which we now show is in fact |(sevg obj)|. Since (termp obj) and the car of obj is 'QUOTE, (cdr obj)

is a list, (cadr obj) is an evg (and thus not lsqm) and (cddr obj) is NIL. Thus, |(sevg obj)| is (CONS "QUOTE" (CONS |(sevg (cadr obj))| "NIL")), which is (LIST "QUOTE" |(sevg (cadr obj))|).

(b). If (car obj) is not 'QUOTE, then obj has the form (fn arg₁ ... arg_n), where n is the length of (cdr obj), (arity fn) is n, and (termp arg_i) for each i. Hence |(sevg obj)| is (LIST "fn" |(sevg arg₁)| ... |(sevg arg_n)|) since no arg_i is lsqm. By inductive hypothesis, each |(sevg arg_i)| is a quotation of arg_i. Q.E.D.

AD-A094 385

SRI INTERNATIONAL MENLO PARK CA COMPUTER SCIENCE LAB F/G 12/1
METAFUNCTIONS: PROVING THEM CORRECT AND USING THEM EFFICIENTLY --ETC(U)
DEC 79 R S BOYER, J S MOORE N00014-75-C-0816
SRI/CSL-108 NL

UNCLASSIFIED

2 of 2

AL
AD741595

END

DATE

FILMED

2 81

DTIC

VIII EFFICIENT COMPUTATION ON EXPLICIT VALUES

To use metafunctions efficiently we need a method for rapidly computing the object `objd` such that the term represented by `(list 'QUOTE objd)` is the reduction of `(simp |(s (list 'QUOTE objc))|)`, when `simp` is an explicit value preserving function.

Every time an explicit value preserving function `fn` is defined in our theorem-proving system, we store in the definition cell of the INTERLISP literal atom `lfn` a routine with the following property:

If `fn` takes `n` arguments and `c1, ..., cn` are explicit values represented by `(list 'QUOTE obj1)`, ..., `(list 'QUOTE objn)` respectively, then `(list 'QUOTE (lfn obj1 ... objn))` represents the reduction of `(fn c1 ... cn)`.

Below we show how we generate the INTERLISP routine for `lfn`. We leave to the reader the proof that the program constructed has the desired property. In most cases the proof is straightforward, given the lemmas already proved. The statement of this lemma makes no claim about the efficiency of `lfn` but we will discuss efficiency after indicating how the routines are generated.

Consider first those functions that are built in. A suitable definition of `ltrue`, the routine corresponding to `TRUE`, is `(lambda () lt)`. `FALSE` is similar. The routine for `EQUAL` is `(lambda (x y)(cond ((equal x y) lt)(T lf)))` -- i.e., it returns `lt` if the two evgs are equal INTERLISP objects and `lf` otherwise. The routine for `IF` should return the value of its third argument if that of its first is `lf` and otherwise return the value of its second argument. Thus, `(lif x y z)` should be macro-expanded into `(cond ((eq x lf) z)(T y))`. Any function definition of `lif` must first evaluate `x` in the environment of the calling procedure and then selectively evaluate either `y` or `z` in the

environment of the calling procedure. We explain why `lif` must not evaluate all three arguments when we examine the case for recursive functions.

Before proceeding, recall the property that `lfunctions` are supposed to have. Consider `lequal`. It is supposed to be the case that if c_1 and c_2 are explicit values represented by `(list 'QUOTE obj1)` and `(list 'QUOTE obj2)`, then the reduction of `(EQUAL c1 c2)` is represented by `(list 'QUOTE (lequal obj1 obj2))`. But Lemma 11 ("EQUAL iff identical") establishes that `(EQUAL c1 c2)` reduces to `(TRUE)` if and only if `(equal obj1 obj2)` is non-NIL, and `(list 'QUOTE (lequal obj1 obj2))` represents `(TRUE)` or `(FALSE)` according to whether `(equal obj1 obj2)`. So `lequal` is has the property claimed. The proofs of the other `lfunctions` are similar.

The `lfunctions` for the various primitive shell functions are defined similarly so we will only exhibit the definitions of `llistp`, `lcons`, `lcar`, and `lcdr`.

```
(llistp (lambda (x)
  (cond ((and (listp x)
    (not (eq (car x) lsqm)))
    lt)
    (T lf))))

(lcons (x y) (cons x y))

(lcar (lambda (x)
  (lif (llistp x) (car x) 0)))

(lcdr (lambda (x)
  (lif (llistp x) (cdr x) 0)))
```

Observe how their correctness follows immediately from such lemmas as 8 ("`LISTP` iff `lisp` and not `lsqm`") and 9 ("`CDR` is `cdr` when not `lsqm`").

Now we consider functions introduced by the user, either via the shell principle or the principle of definition. Suppose we have correctly obtained the INTERLISP routines for all the previously introduced explicit value preserving functions and are now considering some newly introduced function `fn`.

Suppose `fn` is introduced by the shell principle. If `fn` is a recognizer, `lfn` is the INTERLISP function that returns `lt` or `lf` according to whether its argument is a `listp` whose `car` is `lsqm` and whose `cadr` is the name of the shell constructor or bottom object of the class. If `fn` is a bottom object function, `lfn` returns the list of length 2 with `lsqm` as its `car` and the bottom object name as its `cadr`. If `fn` is a constructor function, `lfn` returns a list of length $n+2$, with `lsqm` as its `car`, the constructor function name as its `cadr`, and n elements in the `cddr`. The i th element of the `cddr` is just the i th argument to `lfn` if that argument satisfies the i th type restriction and otherwise is the `evg` representing the i th default value. Type restrictions are checked by calling the already obtained routines corresponding to the finite set of recognizers that must approve or disapprove of the argument. The `evg` for the default value is obtained by calling the already defined routine for it. Finally, if `fn` is the i th accessor function of a shell, `lfn` returns the $i+2$ nd element of its argument if its argument satisfies the recognizer routine for its shell class (but is not the representation of the optional bottom object), and otherwise returns the `evg` for the i th default value.

If `fn` is none of the above, it must be a defined function. Its definition must be of the form `(EQUAL (fn  $x_1$  ...  $x_n$ ) body)`, where every function symbol in `body` (other than `fn`) is explicit value preserving. Thus, for each such function symbol we have a routine. Let `lbody` be the INTERLISP expression obtained by replacing uses of `fn` in `body` as a function symbol by `lfn` and uses of other function symbols in `body` by the name of the corresponding routine. Define the INTERLISP routine `lfn` with `(lambda ( $x_1$  ...  $x_n$ ) lbody)`. For example, given the definition of `APPEND`:

```

Definition.
(APPEND X Y)
-
(IF (LISTP X)
    (CONS (CAR X) (APPEND (CDR X) Y))
    Y),

```

the definition for `lappend` is:

```

(lappend (lambda (x y)
  (lif (llistp x)
    (lcons (lcar x) (lappend (lcdr x) y))
    y))).

```

lfn always terminates and has the desired property. The key observation is that a certain measure of the arguments decreases on every recursive call (namely the measure that "lifts" the evgs back into the theory and measures them with the function used to justify the definition of fn). The proof relies upon the fact that (lif x y z) only evaluates y when x evaluates to non-lf, and only evaluates z otherwise. The reason is that the measure justifying the admission of fn was proved to decrease in all recursive calls of fn in the true-branch of the IF provided the test was true, and was proved to decrease in the false-branch provided the test was false. Thus, the inductive hypothesis that the computation of y is correct and terminates can only be obtained in the case where x is known to have computed to non-lf.

This concludes the sketch of how we can generate routines for each explicit value preserving function in the theory.

For efficiency the theorem-prover actually includes built-in definitions of PLUS and LESSP and hand-coded versions of lplus and llessp that take advantage of the hardware for operating on evgs representing Peano integers and avoid the necessity for recursion by SUB1. However, once one gets away from the hardware level the functions one defines can usually take advantage of the same algorithms an efficient procedure might.

While the code we generate for user-defined functions is equivalent to that sketched above, we actually compile it after optimizing it in four ways.

The first optimization technique is to expand certain built-in functions to avoid incurring an INTERLISP procedure call in cases where the compiled code represents only a few machine instructions. For example we expand references to such basic functions as IF and LISTP by expanding the definitions of the corresponding INTERLISP procedures "inline."

The second optimization technique eliminates the tension between INTERLISP's convention of testing against NIL and the theory's convention of testing against (FALSE). In general, the code for (LISTP X) tests (and (listp x) (neq (car x) lsqm)) against NIL and branches to return lf or lt accordingly. According to the optimization presented in the previous paragraph, if (LISTP X) occurs in the test of an IF, we might merely expand (LISTP X) and then test the result against lf and branch accordingly. But it is inefficient for the expansion of (LISTP X) to branch on NIL to return lt or lf only for IF to test the result against lf and branch again. By keeping track of whether the results of built-in predicates such as LISTP, EQUAL, and AND are only being tested in IFs, our expansion avoids the redundant returning of lt and lf and the testing against lf.

The third optimization technique eliminates much of the testing of listp and lsqm that would otherwise be necessary in list processing. In general, the code for (CAR X) expands to

```
(cond ((and (listp x) (neq (car x) lsqm))
      (car x))
      (t 0)).
```

However, if we can prove that the tests governing that occurrence of (CAR X) imply (LISTP X), then (CAR X) can be expanded into (car x) -- which compiles into a single machine instruction. Similarly, in expanding (EQUAL X Y), which in general must test (equal x y), we actually test (eq x y) -- which requires a single machine instruction -- when we know that one of X or Y is a QUOTEd literal atom.

The three optimization techniques above produce the following code from the definition of APPEND:

```
(lappend (lambda (x y)
  (cond ((and (listp x) (neq (car x) lsqm))
        (cons (car x) (lappend (cdr x) y)))
        (T y)))).
```

The fourth optimization technique eliminates the expense of recomputing common subexpressions in the body of a definition during any evaluation of that body. To each common subexpression we allocate a temporary variable that is set to the value of the subexpression the first time it is evaluated. Because we do not put the code into "COND-normal form," thereby removing all conditionals from the tests of other conditionals, a given occurrence *o* of a subexpression *s* of a definition can have the property that during some evaluations of the body, a prior occurrence of *s* been evaluated before the occurrence at *o* is reached, while on other evaluations of the body, no prior occurrence of *s* has been evaluated before the evaluation at *o*. We therefore initialize our temporary variables with the atom *lX* (which is not an evg) and during the evaluation of a body test the temporary variables against *lX* in those situations in which our optimizer could not determine that the variable had been previously set. We do not save the values of *car/cdr* nests since they are compiled efficiently.

The INTERLISP compiler compiles certain forms of recursion as iteration. Thus, the second call of *BAGINT* in the compiled version of that function is actually implemented as a PDP-10 jump instruction rather than a true recursion.

Here is the INTERLISP code that is compiled for the definition of *CANCEL* discussed in Section II. Each *setq* requires one instruction. Each *neq* test requires one instruction.


```

(lcancel (lambda (x)
  (prog ((2temp1 (quote 1X)) (2temp2 (quote 1X))
        (2temp3 (quote 1X)) (2temp4 (quote 1X))
        (2temp5 (quote 1X)) (2temp6 (quote 1X)))
    (return
      (cond
        ((and (setq 2temp6 (neq (lequality? x) 1f))
              (setq 2temp5 (neq (lplus.tree? (lcar (cdr x))) 1f))
              (neq (lplus.tree? (setq 2temp4 (lcar (cddr x))) 1f))
              (list
                (quote EQUAL)
                (lplus.tree
                  (lbagdiff
                    (setq 2temp3 (lfringe (cadr x)))
                    (setq 2temp2
                      (lbagint 2temp3
                        (setq 2temp1 (lfringe (caddr x)))))))
                  (lplus.tree (lbagdiff 2temp1 2temp2))))
              ((and 2temp6
                    (cond ((neq 2temp5 (quote 1X)) 2temp5)
                          (T (neq (lplus.tree? (lcar (cdr x))) 1f)))
                    (neq (lmember (cond ((neq 2temp4 (quote 1X)) 2temp4)
                                         (T (setq 2temp4 (lcar (cddr x)))))
                          (setq 2temp3 (lfringe (cadr x))))
                    1f))
              (cons
                (quote IF)
                (cons
                  (list (quote NUMBERP) 2temp4)
                  (cons (cons (quote EQUAL)
                              (cons (lplus.tree (ldelete 2temp4 2temp3))
                                    (quote ((ZERO))))
                            (quote ((FALSE))))))
              ((and 2temp6
                    (neq (lplus.tree? (lcar (lcdr (cdr x)))) 1f)
                    (neq (lmember (cadr x) (setq 2temp1 (lfringe (caddr x)))
                          1f))
                    (cons (quote IF)
                          (cons (list (quote NUMBERP) (cadr x))
                                (cons (list (quote EQUAL)
                                              (quote (ZERO))
                                              (lplus.tree (ldelete (cadr x) 2temp1)))
                                      (quote ((FALSE))))))
              (T x))))))

```

For example, if obj is the INTERLISP list structure that prints as

```
(EQUAL (PLUS (PLUS A I) (PLUS B K))
        (PLUS J (PLUS K (PLUS I X))))),
```

then (lcancel obj) is the INTERLISP list structure

```
(EQUAL (PLUS A B)
        (PLUS J X)).
```

If obj is the INTERLISP list structure (EQUAL A (PLUS A B)) then
(lcancel obj) is the INTERLISP list structure

```
(IF (NUMBERP A)
    (EQUAL (ZERO) (FIX B))
    (FALSE)).
```

By all of the foregoing, we know that if obj represents a term, then
(lcancel obj) represents a term that is provably equal to that
represented by obj.

Note that lcancel sometimes returns a term with ZERO as its
function symbol. The theorem-prover will have to spend a small amount
of time converting that term to its normal internal form, (QUOTE 0),
during the course of routine simplification. We could have defined
CANCEL to return (LIST "QUOTE" 0) instead of (LIST "ZERO"). Both terms
have the same MEANING, so the proof of correctness is no more difficult,
but the former term compiles to '(QUOTE 0), which is the internal normal
form for (ZERO). We did not define CANCEL this way only because at the
time CANCEL was first described in this paper we had not defined the
MEANING of QUOTE. The use of FALSE in lcancel can be similarly
eliminated.

IX PROOF OF THE CORRECTNESS OF CANCEL

The theorem-prover compiles every explicit value preserving function as soon as it has been admitted into the theory. During subsequent proofs, the compiled code is executed whenever constant expressions, such as (APPEND (LIST 1 2 3) (LIST 4 5 6)), arise. But the theorem-prover cannot use `lcancel` as a proof procedure until it has been proved correct.

This raises the question: how hard is it to prove the correctness of metafunctions mechanically? We can report that it was not particularly difficult to prove the correctness of CANCEL using our theorem-prover.

Recall that we have two things to prove: that CANCEL returns a FORMP when given one, and that CANCEL preserves the MEANING of input FORMPs.

Despite the complicated definition of SYMBOLP and its subfunction LEGAL.CHAR.CODE.SEQ, the proof of the FORMP property of CANCEL is almost trivial. The reason is that because CANCEL constructs no new variable symbols, SYMBOLP never becomes involved in the correctness proof: the FORMP hypothesis lets the theorem-prover establish FORMP for every subform of the output that is a subform of the input. So the only work in proving that CANCEL produces FORMPs when given FORMPs is proving that the function applications "created" by CANCEL and its subfunctions are well-formed in the sense of having a function name in the CAR and the right number of FORMPs in the CDR.

To get the theorem-prover to prove the FORMP property of CANCEL, we suggested that it prove the following easy lemmas: when given a FORMP, FRINGE returns a FORM.LSTP, the result of DELETEing something from a FORM.LSTP is a FORM.LSTP, (BAGDIFF X Y) is a FORM.LSTP when X is, and

(PLUS.TREE X) is a FORMP if X is a FORM.LSTP. The theorem-prover proves these lemmas without user assistance beyond the statement of the lemmas and the implication that they are useful. The proofs require induction -- sometimes on the structure of FORMPs, sometimes on the process of considering the elements of one bag against those of another, and sometimes on linear lists. Besides induction, the proofs require a good deal of simplification and the careful expansion of certain function definitions at the right moments. Once it has established these properties of the subfunctions of CANCEL, the system can easily employ the lemmas to prove that CANCEL produces a FORMP when given one. The entire sequence of FORMP proofs requires about 25 seconds of CPU time on a DEC KL.

The proof that CANCEL preserves the MEANING of its input and output is somewhat more interesting. Starting from the basic axioms of the theory and the definitions of the functions concerned, we first got the theorem-prover to prove some obvious facts about the theory of lists (e.g., that X is a MEMBER of (APPEND A B) iff it is a MEMBER of A or B), the theory of bags (e.g., that the bag intersection of two bags is a subbag of both), and the theory of numbers (e.g., that PLUS is associative, commutative, and allows cancellation of a common first argument on each side of an equation). Most of these classic theorems require induction to prove.

Once these facts are available, we instructed the system to prove the fundamental relationships induced by MEANING and PLUS.TREE between bags and numbers. There are three key lemmas: (a) If X is a subbag of Y, then the MEANING of the PLUS.TREE constructed from the bag difference of Y and X is equal to the Peano difference of the MEANINGS of the PLUS.TREES constructed from Y and X. (b) If X is a subbag of Y then the MEANING of the PLUS.TREE constructed from Y is a number greater than or equal to that constructed from X. (c) The MEANING of (PLUS.TREE (FRINGE X)) is the MEANING of X, when (PLUS.TREE? X) is true. The lemmas are all proved by induction -- sometimes on the structure of FORMPs and sometimes on that of bags. The first lemma is the hardest and we invite the reader to prove it as an exercise.

Once these and several similar lemmas have been proved, the fact that CANCEL preserves MEANING is fairly obvious. We will sketch the system's proof for the first branch of CANCEL. Suppose the expression to be CANCELED has the form (EQUAL u v), where u and v are PLUS-trees. By expanding the definition of MEANING, we must prove that the MEANING of the output of CANCEL is equal to:

*1 (EQUAL (MEANING u A) (MEANING v A)).

The output of CANCEL in this case is

```
(LIST "EQUAL"
      (PLUS.TREE (BAG.DIFF (FRINGE u) int))
      (PLUS.TREE (BAG.DIFF (FRINGE v) int))),
```

where int is the bag intersection of the FRINGES of u and v. The MEANING of the output is thus the equation of the MEANINGS of the two PLUS.TREE expressions:

*2 (EQUAL (MEANING (PLUS.TREE (BAG.DIFF (FRINGE u) int)) A)
 (MEANING (PLUS.TREE (BAG.DIFF (FRINGE v) int)) A)),

and we must show that *1 and *2 are equal. But the MEANING of (PLUS.TREE (BAG.DIFF Y X)) is equal to the MEANING of (PLUS.TREE Y) minus the MEANING of (PLUS.TREE X), provided X is a subbag of Y. Since int is a subbag of both (FRINGE u) and (FRINGE v) -- by the fact that the bag intersection of two bags is a subbag of both -- we can rewrite *2 to:

*3 (EQUAL (DIFFERENCE (MEANING (PLUS.TREE (FRINGE u)) A)
 (MEANING (PLUS.TREE int) A))
 (DIFFERENCE (MEANING (PLUS.TREE (FRINGE v)) A)
 (MEANING (PLUS.TREE int) A))).

Since the MEANING of (PLUS.TREE int) is less than or equal to the two minuends, and the two minuends are always numeric, lemmas from Peano arithmetic let us reduce the above equality to:

*4 (EQUAL (MEANING (PLUS.TREE (FRINGE u)) A)
 (MEANING (PLUS.TREE (FRINGE v)) A)).

But the MEANING of (PLUS.TREE (FRINGE X)) is the MEANING of X, when
(PLUS.TREE? X) is true. Thus, we can simplify *4 to:

*5 (EQUAL (MEANING u A) (MEANING v A)),

which is *1. Q.E.D.

The total CPU time required for the MEANING part of the CANCEL proofs (not counting the proofs of the list, bag, and arithmetic lemmas which are part of the system's standard repertoire) is about seven minutes. Thus, the entire CANCEL exercise consumes about eight CPU minutes plus the user's time to formulate the necessary lemmas -- a small price to pay for the assurance that the new procedure is sound.

The theorem-prover has proved the correctness of a much more difficult metafunction, namely, the totality, soundness, and completeness of a decision procedure for propositional calculus. The proof of that theorem is discussed in [1]. The theorem-prover required no modification to prove the correctness of CANCEL. In particular, the heuristics developed to prove "ordinary" theorems were just as effective when applied to "metatheorems" stated in terms of MEANING. The proof of the correctness of CANCEL involved much less user direction (in the form of lemmas) than many other mathematical results the system has proved (e.g., the prime factorization theorem derived from our shell axioms for numbers and lists). The proof is also easier than the correctness proofs for many programs (e.g., our fast string searching algorithm).

We are therefore optimistic about the prospects for adding useful new proof procedures to our theorem-prover via this approach.

X USING METAFUNCTIONS EFFICIENTLY

Whenever the user commands the theorem-prover to prove a theorem, he provides the system with a list of tokens indicating how the theorem is to be stored for future use. In [1] we employed four such tokens: REWRITE, indicating that the theorem was to be used as a rewrite rule, ELIM, indicating that it is to be used to eliminate certain "undesirable" function symbols, GENERALIZE, indicating that the theorem suggests properties to keep in mind when generalizing subgoals, and INDUCTION, indicating the theorem is useful in the search for well-founded relations and measures explaining definitions and inductions. The system checks that the theorem is suitable for use in the ways indicated (e.g., that an INDUCTION lemma really does state a property about a known well-founded relation). The purpose of the tokens is to allow the user to inform the system that the theorem should be used in the ways indicated.

We have added the new token META0, indicating that the lemma establishes that a certain function is a correct simplifier. A META0 lemma must have the form of *META in our Metatheorem. Once proved, the compiled code for the metafunction, e.g., lcancel, is stored so that it is executed on every term at the propositional level of every goal to which the simplifier is applied (i.e., the function is applied in turn to the atom of every literal in each clause simplified). Whenever the term returned is different from the input term, that occurrence of the input term is replaced by the output.

The Metatheorem justifies not only the implementation of META0 lemmas -- which let the user add new simplifiers to be applied at the propositional level -- but the implementation of what we call META1 lemmas -- which let the user add new simplifiers to be applied to every term simplified. We envision ultimately providing a variety of META

tokens corresponding to different "hooks" within the system where users need the ability to place new procedures. For each such hook the form required of the *META-lemma may be different (e.g., in some places it is sufficient to know that the MEANING of the output implies that of the input).

CANCEL is now in standard use as a META0-type proof procedure in our system. The actual definition of CANCEL in use differs slightly from the one presented in Section II. The real definition uses (LIST "QUOTE" 0) and (LIST "QUOTE" (FALSE)) instead of (LIST "ZERO") and (LIST "FALSE"). In addition, its propositional structure is slightly different so that it is more efficient: LISTP and EQUAL tests are used in place of the functions EQUALITY? and PLUS.TREE?, and the outermost IF first tests whether the argument is an equality and exits immediately when it is not, while the definition presented here tests for equality three times. Both versions of CANCEL have been proved correct and the proofs are virtually identical. The use of CANCEL as a META0-type proof procedure slows down our system by roughly one half of one percent on a sample of several hundred theorems, most of which do not involve arithmetic.

To complete this description of our work on metafunctions, we give below our theorem-prover's output on a simple theorem, concocted to illustrate CANCEL at work. The proof is produced immediately after CANCEL has been proved correct and the numerically valued functions TIMES and EXPT have been introduced. The proof involves only equality reasoning and cancellation.

Theorem.

```
(IMPLIES (AND (NUMBERP A)
              (NUMBERP X)
              (NUMBERP B)
              (EQUAL (PLUS (PLUS A B) D)
                    (PLUS B (PLUS (TIMES I J) D))))
         (EQUAL (PLUS A X)
               (PLUS B (TIMES I J))))
(EQUAL (EXPT A X) (EXPT A B)))
```

This simplifies, applying the lemma CORRECTNESS.OF.CANCEL and expanding the definition FIX, to the new conjecture:


```

(IMPLIES (AND (NUMBERP A)
              (NUMBERP X)
              (NUMBERP B)
              (EQUAL A (TIMES I J))
              (EQUAL (PLUS A X)
                    (PLUS B (TIMES I J)))))
(EQUAL (EXPT A X) (EXPT A B))),

```

which again simplifies, rewriting with CORRECTNESS.OF.CANCEL and unfolding FIX, to the conjecture:

```

(IMPLIES (AND (NUMBERP X)
              (NUMBERP B)
              (EQUAL X B))
(EQUAL (EXPT (TIMES I J) X)
      (EXPT (TIMES I J) B))),

```

which again simplifies, clearly, to:

```

(TRUE).

```

Q.E.D.

REFERENCES

1. R. S. Boyer and J S. Moore, A Computational Logic, Academic Press, New York, 1979.
2. F. M. Brown, The theory of meaning, Department of Artificial Intelligence Research Report No. 35, University of Edinburgh, 1977.
3. F. M. Brown, An investigation into the goals of research in automatic theorem proving as related to mathematical reasoning, Department of Artificial Intelligence Research Report No. 49, University of Edinburgh, 1978.
4. M. Davis and J. Schwartz, Metamathematical extensibility for theorem verifiers and proof-checkers, Courant Computer Science Report No. 12, Courant Institute of Mathematical Sciences, New York, 1977.
5. M. Gordor, R. Milner, L. Morris, M. Newey, and C. Wadsworth, A metalanguage for interactive proof in LCF, Department of Computer Science Internal Report CSR-16-77, University of Edinburgh, 1977.
6. J S. Moore, The INTERLISP virtual machine specification, CSL-76-5, Xerox Palo Alto Research Center, Palo Alto, California 1976.
7. R. W. Weyhrauch, Prolegomena to a theory of mechanized formal reasoning, (to appear in Artificial Intelligence).

INDEX

*META 58	Lemma 6 85
lcancel 103	Lemma 7 85
lf 71	Lemma 8 87
lsqm 71	Lemma 9 87
lt 71	Lemma 10 87
addl.nest 71	Lemma 11 87
APPLY 56	Lemma 12 88
arity 41, 57, 70	Lemma 13 89
axiomatic acts 45	Lemma 14 89
baddl 71	Lemma 15 90
BAGDIFF 31	Lemma 16 90
BAGINT 30	Lemma 17 91
basic axioms 45	Lemma 18 92
bminus 71	Lemma 19 94
can be proved 45	LOOKUP 44
can be proved directly from 45	MA[T] 57
CANCEL 33	MD[T] 57
constructive 46	MEANING 56
definitional extension 45	MEANING.LST 56
dequotation 50	metaaxioms 54
EQUALITY? 29	metadefinitions 56
evg 76	MINUS 43
explicit value 47	NIL 45
explicit value preserving 48	ordinary 46
extension 45	pack0 70
FORM.LSTP 57	plistp 71
FORMP 57	PLUS 20
FRINGE 29	PLUS.TREE 31
has the form 42	PLUS.TREE? 29
history 45	quotation 50
ILLEGAL.FIRST.CHAR.CODES 44, 69	quotation marks 20
INTERLISP term 75	QUOTE 45
LEGAL.CHAR.CODE.SEQ 44, 69	reducible 48
LEGAL.CHAR.CODES 44, 69	reduction 48
Lemma 1 81	s 77
Lemma 2 81	sevg 78
Lemma 3 82	shell.state 70
Lemma 4 83	standard alist 59
Lemma 5 84	symbol 41
	SYMBOLP 44, 70

term 42
term of 46
term 75
T_{1.5} 58
unpack0 69

DISTRIBUTION LIST

Office of Naval Research Information Systems Program Code 437 Arlington, VA. 22217	2 copies	Office of Naval Research Code 455 Arlington, VA. 22217	1 copy
Office of Naval Research Branch Office, Boston 495 Summer Street Boston, MASS. 02210	1 copy	Office of Naval Research Code 458 Arlington, VA. 22217	1 copy
Office of Naval Research Branch Office, Chicago 536 South Clark Street Chicago, ILL. 60605	1 copy	Naval Elec. Laboratory Center Advanced Software Tech. Div. Code 5200 San Diego, CA. 92152	1 copy
Office of Naval Research Branch Office, Pasadena 1030 East Green Street Pasadena, CA. 91106	1 copy	Mr. E. H. Gleissner Naval Ship Res. & Dev. Center Computation & Math. Dept. Bethesda, MD. 20084	1 copy
New York Area Office 715 Broadway - 5th Floor New York, N.Y. 10003	1 copy	Captain Grace M. Hopper NAICOM/MIS Planning Branch (OP-916D) Office of Chief of Naval Operations Washington, D.C. 20350	1 copy
Assistant Chief for Technology Office of Naval Research Code 200 Arlington, VA. 22217	1 copy	Officer-in-Charge Naval Surface Weapons Center Dahlgren Laboratory Dahlgren, VA. 22448 Attn: Code KP	1 copy
Naval Research Laboratory Technical Information Div., Code 2627 Washington, D.C. 20375	6 copies		
Dr. A. L. Slafkosky Scientific Advisor Commandant of the Marine Corps (Code RD-1) Washington, D.C. 20380	1 copy		

DATE
FILMED
- 8