.

The Addition of Bounded Quantification and Partial Functions to A Computational Logic and Its Theorem Prover

Robert S. Boyer and J Strother Moore

ICSCA-CMP-52 January, 1987

The research reported here was supported by the Venture Research Unit of British Petroleum, Ltd., London, National Science Foundation Grant MCS-8202943, and Office of Naval Research Contract N00014-81-K-0634.

Institute for Computing Science and Computer Applications
The University of Texas at Austin
Austin, Texas 78712

Abstract

We describe an extension to our quantifier-free computational logic to provide the expressive power and convenience of bounded quantifiers and partial functions. By *quantifier* we mean a formal construct which introduces a *bound* or *indicial* variable whose *scope* is some subexpression of the quantifier expression. A familiar quantifier is the Σ operator which sums the values of an expression over some range of values on the bound variable. Our method is to represent expressions of the logic as objects in the logic, to define an interpreter for such expressions as a function in the logic, and then define quantifiers as "mapping functions." The novelty of our approach lies in the formalization of the interpreter and its interaction with the underlying logic. Our method has several advantages over other formal systems that provide quantifiers and partial functions in a logical setting. The most important advantage is that proofs not involving quantification or partial recursive functions are not complicated by such notions as "capturing," "bottom," or "continuity." Naturally enough, our formalization of the partial functions is nonconstructive. The theorem prover for the logic has been modified to support these new features. We describe the modifications. The system has proved many theorems that could not previously be stated in our logic. Among them are:

- classic quantifier manipulation theorems, such as

$$\sum_{i=0}^{n} g(i)+h(i) \;=\; \sum_{i=0}^{n} g(i) \;+\; \sum_{i=0}^{n} h(i);$$

- elementary theorems involving quantifiers, such as the Binomial Theorem:

$$(a+b)^n \;=\; \sum_{i=0}^{n} \binom{n}{i} a^i b^{n-i};$$

- termination properties of many partial recursive functions such as the fact that an application of the partial function described by

```
(LEN X)
   @val(ARROW)
(IF (EQUAL X NIL)
    0
    (ADD1 (LEN (CDR X))))
```

  terminates if and only if the argument ends in `NIL`;

- theorems about functions satisfying unusual recurrence equations such as the 91-function and the following list reverse function:

```
(RV X)
   @val(ARROW)
(IF (AND (LISTP X) (LISTP (CDR X)))
    (CONS (CAR (RV (CDR X)))
          (RV (CONS (CAR X)
                    (RV (CDR (RV (CDR X)))))))
    X).
```

1. Introduction

The research reported here was motivated by a desire to add the expressive power of quantifiers to the logic mechanized by the theorem prover described in [acl, meta, linear]. By *quantifier* we mean a formal construct which introduces a *bound* or *indicial* variable whose *scope* is limited to some given subformula or subterm, which we here call the *body*, of the quantified expression.

Everyday mathematics makes frequent use of a wide variety of quantifiers. Among them are:

$$\sum_{v \in S} f(v)$$

$$\prod_{v \in S} f(v)$$

$$\{v \mid p(v)\}$$

$$\exists_{v \in S} f(v)$$

Such constructs are also common in computer programming: all high level programming languages provide convenient "iterative forms:" the FORTRAN DO-loop, the Pascal *for*-statement, the Interlisp FOR statement, the Maclisp LOOP statement. As evidence for the power and convenience of such iterative forms, in the code for our theorem prover we use the LOOP construct more often than the procedure definition construct.

We limit our attention to bounded quantification, that is, quantification where the bound variable ranges over some finite sequence. It turns out that the formalization we employ also gives us partial recursive functions, though we do not focus on that aspect of the solution yet.

The logic to which we will add quantifiers is described in detail in [ACL] and amended slightly in [META]. The theorem prover for that logic is described in [ACL, META, LINEAR]. In section BACKGROUND we outline the necessary background information, where we also give an annotated list of references to representative applications. Section BACKGROUND can be summarized as follows: The logic is a quantifier-free first-order logic providing for the definition of total recursive functions on finite, inductively constructed data objects such as numbers, atomic symbols, and ordered pairs. The logic resembles Pure Lisp. The theorem prover is a large and complicated computer program that can discover some proofs in the logic. The theorem prover consists of 570,000 bytes of source code and is the product 15 years work by the two authors. Under the guidance of human users the theorem prover has checked proofs of such theorems as Gauss' law of quadratic reciprocity and G@um(o)del's incompleteness theorem. The logic has proved useful in formalizing the properties of computer programs, algorithms, and systems. The theorem prover has proved many algorithms and computer programs correct, has verified the correctness of digital hardware designs, and has been used to investigate various properties of system designs.

Logic is a vehicle. Ours is a truck. It is not particularly small or elegant, but it is capable of hauling a lot of cargo. A major constraint in our formalization of quantifiers was to take advantage of the existing logic and theorem prover. Quantifiers and partial functions are, in essence, just mechanisms for controlling the application of the elementary operators of the underlying theories (e.g., arithmetic, list processing, etc.). To build a formal system or mechanical theorem prover for the manipulation of quantifiers or partial functions without extensive support for the elementary

theories is akin to building a vehicle with no room for passengers or cargo. Indeed, we believe it is a mistake to let the provisions for the sophisticated features impair the simplicity of the elementary theories they are designed to support. Even in theorems involving quantifiers, the vast majority of the logical manipulations are concerned with the quantifier-free expressions inside the quantifier bodies: precisely the kind of expressions our existing theorem prover was designed to manipulate.

In this introduction we motivate our introduction of quantifiers, briefly sketch the formalization used, and illustrate the new theory by exhibiting both schematic and concrete theorems about quantifiers and partial recursive functions. In succeeding sections we give background information, present the formal details of our new theory, construct several proofs in it, briefly sketch the changes made in the existing theorem prover to support the quantified theory, and show many beautiful results proved by the system.

Suppose the user of the logic desires to discuss the list obtained by doubling the elements of L. The expression (DOUBLE-LIST L) denotes such a list, if the user has defined DOUBLE-LIST as

```
Definition.
(DOUBLE-LIST L)
   =
(IF (NLISTP L)
    NIL
    (CONS (TIMES 2 (CAR L))
          (DOUBLE-LIST (CDR L)))).
```

For example, (DOUBLE-LIST '(1 2 3 4)) = '(2 4 6 8).

A useful theorem about DOUBLE-LIST is that it "distributes" over the list concatenation function APPEND:

```
Theorem.
(DOUBLE-LIST (APPEND A B))
   =
(APPEND (DOUBLE-LIST A)
        (DOUBLE-LIST B)).
```

However, suppose that in addition the user wished to refer to the list obtained by adding 1 to every element of L. To do so he would have to define the function ADD1-LIST, so that (ADD1-LIST '(1 2 3 4)) = '(2 3 4 5). Should it be necessary to use the fact that ADD1-LIST distributes over APPEND, that fact would have to be proved. One is tempted to say "proved again." But since DOUBLE-LIST and ADD1-LIST are different function symbols, the first lemma cannot be used to shorten the proof of the second.

In the quantified version of our logic we permit the user to write such expressions as:

```
(for X in L collect (TIMES 2 X))
```

and

```
(for X in L collect (ADD1 X)).
```

The first expression is equivalent to (DOUBLE-LIST L) and the second is equivalent to

`(ADD1-LIST L)`.

It is possible to state very general lemmas about quantifiers. Consider for example the schematic form:

```
(for V in (APPEND A B) collect body(V))
   =
(APPEND (for V in A collect body(V))
        (for V in B collect body(V))),
```

where *body* is understood here to be a second order variable. (Our formalization does not introduce such variables or other new syntactic classes. We adopt this notation now only for expository purposes.) This lemma is easy to prove because no time is wasted considering special properties of the particular *body* used. This lemma is more useful than the fact that `DOUBLE-LIST` distributes over `APPEND` since it also handles the analogous property for `ADD1-LIST` and other such functions.

The introduction of quantifiers has three major attractions: First, quantifiers are conceptually clarifying. Logical relationships that would otherwise be buried inside of recursive definitions are lifted out into the open. The users and readers of the logic need no longer invent and remember names for many "artificial" recursive functions. Second, it is possible to state general purpose, schematic theorems about quantifiers, thus reducing the number of theorems needed. Third, these schematic theorems are generally easier to prove than the corresponding theorems about recursively defined instances of the quantifiers because irrelevant concepts have been abstracted out of the theorems. Thus, quantifiers are a boon to the reader, to the writer, and to the theorem prover.

The quantifiers most commonly found in formal systems are the universal and existential quantifiers, @Z(8) and @Z(9), of predicate calculus and the λ-expression of lambda-calculus. In [Morse] Morse defines a formal system in which new quantifiers can be introduced. Such classical formal treatments of quantifiers include the new class of "bound" or "indicial" variables, "quantifier" symbols for which some "argument" slots are known to contain bound variables and others to contain terms in which the variables may or may not be considered bound by the quantifier, and higher-order "schematic" variables to permit the statement of general lemmas. For example, in the expression:

$$\sum_{v \in S(u,v)} f(v)$$

the "Σ" is a "quantifier" symbol with three argument slots. The first, here occupied by the first occurrence of "v," is an "indicial variable" slot, indicating that "v" is the variable bound by this quantifier. The second, here occupied by the term "s(u,v)," is "normal" in the sense that occurrences of "v" in this term are not bound by this quantifier. The third, here occupied by the term "f(v)," is the "body" of the quantifier and occurrences of "v" here are bound. The rule of instantiation is changed so that substitution only replaces "free" variables and has additional restrictions to avoid "capturing." New rules of inference are added to permit renaming of bound variables and schematic instantiation.

We find such elaborations of elementary logic unattractive for three reasons. First, it is surprisingly easy to get the rules wrong and produce an inconsistent logic. For example, Alonzo Church writes "Especially difficult is the matter of a correct statement of the rule of substitution for

functional variables," and he observes that Hilbert, Ackermann, Carnap, and Quine all published incorrect statements of the rule ([CHURCH-LOGIC], pp. 289). A. P. Morse (private communication) has observed that Kelley incorrectly states the rule for instantiating the axiom of comprehension in [kelley]. Second, the elaborations complicate the proofs of theorems that can be stated and proved in far simpler systems. Finally, the impact of such elaborations on an existing theorem prover for a simpler logic are enormous. In our system, code for choosing proof tactics is often mingled with code for carrying out logical transformations justified by rules of inference. Such elaborations as enumerated above require modifying virtually every program in the system. For example, many (if not most) of our programs "know" the structure of terms and would have to be reprogrammed to accommodate bound variable slots. Many of our programs explore "virtual" terms that are specified as instantiations of a given term under a given substitution. Such programs carry out the substitution "on the fly" and thus would be affected by a change in the rule of instantiation.

We have discovered a device for obtaining the power of bounded quantification while avoiding many of these problems. Its great advantage is that to introduce quantifiers we did not have to change either the syntax or the rules of inference of the logic. Instead, we introduced FOR as a new function symbol and added some new axioms to the logic to define it. The impact of this approach is tremendous: the old version of the theorem prover is sound for the quantified version of the logic and proofs of quantifier-free formulas are not complicated by the provisions for quantifiers.

Our syntax for quantified expressions is borrowed from Teitleman's FOR operator in Interlisp [INTERLISP]. The expression:

```
(for X in (APPEND U V)
      when (MEMBER X B)
      collect (TIMES X C))
```

is formally represented in the logic and in the implementation by:

```
(FOR 'X (APPEND U V)
        '(MEMBER X B)
        'COLLECT '(TIMES X C)
        (LIST (CONS 'B B) (CONS 'C C))).
```

FOR is a function of six arguments. In common usage the first, third, fourth and fifth are explicit constants supplying the *bound variable symbol*, an s-expression to be treated as a *conditional* expression, an *operator* indicating what should be done on each iteration, and an s-expression to be treated as the *body*. The second argument is the range of values the bound variable is to take on and the last argument is an association list mapping the free variables in the conditional and body s-expressions to their values. FOR maps over the range, binding the bound variable successively to the elements of the range, and performs the indicated operation on the result of evaluating the body whenever the conditional expression evaluates to true.

The formal definition of FOR is:

Definition.
```
(FOR V L COND OP BODY A)
   =
(IF (NLISTP L)
    (QUANTIFIER-INITIAL-VALUE OP)
    (IF (EVAL COND (CONS (CONS V (CAR L)) A))
        (QUANTIFIER-OPERATION OP
           (EVAL BODY (CONS (CONS V (CAR L)) A))
           (FOR V (CDR L) COND OP BODY A))
        (FOR V (CDR L) COND OP BODY A))).
```

`(QUANTIFIER-INITIAL-VALUE OP)` defines the value of each quantifier operation on the empty range. For example, `(QUANTIFIER-INITIAL-VALUE 'SUM)` is `0` and `(QUANTIFIER-INITIAL-VALUE 'ALWAYS)` is `T`. `(QUANTIFIER-OPERATION OP X Y)` performs the indicated operation at each iteration. For example, `(QUANTIFIER-OPERATION 'SUM X Y)` is `(PLUS X Y)` and `(QUANTIFIER-OPERATION 'ALWAYS X Y)` is `(AND X Y)`.

This approach to quantification is well known in the Lisp community. So what is new? It is our formal treatment of the key concept underlying the above definition of `FOR`, namely `EVAL`. Formalizing an interpreter for a language within the language is a subtle task that easily leads to formal theories that are either too powerful (i.e., inconsistent) or too weak.

`EVAL` is a function that takes an s-expression and an association list (or *alist*) pairing quoted variable symbols to values and returns the value of the given s-expression under the assignments in the alist.[1] For example, we wish to have the theorem that:

`(EVAL '(TIMES X Y)` σ`)` = `(TIMES X Y)`

provided σ assigns `'X` the value `X` and `'Y` the value `Y`. However, one must be careful to avoid introducing inconsistency by making `EVAL` "too powerful." For example, if we have, for every n-ary function symbol `fn`:

`(EVAL '(fn` x_1 **`...`** x_n**`) A)`**
`   =`
`(fn (EVAL '`x_1**` A) ... (EVAL '`**x_n**` A))`**

then a contradiction arises if it is possible to define

`(RUSSELL X)` = `(NOT (EVAL X NIL))`

and

`(CONST)` = `'(RUSSELL (CONST)).`

For then

```
   (RUSSELL (CONST))
= (NOT (EVAL (CONST) NIL))
= (NOT (EVAL '(RUSSELL (CONST)) NIL))
= (NOT (RUSSELL (EVAL '(CONST) NIL)))
= (NOT (RUSSELL (CONST))).
```

[1] In our formalization, `EVAL` takes an additional flag argument which we have here suppressed. Strictly speaking `(EVAL x va)` is an abbreviation for `(EVAL T x va)`.

But neither the definition of RUSSELL nor of CONST is, by itself, "bad." RUSSELL is just an abbreviation for a simple expression involving previously introduced functions. CONST is defined to be a list constant.

Brutal solutions to this problem, such as disallowing the use of EVAL in definitions or preventing EVAL from being able to evaluate 'EVAL expressions, prevent the conventional use of quantifiers. The former suggestion disallows the definition of FOR; the latter prevents nested FORs since the outer FOR would be evaluating the inner one with EVAL and the inner one involves EVAL.

Such paradoxes arise in naive formulations of higher order logic, set theory, and the semantics of lambda calculus. A mechanism must be introduced to prevent these formal systems from being "too powerful." This mechanism is frequently quite artificial (e.g., hierarchies of types, subscripts on variable and function symbols, notions such as continuity and unusual objects such as @val(BTM) etc.) and so clumsy that its presence is frequently ignored in simple theorems (where it would otherwise dominate the proof process) or is suppressed by informal syntactic conventions. Of course, in the formal logic -- as implemented in our machine -- the mechanism cannot be suppressed and its clumsiness is reflected both in the code and in the proofs produced.

We solve this problem by axiomatizing within the logic a nonconstructive function, here called V&C, which takes a quoted expression, an alist, and (implicitly) a set of recurrence equations constituting the definitions of all the non-primitive functions, and either returns a pair <v,c> specifying the value and cost in function calls of evaluating the expression or F if no cost is sufficient. In other words, V&C solves the halting problem for partial recursive functions. While no such recursive function exists, the axiom characterizing V&C is satisfied by one and only one function.

The presence of V&C within the logic permits us to address ourselves formally to termination questions directly. For example, using V&C we can introduce a recurrence equation which does not always terminate -- a condition that prevents the equation's admission as a recursive definition under the logic's principle of definition -- and explore the logical properties of the equation, including its termination properties.

Unlike the Scott-Strachey approach [STOY, Gordon79], our system does not actually provide *functions* as objects. We merely provide *descriptions of* or *recipes for computing* functions as objects. Nevertheless, we can describe any partial recursive function and address ourselves to the proof of its properties. Furthermore, the rules for proving properties of (descriptions of) functions are very similar to those used in the Scott-Strachey system. Within the framework of a formal, logical system, it matters little in practice whether the formulas are thought of as talking about functions or their descriptions (if the underlying rules are the same): proofs are syntactic.

We discuss our termination proofs after we have introduced V&C formally. We now continue with our discussion of quantifiers.

How, in this formalism, do we state theorems about quantifiers, e.g., theorems apparently requiring the use of second order variables? Consider the familiar theorem:

```
(for I in L sum (PLUS g(I) h(I)))
   =
(PLUS (for I in L sum g(I))
      (for I in L sum h(I))),
```

where *g* and *h* are second order. The statement of this theorem in our formulation is:

```
Theorem.  SUM-DISTRIBUTES-OVER-PLUS:
(FOR I L COND 'SUM (LIST 'PLUS G H) A)
   =
(PLUS (FOR I L COND 'SUM G A)
      (FOR I L COND 'SUM H A)).
```

We prove SUM-DISTRIBUTES-OVER-PLUS below. We assume the following fact about EVAL:

```
Theorem.  EVAL-DISTRIBUTES-OVER-PLUS:
(EVAL (LIST 'PLUS X Y) A)
   =
(PLUS (EVAL X A) (EVAL Y A)).
```

Proof. The proof is by induction on L. In the base case, where L is empty, both sides reduce to 0, by expanding the definition of FOR and (QUANTIFIER-INITIAL-VALUE 'SUM).

In the induction step, assume the theorem holds for L and prove that it holds for (CONS K L). Let σ be (CONS (CONS I K) A). The induction step breaks into two cases according to whether the condition COND evaluates to F. The two cases are similar and we show only the case where the condition is non-F. We will transform the left-hand side of our induction conclusion into the right-hand side:

```
(FOR I (CONS K L) COND 'SUM (LIST 'PLUS G H) A)
     =                                          [1]
(PLUS (EVAL (LIST 'PLUS G H) σ)
      (FOR I L COND 'SUM (LIST 'PLUS G H) A))
     =                                          [2]
(PLUS (PLUS (EVAL G σ) (EVAL H σ))
      (FOR I L COND 'SUM (LIST 'PLUS G H) A))
     =                                          [3]
(PLUS (PLUS (EVAL G σ) (EVAL H σ))
      (PLUS (FOR I L COND 'SUM G A)
            (FOR I L COND 'SUM H A)))
     =                                          [4]
(PLUS (PLUS (EVAL G σ)
            (FOR I L COND 'SUM G A))
      (PLUS (EVAL H σ)
            (FOR I L COND 'SUM H A)))
     =                                          [5]
(PLUS (FOR I (CONS K L) COND 'SUM G A)
      (FOR I (CONS K L) COND 'SUM H A)).
```

Step 1 is the expansion of the definitions of FOR and QUANTIFIER-OPERATION. Step 2 is by EVAL-DISTRIBUTES-OVER-PLUS. Step 3 is the use of the induction hypothesis. Step 4 is simple arithmetic and step 5 is by the definitions of FOR and QUANTIFIER-OPERATION again. **Q.E.D.**

The proof just given can be constructed automatically by the unmodified version of our theorem

prover, given the definition of FOR and the assumed property of EVAL. Furthermore, once this theorem has been proved as a rewrite rule, it can be instantiated and used by a standard term rewrite system. For example, consider the term

```
(for K in (FROM-TO 0 N)
       when (PRIME K)
       sum (PLUS K (SQ K))).
```

This term is an abbreviation for:

```
(FOR 'K (FROM-TO 0 N)
     '(PRIME K)
     'SUM '(PLUS K (SQ K))
     NIL).
```

Call this the "target" term. The left-hand side of the theorem SUM-DISTRIBUTES-OVER-PLUS,

```
(FOR I L COND 'SUM (LIST 'PLUS G H) A)
```

matches the target term under the instantiation that replaces I with 'K, L with (FROM-TO 0 N), COND with '(PRIME K), G with 'K, H with '(SQ K), and A with NIL. Thus we can replace the target with the instantiation of the right-hand side of SUM-DISTRIBUTES-OVER-PLUS, which may be abbreviated:

```
(PLUS (for K in (FROM-TO 0 N)
           when (PRIME K)
           sum K)
      (for K in (FROM-TO 0 N)
           when (PRIME K)
           sum (SQ K))).
```

Thus, our formalization of FOR permits the statement, proof, and use of "schematic" lemmas without any change in the logic.

Recall the primary advantage of introducing FOR as an ordinary function symbol: the existing theorem prover is sound for the quantified logic. While the existing theorem prover may not be very "smart" about quantified expressions, its proofs of unquantified theorems remain unchanged and its manipulations of quantified formulas are correct. To our surprise, the system can prove a wide variety of theorems about quantifiers and it has successfully applied those theorems in other proofs.

We have added several new proof techniques explicitly for dealing with FOR, EVAL, and V&C. We describe them briefly in this paper and illustrate several mechanical proofs, including a proof of the Binomial Theorem and some termination proofs.

2. Background: The Unquantified Logic, Its Theorem Prover and Capabilities

Readers already familiar with our logic and theorem prover can skip this section.

We here describe the unquantified logic and its theorem prover. In so doing we familiarize the reader with the working logic. We also put into perspective one of the major constraints on our quantifier work: the new theory and theorem prover should be as rugged and practicable as the old.

In [acl, meta, linear] we describe a quantifier free first-order logic and a large and complicated computer program that proves theorems in that logic. The major application of the logic and theorem prover is the formal verification of properties of computer programs, algorithms, system designs, etc. In this section we describe the logic and the theorem prover briefly and we list some of the major applications.

2.1. The Unquantified Logic

A complete and precise definition of the logic can be found in Chapter III of [ACL] together with the minor revisions detailed in section 3.1 of [META].

We use the prefix syntax of Pure Lisp [McCarthy65] to write down terms. For example, we write `(PLUS I J)` where others might write `PLUS(I,J)` or `I+J`.

The logic is first-order, quantifier free, and constructive. It is formally defined as an extension of propositional calculus with variables, function symbols, and the equality relation. We add axioms defining the following:

- the Boolean objects `(TRUE)` and `(FALSE)`, abbreviated `T` and `F`;
- The if-then-else function, `IF`, with the property that `(IF x y z)` is `z` if `x` is `F` and `y` otherwise;
- the Boolean "connector functions" `AND`, `OR`, `NOT`, and `IMPLIES`; for example, `(NOT p)` is `T` if `p` is `F` and `F` otherwise;
- the equality function `EQUAL`, with the property that `(EQUAL x y)` is `T` or `F` according to whether `x` is `y`;
- inductively constructed objects, including:
    - Natural Numbers. Natural numbers are built from the constant `(ZERO)` by successive applications of the constructor function `ADD1`. The function `NUMBERP` recognizes natural numbers, e.g., is `T` or `F` according to whether its argument is a natural number or not. The function `SUB1` returns the predecessor of a non-`0` natural number.
    - Ordered Pairs. Given two arbitrary objects, the function `CONS` returns an ordered pair containing them. The function `LISTP` recognizes such pairs. The functions `CAR` and `CDR` return the two components of such a pair.
    - Literal Atoms. Given an arbitrary object, the function `PACK` constructs an atomic symbol with the given object as its "print name." `LITATOM` recognizes such objects and `UNPACK` returns the print name.
- We call each of the classes above a "shell." `T` and `F` are each considered the elements of two singleton shells. Axioms insure that all shell classes are disjoint;
- the definitions of several useful functions, including:
    - `LESSP` which, when applied to two natural numbers, returns `T` or `F` according to whether the first is smaller than the second;
    - `LEX2`, which, when applied to two pairs of naturals, returns `T` or `F` according as whether the first is lexicographically smaller than the second; and
    - `COUNT` which, when applied to an inductively constructed object, returns its "size;" for example, the `COUNT` of an ordered pair is one greater than the sum of the `COUNT`s of the components.

The logic provides a principle under which the user can extend it by the addition of new shells. By instantiating a set of axiom schemas the user can obtain a set of axioms describing a new class of inductively constructed n-tuples with type-restrictions on each component. For each shell there is a recognizer (e.g., LISTP for the ordered pair shell), a constructor (e.g., CONS), an optional empty object (e.g., there is none for the ordered pairs but (ZERO) is the empty natural number), and n accessors (e.g., CAR and CDR).

The logic provides a principle of recursive definition under which new function symbols may be introduced. Consider the definition of the list concatenation function:

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

The equations submitted as definitions are accepted as new axioms under certain conditions that guarantee that one and only one function satisfies the equation. One of the conditions is that certain derived formulas be theorems. Intuitively, these formulas insure that the recursion "terminates" by exhibiting a "measure" of the arguments that decreases, in a well-founded sense, in each recursion. A suitable derived formula for APPEND is:

```
(IMPLIES (LISTP X)
         (LESSP (COUNT (CDR X))
                (COUNT X))).
```

However, in general the user of the logic is permitted to choose an arbitrary measure function (COUNT was chosen above) and one of several relations (LESSP above).

The rules of inference of the logic, in addition to those of propositional calculus and equality, include mathematical induction. The formulation of the induction principle is similar to that of the definitional principle. To justify an induction schema it is necessary to prove certain theorems that establish that, under a given measure, the inductive hypotheses are about "smaller" objects than the conclusion.

Using induction it is possible to prove such theorems as the associativity of APPEND:

```
Theorem.
(EQUAL (APPEND (APPEND A B) C)
       (APPEND A (APPEND B C))).
```

2.2. The Mechanization of the Unquantified Logic

The theorem prover for the unquantified logic, as it stood in 1979, is described completely in [acl]. Many improvements have been added since. In [Meta] we describe a "metafunction" facility which permits the user to define new proof procedures in the logic, prove them correct mechanically, and have them used efficiently in subsequent proof attempts. During the period 1980-1985 a linear arithmetic decision procedure was integrated into the rule-driven simplifier. The problems of integrating a decision procedure into a heuristic theorem prover for a richer theory are discussed in [Linear]. The theorem prover is briefly sketched here.

The theorem prover is a computer program that takes as input a term in the logic and repeatedly transforms it in an effort to reduce it to non-F. The theorem prover employs eight basic transformations:

- decision procedures for propositional calculus, equality, and linear arithmetic;
- term rewriting based on axioms, definitions and previously proved lemmas;
- application of verified user-supplied simplifiers called "metafunctions;"
- renaming of variables to eliminate "destructive" functions in favor of "constructive" ones;
- heuristic use of equality hypotheses;
- generalization by the replacement of terms by type-restricted variables;
- elimination of apparently irrelevant hypotheses; and
- mathematical induction.

The theorem prover contains many heuristics to control the orchestration of these basic techniques.

In a shallow sense, the theorem prover is fully automatic: the system accepts no advice or directives from the user once a proof attempt has started. The only way the user can alter the behavior of the system during a proof attempt is to abort the proof attempt. However, in a deeper sense, the theorem prover is interactive: the system's behavior is influenced by the data base of lemmas which have already been formulated by the user and proved by the system. Each conjecture, once proved, is converted into one or more "rules" which guide the theorem prover's actions in subsequent proof attempts.

A data base is thus more than a logical theory: it is a set of rules for proving theorems in the given theory. The user leads the theorem prover to "difficult" proofs by "programming" its rule base. Given a goal theorem, the user generally discovers a proof himself, identifies the key steps in the proof, and then formulates them as lemmas, paying particular attention to their interpretation as rules.

The key role of the user in our system is guiding the theorem prover to proofs by the strategic selection of the sequence of theorems to prove and the proper formulation of those theorems. Successful users of the system must know how to prove theorems in the logic and must understand how the theorem prover interprets them as rules.

2.3. Capabilities of the Unquantified Theorem Prover

What can be formalized in the unquantified logic? What can be proved by the unquantified version of the theorem prover? We indicate answers to these questions by discussing some of the applications of that theorem prover.

Below is an annotated list of selected references to theorems proved by the system. The reader should understand, in view of the foregoing remarks on the role of the user, that -- for the deep theorems at least -- the theorem prover did not so much *discover* proofs as *check* proofs sketched by the user.

- **Elementary List Processing**: Many elementary theorems about list processing are

discussed among the examples in [ACL]. The appendix includes theorems proved about such concepts as concatenation, membership, permuting (including reversing and sorting) and tree exploration.

- **Elementary Number Theory**: Euclid's Theorem and the existence and uniqueness of prime factorizations are proved in [ACL]. A version of the pigeon hole principle and Fermat's theorem are proved in [RSA]. Wilson's Theorem is proved in [Russinoff]. Finally, Gauss' Law of Quadratic Reciprocity has been checked; the theorem, its definitions, and the lemmas suggested by Russinoff are included among the examples in the standard distribution of the theorem-proving system.

- **Metamathematics**: The soundness and completeness of a decision procedure for propositional calculus, similar to the Wang algorithm, is proved in [ACL]. The soundness of an arithmetic simplifier for the logic is proved in [Meta]. The Turing completeness of Pure LISP is proved in [TMI]. The recursive unsolvability of the halting problem for Pure LISP is proved in [Unsolv]. The Tautology Theorem, i.e., that every tautology has a proof in Shoenfield's formal system, is proved in [Shankar-85a]. The Church-Rosser theorem is proved in [Shankar-85b]. G@um (o)del's incompleteness theorem is proved in [Shankar-86].

- **Communications Protocols**: Safety properties of two transport protocols, the Stenning protocol and the "NanoTCP" protocol, are proved in [DiVito].

- **Concurrent Algorithms**: A mechanized theory of "simple" sorting networks and a proof of the equivalence of sequential and parallel executions of an insertion sort program are described in [Len85]. A more general treatment of sorting networks and an equivalence proof for a bitonic sort are given in [HuLe84]. A proof of the optimality of a given transformation for introducing concurrency into sorting networks is described in [LeHu85b].

- **Fortran Programs**: A verification condition generator for a subset of ANSI Fortran 66 and 77 is presented in [VCG]. The same paper describes the correctness proof of a Fortran implementation of the Boyer-Moore fast string searching algorithm. A correctness proof for a Fortran implementation of an integer square root algorithm based on Newton's method is described in [isqrt]. The proof of a linear time majority vote algorithm in Fortran is given in [mjrty].

- **Real Time Control**: A simple real time control problem is considered in [controller]. The paper presents a recursive definition of a "simulator" for a simple physical system -- a vehicle attempting to navigate a straightline course in a varying cross-wind. Two theorems are proved about the simulated vehicle: the vehicle does not wander outside of a certain corridor if the wind changes "smoothly" and the vehicle homes to the proper course if the wind stays steady for a certain amount of time.

- **Assembly Language**: A simple assembly language for a stack machine is formalized in [acl]. The book also gives a correctness proof for a function that compiles expressions into that assembly language. In our standard benchmark of definitions and theorems is a collection that defines another simple assembly language, including "jump" and "move to memory" instructions, and proves the correctness of a program that iteratively computes the sum of the integers from 0 to n. The correctness proof is complicated by the fact that the instructions are fetched from the same memory being modified by the execution of the program. The list of events is included in the standard distribution of our theorem-proving system.

- **Hardware Verification**: The correctness of a ripple carry adder is given in [Hunt85b]. The adder is a recursively defined function which maps a pair of bit vectors and an input carry bit to a bit vector and an output carry bit. The theorem establishes that the natural number interpretation of the output is the Peano sum of the natural number interpretations of the inputs, with appropriate consideration of the carry flags. An analogous result is proved for twos-complement integer arithmetic.

The recursive description of the circuit can be used to generate an adder of arbitrary width. A 16-bit wide version is shown. Propagate-generate and conditional-sum adders have also been proved correct. Also in [Hunt85b] is the correctness proof of the combinational logic for a 16-bit wide arithmetic logical unit providing the standard operations on bit vectors, natural numbers, and integers. The dissertation then presents a recursively described microcoded cpu, called the FM8501, comparable in complexity to a PDP-11 and proves that the device correctly implements an instruction set defined by a high-level interpreter.

3. The Formal Definition of V&C

In order to add the power of quantifiers and partial functions we extend the logic of [ACL] by (a) adopting the abbreviation conventions of [meta], (b) adding definitions of several recursive functions, (c) adding several unproblematic axioms defining functions that map between syntactic objects (e.g., function symbols) and objects in the logic (e.g., `LITATOM` constants), (d) adding an axiom describing the uncomputable function `V&C`, and then (e) defining several useful functions on top of `V&C`. In this section we take steps (a)-(d).

Technically, the extended logic remains first-order (though we have some of the power of second order variables and functional objects) and quantifier free (though we have some of the power of quantifiers). However, the extended logic is non-constructive.

3.1. Explicit Values, Abbreviations and Quotations

We start with the logic described in Chapter III of [acl].[2] We add to that logic a new convention for writing down certain constants. The convention is, essentially, the "quote" or "dot" notation of Lisp.

In the "formal syntax" of the logic every term is either a variable symbol or is the application of an n-ary function symbol `fn` to n other terms, t_1, ..., t_n, written (`fn` t_1 `...` t_n).

The "extended syntax" of the logic provides succinct abbreviations for certain constants (i.e., variable-free terms), namely those composed entirely of shell constructors and empty shell objects. We call such terms *explicit value terms*. See [Meta] for details.

Here are some examples of explicit value terms displayed in the formal syntax:

```
(ADD1 (ADD1 (ZERO)))

(CONS (TRUE) (CONS (FALSE) (ZERO)))

(PACK (CONS (ADD1 (ZERO)) (ZERO)))
```

Explicit value terms play a key role in our formalization of quantifiers and partial functions because they are used to encode the terms of the logic as objects in the logic. Before discussing the "quotation" of terms we illustrate the notation in which explicit value terms are usually

[2] Actually, we have revised the logic in three ways that are unimportant to the thrust of the current work. (1) We have abandoned the old convention for writing down explicit `LITATOM`s and adopt a new one described here. (2) `NIL` is no longer the bottom object of the `LITATOM` shell -- the shell has no bottom object now. (3) The default value returned when `CAR` or `CDR` is applied to a non-`LISTP` is `0`, now, instead of `NIL`. These changes to the theory were described in [meta].

displayed.

Nests of ADD1's around (ZERO) are abbreviated by natural numbers in decimal notation. Thus, the term (ADD1 (ADD1 (ZERO))) may be abbreviated 2.

Literal atoms constructed from certain lists of ASCII character codes are abbreviated by quoted symbols. For example, the term (PACK (CONS 65 (CONS 66 (CONS 67 0)))) may be abbreviated 'ABC. (The ASCII codes for the letters "A", "B" and "C" are, respectively, 65, 66, and 67.) The term 'NIL is further abbreviated NIL.

Terms of the form (CONS t_1 (CONS t_2 ... (CONS t_n NIL)...)) may be abbreviated (LIST t_1 t_2 ... t_n).

Finally, lists of explicit values may be abbreviated with the "dot notation" of Pure Lisp. For example, the explicit value term:

```
(LIST (CONS 'X 7)
      (CONS 'Y 8)
      (CONS 'Z 9))
```

may be abbreviated '((X . 7) (Y . 8) (Z . 9)). The explicit term

```
(LIST 'PLUS
      (LIST 'ADD1 'X)
      'Y)
```

may be abbreviated '(PLUS (ADD1 X) Y).

The "quote notation" just illustrated has a remarkable relationship to the formal syntax of the logic itself.

The formation rules for the functions and variable symbols of our language are such that to each symbol, sym, there corresponds a LITATOM which we write 'sym. For example, X is a variable symbol and 'X or (PACK (CONS 88 0)) is a LITATOM. ADD1 is a function symbol and 'ADD1 is a LITATOM.

To each term, t, there corresponds an explicit value term, 't, which may be written down by writing down t in the formal syntax and preceding it with a single quote mark. For example, (ADD1 X) is a term in the logic, and '(ADD1 X) is an explicit value term, namely (CONS 'ADD1 (CONS 'X NIL)), or, using fewer abbreviations:

```
(CONS (PACK (CONS 65
                  (CONS 68
                    (CONS 68 (CONS 49 0)))))
      (CONS (PACK (CONS 88 0))
            (PACK (CONS 78
                        (CONS 73 (CONS 76 0)))))).
```

We call 't a "quotation" of t. The quotation of a term is an explicit value term that "represents" the given term. We define the notion precisely in [meta]. It turns out that terms have many quotations. For example, each of the explicit value terms below is a quotation of (PLUS 1 X):

```
'(PLUS (ADD1 (ZERO)) X)

'(PLUS (ADD1 (QUOTE 0)) X)

'(PLUS (QUOTE 1) X)
```

The reader should note that the notion of "quotation" is not, technically, an extension to the logic, merely a convention for referring to certain constants in it.

3.2. The Subfunctions of V&C

We extend the logic by the definition of several recursive functions. These definitions are all admissible under the definitional principle.

`(ASSOC X ALIST)` returns the first pair in `ALIST` whose `CAR` is `X`, or `F` if no such pair exists:

```
Definition.
(ASSOC X ALIST)
   =
(IF (NLISTP ALIST)
    F
    (IF (EQUAL X (CAAR ALIST))
        (CAR ALIST)
        (ASSOC X (CDR ALIST)))).
```

`(MEMBER X L)` returns `T` or `F` according to whether `X` is an element of `L`:

```
Definition.
(MEMBER X L)
   =
(IF (NLISTP L)
    F
    (IF (EQUAL X (CAR L))
        T
        (MEMBER X (CDR L)))).
```

`(STRIP-CARS L)` returns a list of the successive `CAR`s of the elements of `L`:

```
Definition.
(STRIP-CARS L)
   =
(IF (NLISTP L)
    NIL
    (CONS (CAAR L) (STRIP-CARS (CDR L)))).
```

`(SUM-CDRS L)` returns the sum of the natural numbers in the `CDR`s of the elements of `L`:

```
Definition.
(SUM-CDRS L)
   =
(IF (NLISTP L)
    0
    (PLUS (CDAR L) (SUM-CDRS (CDR L)))).
```

`(PAIRLIST L1 L2)` returns the list of pairs of corresponding elements of `L1` and `L2`:

```
Definition.
(PAIRLIST L1 L2)
   =
(IF (LISTP L1)
    (CONS (CONS (CAR L1) (CAR L2))
          (PAIRLIST (CDR L1) (CDR L2)))
    NIL).
```

(FIX-COST VC N) treats VC as though it is either a "value-cost" pairof FIX-COST is

```
Definition.
(FIX-COST VC N)
   =
(IF VC
    (CONS (CAR VC) (PLUS N (CDR VC)))
    F).
```

In addition we add axioms characterizing four new functions. These functions are essentially just tables that give information about the LITATOMs in the logic corresponding to function symbols in the language. These four functions could be defined for any given extension of the logic. But because the logic may be extended by the user with the shell and definitional principles, these functions are in fact characterized by axiom schemas.

(SUBRP X) is T or F according to whether X is the LITATOM corresponding to one of the primitive function symbols (listed below) or a constructor, recognizer, bottom, or accessor function symbol of a shell. E.g., (SUBRP 'CAR) is T but (SUBRP 'REVERSE) is F. The primitive function symbols are ADD-TO-SET, AND, APPEND, APPLY-SUBR, ASSOC, BODY, COUNT, DIFFERENCE, EQUAL, FALSE, FALSEP, FIX, FIX-COST, FORMALS, GEQ, GREATERP, IF, IMPLIES, LENGTH, LEQ, LESSP, MAX, MEMBER, NLISTP, NOT, OR, ORDINALP, ORDP, ORD-LESSP, PAIRLIST, PLUS, QUANTIFIER-INITIAL-VALUE, QUANTIFIER-OPERATION, QUOTIENT, REMAINDER, STRIP-CARS, SUBRP, SUM-CDRS, TIMES, TRUE, TRUEP, UNION, and ZEROP.

(APPLY-SUBR FN ARGS) "applies" the primitive function or shell function denoted by FN to the appropriate number of elements of ARGS. E.g., (APPLY-SUBR 'CDR (LIST X)) is (CDR X) and (APPLY-SUBR 'EQUAL (LIST X Y)) is (EQUAL X Y). APPLY-SUBR is so defined just on the LITATOMs for which SUBRP is T.

For every user defined function introduced with a definition of the form (fn x_1 ... x_n) = body we have:

(FORMALS 'fn) = **'(x_1 ... x_n)**

and

(BODY 'fn) = **'body.**

Thus

(FORMALS 'APPEND) = **'(X Y)**

```
(BODY 'APPEND) = '(IF (LISTP X)
                      (CONS (CAR X)
                            (APPEND (CDR X) Y))
                      Y)
```

To be perfectly precise, the body in the right-hand side above may differ from that given in two minor ways. One is related to the handling of abbreviations. The other permits the user to define a symbol to be the result returned by EVALuating a given body. Neither is important to the current discussion. See the user's manual [NQTHM-MANUAL] for details.

3.3. The Axiom for V&C

V&C is axiomatized as a function of three arguments, FLG, X, and VA. X is treated either as (a quotation of) a term or else a list of (quotations of) terms, depending on FLG. VA is an association list assigning values to (the quotations of) variable symbols. V&C returns the value and cost of X under VA, if one exists, and F otherwise. Informally, the cost of evaluating an expression is the number of function symbols that must be applied to produce the value with a call-by-value interpreter. If V&C returns F we say X is *undefined* under VA.

There are six cases to consider when X is a term: it is a variable, a constant of some non-LISTP type, a constant embedded in a QUOTE form, an IF expression, an application of a SUBRP or an application of a (presumably) defined function.

Whenever it is necessary to evaluate a subterm of X recursively, we ask whether the result is F and if so return F. Otherwise, the result is a pair containing value and cost components. The value component is used in the determination of the value of X and the cost component is added into the cost of X. STRIP-CARS is used to collect together the value components of a list of evaluated arguments. SUM-CDRS is used to sum their cost components. FIX-COST is used to increment the cost of a recursively obtained "pair," conditional upon the "pair" being a pair rather than F.

The FLG argument is a technical device to handle mutual recursion; if FLG is 'LIST, X represents a list of terms, otherwise, X represents a single term.

Below is the axiom characterizing V&C. Following the conventions of Lisp, comments are delimited on the left by ; and on the right by end-of-line.

Axiom.

```
(V&C FLG X VA)
   =
(IF (EQUAL FLG 'LIST)

    ;X is a list of terms.  Return a list of value-cost
    ;"pairs" -- some "pairs" may be F.

    (IF (NLISTP X)
        NIL
        (CONS (V&C T (CAR X) VA)
              (V&C 'LIST (CDR X) VA)))
```

```
    ;Otherwise, consider the cases on the X.

(IF (LITATOM X)                         ;Variable
    (CONS (CDR (ASSOC X VA)) 0)

(IF (NLISTP X)                          ;Constant
    (CONS X 0)

(IF (EQUAL (CAR X) 'QUOTE)              ;QUOTEd
    (CONS (CADR X) 0)


(IF (EQUAL (CAR X) 'IF)                 ;IF-expr

    ;If the test of the IF is defined, test the value and
    ;interpret the appropriate branch.  Then, if the branch
    ;is defined, increment its cost by that of the test plus
    ;one.  If the test is undefined, X is undefined.

    (IF (V&C T (CADR X) VA)
        (FIX-COST
         (IF (CAR (V&C T (CADR X) VA))
             (V&C T (CADDR X) VA)
             (V&C T (CADDDR X) VA))
         (ADD1 (CDR (V&C T (CADR X) VA))))
        F)

    ;Otherwise, X is the application of a SUBRP or
    ;defined function.  If some argument is undefined, so is X.

(IF (MEMBER F (V&C 'LIST (CDR X) VA))
    F

(IF (SUBRP (CAR X))                     ;SUBRP

    ;Apply the primitive to the values of the arguments and
    ;let the cost be one plus the sum of the argument costs.

    (CONS (APPLY-SUBR (CAR X)
                      (STRIP-CARS (V&C 'LIST (CDR X) VA)))
          (ADD1 (SUM-CDRS (V&C 'LIST (CDR X) VA))))


                                        ;Defined fn

    ;Interpret the BODY on the values of the arguments
    ;and if that is defined increment the cost by one plus
    ;the sum of the argument costs.

    (FIX-COST
     (V&C T (BODY (CAR X))
            (PAIRLIST
              (FORMALS (CAR X))
              (STRIP-CARS (V&C 'LIST (CDR X) VA))))
     (ADD1
       (SUM-CDRS
         (V&C 'LIST (CDR X) VA)))))))))))
```

We also add the axioms that (SUBRP 'V&C) = F, that (FORMALS 'V&C) = '(FLG X VA) and that the BODY of 'V&C is the quotation of the term on the right-hand side of the equal sign in the axiom above.

We make the following claim about V&C. Consider the primitive recursive function that attempts to compute the value of an expression x, under an assignment va of values to its variables, using the recurrence equations specified by BODY, but which counts the total number of function applications used and "fails" when that count exceeds a specified limit n. Call the function Ψ and suppose that when it succeeds it returns a singleton set, {v}, containing the computed value and when it fails it returns the empty set ϕ. Consider any expression x and assignment va. Suppose that there exists an n such that Ψ(x,va,n)={v}. Let k be the least such n. Then (V&C T x va) = <v,k>. If, on the other hand, there exists no such n, (V&C T x va) = F. The nonconstructive assumption in our extended logic is that it is mathematically meaningful to discuss a function that determines whether, for some n, the primitive recursive function Ψ returns non-ϕ.

3.4. Window Dressings

Two trivial but useful theorems about V&C are:

```
Theorem.
(V&C 'LIST L VA)
   =
(IF (NLISTP L)
    NIL
    (CONS (V&C T (CAR L) VA)
          (V&C 'LIST (CDR L) VA)))
```

That is, the V&C of a list of expressions is the list of V&C's.

We can define the auxiliary function V&C-APPLY (see below) so that the following formula is also a theorem:

```
Theorem.
(V&C T X VA)
   =
(IF (LITATOM X)
    (CONS (CDR (ASSOC X VA)) 0)
(IF (NLISTP X)
    (CONS X 0)
(IF (EQUAL (CAR X) 'QUOTE)
    (CONS (CADR X) 0)
    (V&C-APPLY (CAR X)
               (V&C 'LIST (CDR X) VA)))))
```

This theorem tells us that the V&C of an expression can be determined by considering four cases. The first three (variable, constant, and QUOTEd constant) are straightforward. The fourth, function application, is much simpler than perhaps is apparent from the axiom for V&C: the V&C of (fn t_1 ... t_n) is a function only of fn and the V&C's of the t_i's. This holds whether fn is IF, a SUBRP, or a non-SUBRP. The function that determines the V&C of an application is:

```
Definition.
(V&C-APPLY FN ARGS)
   =
(IF (EQUAL FN 'IF)
    (IF (CAR ARGS)
        (FIX-COST (IF (CAAR ARGS)
                      (CADR ARGS)
                      (CADDR ARGS))
                  (ADD1 (CDAR ARGS)))
        F)
 (IF (MEMBER F ARGS)
     F
 (IF (SUBRP FN)
     (CONS (APPLY-SUBR FN
                       (STRIP-CARS ARGS))
           (ADD1 (SUM-CDRS ARGS)))
     (FIX-COST
       (V&C T
            (BODY FN)
            (PAIRLIST
              (FORMALS FN)
              (STRIP-CARS ARGS)))
       (ADD1 (SUM-CDRS ARGS))))))
```

Another important result is that neither the definedness nor the value of a function application is affected by the particular costs of the arguments, provided all the arguments are defined. Another way to put this is that we can replace any "actual" expression by one that always has the same value without affecting the definedness or value of the result. Formally:

```
Theorem.  EQ-ARGS-GIVE-EQ-VALUES:
(IMPLIES
  (AND (NOT (EQUAL FN 'QUOTE))
       (NOT (MEMBER F (V&C 'LIST ARGS1 VA1)))
       (NOT (MEMBER F (V&C 'LIST ARGS2 VA2)))
       (EQUAL (STRIP-CARS (V&C 'LIST ARGS1 VA1))
              (STRIP-CARS (V&C 'LIST ARGS2 VA2))))
  (AND (IFF
         (V&C T (CONS FN ARGS1) VA1)
         (V&C T (CONS FN ARGS2) VA2))
       (EQUAL
         (CAR (V&C T (CONS FN ARGS1) VA1))
         (CAR (V&C T (CONS FN ARGS2) VA2))))).
```

We now introduce some abbreviations. Suppose `'fn` is a `LITATOM` corresponding to an n-ary function symbol `fn`, `'(`v_1 `...` v_n`)` is a list of n distinct `LITATOM` constants, and `'body` is some constant. By

`(fn` v_1 **`...`** v_n**`) @val(ARROW) body`**

we mean

```
   (SUBRP 'fn) = F
@VAL(A) (FORMALS 'fn) = '(v1 ... vn)
@VAL(A) (BODY 'fn) = 'body.
```

Observe that for all user-defined functions

Definition.
(fn v_1 ... v_n) = body

we have

(fn v_1 ... v_n) @val(ARROW) body.

A *semi-concrete alist* corresponding to a substitution {<v_1,t_1>, ..., <v_n,t_n>} is a term of the form (LIST (CONS 'v$_1$ t$_1$) ... (CONS 'v$_n$ t$_n$)). A *standard alist* on a set of variables is a semi-concrete alist corresponding to the identity substitution on those variables.

For example, (LIST (CONS 'X (ADD1 I)) (CONS 'Y (G Z))) is a semi-concrete alist corresponding to the substitution {<X,(ADD1 I)>, <Y,(G Z)>}. The term (LIST (CONS 'X X) (CONS 'Y Y)) is a standard alist on {X, Y}.

If t is a term and σ is a term, then by [t,σ] we mean (V&C T 't σ). When σ is a standard alist on the variables of t we write merely [t]. We write *v*.vc as an abbreviation for the value component, i.e., (CAR vc), and *c*.vc as an abbreviation for the cost component, i.e., (CDR vc). We sometimes write (CONS v c) as <v,c>.

Thus, [(APPEND A B)] is an abbreviation for

```
(V&C T '(APPEND A B)
       (LIST (CONS 'A A)
             (CONS 'B B))).
```

This notation is potentially confusing because while we think of t as a *term* it "becomes" a *constant* in [t]. Care must be exercised when doing substitutions.

Let τ be any substitution. We write t/τ to denote the result of applying τ to term t. Let σ be the semi-concrete alist corresponding to τ. Then [t]/τ is [t,σ].

For example, suppose τ is {<A, (CDR A)>, <B, B>} and t = (APPEND A B). σ is (LIST (CONS 'A (CDR A)) (CONS 'B B)). Then

```
  [t]/τ
=
  (V&C T '(APPEND A B)
         (LIST (CONS 'A A)
               (CONS 'B B)))/τ
=
  (V&C T '(APPEND A B)
         (LIST (CONS 'A (CDR A))
               (CONS 'B B)))
=
  [t,σ].
```

This is a trivial consequence of the definitions of "standard" and "semi-concrete" alists and our [...] notation; it has nothing whatsoever to do with V&C.

We use the notation fn@val(FOPEN)vc$_1$, ..., vc$_n$@val(FCLOSE) to denote (V&C-APPLY 'fn (LIST vc$_1$ ... vc$_n$)). Note that for all function symbols, [(fn t$_1$ ... t$_n$)] = fn@val(FOPEN)[t$_1$], ..., [t$_n$]@val(FCLOSE).

4. Theorems about Partial Functions

We have now a logic obtained from our unquantified logic by the addition of a few axioms. The rules of inference are the same as before. We now show, without proof, a few theorems about V&C to illustrate various aspects of its definition and use.

Suppose

```
(APP X Y)
   @VAL(ARROW)
(IF (EQUAL X (QUOTE NIL))
    Y
    (CONS (CAR X) (APP (CDR X) Y))).
```

This supposition describes a (partial) function. Consider for a moment the recurrence equation analogous to the prescription above[3]

```
Definition?
(APP X Y)
   =
(IF (EQUAL X NIL)
    Y
    (CONS (CAR X) (APP (CDR X) Y))).
```

The recurrence above is similar to APPEND's, but terminates when X is NIL instead of when X is not a LISTP. This equation is inadmissible under the principle of definition because there is no measure of the arguments that decreases in a well-founded sense. In particular, 0 is not NIL and (CDR 0) is 0. Were the above equation an axiom we could derive:

(APP 0 1) = **(CONS 0 (APP 0 1)),**

contradicting the theorem Y @val(NE) (CONS X Y). Nevertheless, with V&C we can investigate the partial function described by this equation.

Here are some theorems about the "partial function" APP:

```
Theorem.
[(APP X Y)]@VAL(NE)F @VAL(IFF) (PROPERP X)

Theorem.
[(APP X Y)]@val(NE)F
@val(I)
v.[(APP X Y)] = (APPEND X Y).
```

The first theorem can be read "(APP X Y) is defined if and only if X is a proper list." A list is proper iff "it ends in a NIL," i.e., the first non-LISTP in its CDR chain is NIL. The second theorem can be read "If (APP X Y) is defined then its value is (APPEND X Y)."

An alternative to the notion of cost in the formalization of an interpreter for partial functions is to introduce a special object, say @val(BTM), that is used as the "value" of undefined interpretations. The definition of the interpreter then passes this value up when it arises in the interpretation of subexpressions. We rejected this approach because @val(BTM) clutters the

---

[3]When used as terms, NIL, 'NIL and (QUOTE NIL) are all abbreviations for the same LITATOM.

statements of virtually all the theorems about interpretations. For example, in the alternative formalization of the definedness condition for (APP X Y) one must include the conditions that no element of X is @val(BTM), no CDR of X is @val(BTM) and that Y is not @val(BTM).[4]

For admissible definitions not involving V&C the V&C theorems are even simpler. Consider APPEND. We have:

```
Theorem.
[(APPEND X Y)]@val(NE)F
@val(A)
v.[(APPEND X Y)] = (APPEND X Y).
```

That is, the partial function described by

```
(APPEND X Y)
   @VAL(ARROW)
(IF (LISTP X)
    (CONS (CAR X) (APPEND (CDR X) Y))
    Y)
```

always terminates and is APPEND.

We next consider the partial function described by:

```
(RUSSELL) @VAL(ARROW) (NOT (RUSSELL)).
```

We can prove:

```
Theorem.
[(RUSSELL)] = F
```

That is, the partial function RUSSELL is everywhere undefined.

We have also dealt with total functions that are uniquely defined by equations that are simply inadmissible under our principle of definition. Consider for example the 91-function:

```
Definition?
(F91 X)
   =
(IF (LESSP 100 X)
    (DIFFERENCE X 10)
    (F91 (F91 (PLUS X 11))))
```

This definition is inadmissible because no measure can be proved to be decreasing in the outermost recursive call above. The reason is that the derived formulas must be proved *before* the new axiom is admitted but any justification of the outermost recursion above must involve properties of F91.

But we can suppose:

---

[4]In our paper on the mechanical proof of the unsolvability of the halting problem, [UNSOLV], we formalized Pure Lisp with an interpretation that used @val(BTM) as described above. The theorem we proved, while valid, was technically inadequate as a formalization of the alleged result because it admitted a trivial proof if one was permitted to construct a Pure Lisp program whose representation included the object @val(BTM). Our oversight was pointed out by a University of Texas graduate student, Jonathan Bellin, and was easily repaired. However, we cite this as an example of the difficulty of coping with @val(BTM) formally.

```
(F91 X)
   @val(ARROW)
(IF (LESSP 100 X)
    (DIFFERENCE X 10)
    (F91 (F91 (PLUS X 11)))).
```

We can then prove the well-known facts about F91:

```
Theorem.
[(F91 X)]@val(NE)F
@val(A)
  v.[(F91 X)] = (G91 X),
```

where (G91 X) is defined to be (IF (LESSP 100 X) (DIFFERENCE X 10) 91).

The most complicated inadmissible function we have investigated with V&C is the unusual list reverse function first shown to us by Rod Burstall:

```
(RV L)
   @VAL(ARROW)
(IF (LISTP L)
    (IF (LISTP (CDR L))
        (CONS (CAR (RV (CDR L)))
              (RV (CONS (CAR L)
                        (RV (CDR (RV (CDR L)))))))
        (CONS (CAR L) 'NIL))
    'NIL)
```

Our theorem prover has proved

```
Theorem.
[(RV X)]@val(NE)F
@val(A)
  v.[(RV X)] = (REVERSE X).
```

5. Proofs about Partial Functions

In this section we prove some of the foregoing theorems to illustrate the simplicity of the logic. We also use these proofs to develop and illustrate the rules for manipulating [...]-expressions.

```
Theorem.
[(RUSSELL)] = F
```

Proof. If [(RUSSELL)] @val(NE) F then the interpretation of the BODY of 'RUSSELL is also non-F and has smaller cost:[5]

```
c.[(RUSSELL)] > c.[(NOT (RUSSELL))].
```

But the cost of the application of a SUBRP is greater than the cost of its argument (provided the argument is defined). Hence:

```
c.[(NOT (RUSSELL))] > c.[(RUSSELL)]
```

[5]In fact, the cost of interpreting the body of RUSSELL is one less than interpreting the call, but it is not necessary that we concern ourselves with the particular costs involved.

Contradiction. Thus, [(RUSSELL)]=F. **Q.E.D.**

We next prove a lemma about APP.

Lemma. APP-0-LOOPS:
APP@val(FOPEN)<0,1>,<Y,0>@val(FCLOSE) = F

The intent of this lemma is to say that APP does not terminate when the value of its first argument is 0. This is true because the CDR of a non-LISTP -- and thus of 0 -- is 0.[6]

Proof. Assume the contrary; i.e., APP@val(FOPEN)<0,1>, <Y,0>@val(FCLOSE) @val (NE) F. Let σ be (LIST (CONS 'X 0) (CONS 'Y Y)). Then

c.**APP@val(FOPEN)<0,1>, <Y,0>@val(FCLOSE)**
> [1]
c.**[(IF (EQUAL X 'NIL)**
Y
(CONS (CAR X) (APP (CDR X) Y))),σ]
> [2]
c.**[(CONS (CAR X) (APP (CDR X) Y)),σ]**
= [3]
c.**CONS@val(FOPEN)[(CAR X),σ],[(APP (CDR X) Y),σ]@val(FCLOSE)**
= [4]
c.**CONS@val(FOPEN)[(CAR X),σ],APP@val(FOPEN)[(CDR X),σ],[Y,σ]@val(FC**
= [5]
c.**CONS@val(FOPEN)[(CAR X),σ],APP(<0,1>,<Y,0>)@val(FCLOSE)**
> [6]
c.**APP@val(FOPEN)<0,1>,<Y,0>@val(FCLOSE).**

Step 1 is via the definition of V&C-APPLY. Step 2 is by the observation that the cost of a defined IF-expression is greater than the cost of the appropriate branch. In the application above we observe that (EQUAL X 'NIL) is defined and has F as its value under σ. Steps 3 and 4 are by appeal to the relation between V&C and V&C-APPLY. Step 5 is by the definition of V&C. Finally, step 6 is via the observation that the cost of a defined SUBRP application is greater than the cost of each argument. Observe that we have again arrived at a contradiction. Hence, APP@val (FOPEN)<0,1>,<Y,0>@val(FCLOSE)=F. **Q.E.D.**

Using the above lemma we can prove:

Theorem. APP-IS-PARTIALLY-APPEND:
[(APP X Y)]@val(NE)F
@val(I)
v.**[(APP X Y)] = (APPEND X Y)**

Proof. We induct on X.

Base Case: (NLISTP X). If X is 'NIL then both sides of the conclusion reduce to Y. Therefore, suppose X is non-'NIL and consider [(APP X Y)].

[(APP X Y)] @VAL(NE) F
@VAL(IFF) [1]

[6]The statement above actually specifies, in addition, that the cost of the first argument be 1. This statement of the theorem suits our current purposes and is easy to prove.

```
  [(IF (EQUAL X (QUOTE NIL))
       Y
       (CONS (CAR X)
             (APP (CDR X) Y)))]@VAL(NE)F
@VAL(IFF)                                              [2]
  [(CONS (CAR X)
         (APP (CDR X) Y))]@VAL(NE)F
@VAL(IFF)                                              [3]
  [(APP (CDR X) Y)]@VAL(NE)F
@VAL(IFF)                                              [4]
  APP@val(FOPEN)[(CDR X)],[Y]@val(FCLOSE)@VAL(NE)F
@VAL(IFF)                                              [5]
  APP@val(FOPEN)<0,1>,<Y,0>@val(FCLOSE)@VAL(NE)F
```

which contradicts APP-0-LOOPS. Step 1 is justified by the fact that the application of a non-SUBRP is defined if and only if the arguments and the body are defined. Step 2 is justified by the fact that an IF is defined if and only if the test and the appropriate branch are defined. Step 3 uses the fact that every SUBRP is defined if and only if the arguments are. Steps 4 and 5 are just the definition of V&C.

Induction Step. We assume (LISTP X). Let σ be a semi-concrete alist corresponding to {<X,(CDR X)>, <Y,Y>}. Our induction hypothesis is:

```
  Induction Hypothesis
  [(APP X Y),σ]@val(NE)F
@val(I)
  v.[(APP X Y),σ] = (APPEND (CDR X) Y)
```

A useful transformation is to observe that this hypothesis is equivalent to:

```
  Induction Hypothesis
  [(APP (CDR X) Y)]@val(NE)F
@val(I)
  v.[(APP (CDR X) Y)] = (APPEND (CDR X) Y)
```

which we can prove using EQ-ARGS-GIVE-EQ-VALUES.

We assume [(APP X Y)]@val(NE)F and consider [(APP X Y)], using the same rules used in the base case.:

```
  [(APP X Y)]@VAL(NE)F
@VAL(IFF)
  [(IF (EQUAL X 'NIL)
       Y
       (CONS (CAR X)
             (APP (CDR X) Y)))]@VAL(NE)F
@VAL(IFF)
  [(CONS (CAR X) (APP (CDR X) Y))]@VAL(NE)F
@VAL(IFF)
  [(APP (CDR X) Y)]@VAL(NE)F.
```

We can thus detach the conclusion of our induction hypothesis:

```
  v.[(APP (CDR X) Y)] = (APPEND (CDR X) Y).
```

Then, appealing to the just derived chain of definedness results we can derive the value of (APP X Y):

```
  v.[(APP X Y)]
=                                          [1]
  v.[(IF (EQUAL X 'NIL)
         Y
         (CONS (CAR X)
               (APP (CDR X) Y)))]
=                                          [2]
  v.[(CONS (CAR X) (APP (CDR X) Y))]
=                                          [3]
  v.CONS@val(FOPEN)[(CAR X)],[(APP (CDR X) Y)]@val(FCLOSE)
=                                          [4]
  (CONS v.[(CAR X)] v.[(APP (CDR X) Y)])
=                                          [5]
  (CONS (CAR X) v.[(APP (CDR X) Y)])
=                                          [6]
  (CONS (CAR X) (APPEND (CDR X) Y))
=                                          [7]
  (APPEND X Y).
```

Step 1 uses the observation that if the application of a non-SUBRP is defined, then its value is the value of the body. Step 2 uses the corresponding rule for IF. Step 3 is by the previously mentioned relation between V&C and V&C-APPLY. Step 4 is by the observation that, for SUBRP applications that are defined, the value of the application is the primitive function applied to the values of the arguments. Step 6 uses the induction hypothesis. Step 7 is by the definition of APPEND. **Q.E.D.**

The reader should not be discouraged by the length of the proof just presented. The length is a reflection of the fact that we have carefully presented each step. The proof is not deep or complicated. Indeed, it is just the mechanical application of a set of very useful rules for manipulating definedness conditions, and value and cost expressions. We summarize the rules informally here:

Definedness: A variable or QUOTEd constant is always defined. An IF-expression is defined when the test and the appropriate branch are. The call of a SUBRP other than IF is defined if and only if all the arguments are. Finally, a call of a non-SUBRP is defined if and only if all of the arguments are defined and the body is defined.

Value Expressions: The value of a variable or QUOTEd constant is straightforward. The value of a defined IF-expression is the value of the appropriate branch, depending on the value of the test. The value of a defined SUBRP call is the corresponding primitive function applied to the values of the arguments. The value of a defined non-SUBRP call is the value of the body.

Cost Expressions: The cost of a variable or QUOTEd constant is 0. The cost of a defined IF-expression is greater than the costs of both the test and the appropriate branch. The cost of a defined SUBRP call other than IF is greater than the cost of each argument. The cost of a defined non-SUBRP call is greater than the cost of each argument and the cost of the body.

This collection of rules can be expressed as theorems. For example, the definedness condition for calls of SUBRPs may be written:

```
Theorem.
(IMPLIES
   (AND (NOT (EQUAL FN 'IF))
        (SUBRP FN))
   (IFF (V&C T (CONS FN ARGS) A)
        (NOT (MEMBER F (V&C 'LIST ARGS A))))).
```

The formal statement of the "definedness condition for SUBRPs" as a theorem in the logic may look inelegant. One might have expected a *metatheorem*:

Metatheorem.
If `fn` is a primitive function symbol other than `IF`,
then
[(fn t_1 ... t_n),σ] @val(NE) F
if and only if
[t_1,σ]@val(NE)F @val(A) ... @val(A) [t_n,σ]@val(NE)F.

However, when applied to quoted terms the theorem above leads directly to the desired conclusions. The SUBRP hypothesis is relieved by computation given any quoted function symbol and the (`V&C 'LIST ...`) and `MEMBER` expressions expand appropriately on quoted argument lists.

For example, observe how the theorem handles the formula:

```
[(CONS (CAR X) (APP (CDR X) Y))]@VAL(NE)F.
```

Let σ be the standard alist (`LIST (CONS 'X X) (CONS 'Y Y)`). By the abbreviation conventions the above formula is equivalent to:

```
(V&C T '(CONS (CAR X)
              (APP (CDR X) Y))
       σ)@val(NE)F
@VAL(IFF)
  (V&C T (CONS 'CONS
               '((CAR X) (APP (CDR X) Y)))
         σ)@val(NE)F.
```

By the definedness condition for SUBRP calls, above, it is equivalent to

```
(NOT (MEMBER F (V&C 'LIST
                    '((CAR X) (APP (CDR X) Y))
                    σ))),
```

which by the definitions of `MEMBER` and V&C is equivalent to:

```
(NOT (MEMBER F
       (LIST (V&C T '(CAR X) σ)
             (V&C T '(APP (CDR X) Y) σ))))
@VAL(IFF)
  (V&C T '(CAR X) σ)@val(NE)F
    @val(A)
  (V&C T '(APP (CDR X) Y) σ)@val(NE)F
@VAL(IFF)
  [(CAR X),σ]@val(NE)F @VAL(A) [(APP (CDR X) Y)]@val(NE)F
@VAL(IFF)
  [(CAR X)]@val(NE)F @VAL(A) [(APP (CDR X) Y)]@val(NE)F.
```

The last line is justified by the fact that if an alist (σ) contains a binding of a variable (`'Y`) not

occurring in the term ((CAR X)), then that binding may be dropped. We show the formal statement of such a theorem later.

The corresponding "value theorem for SUBRP calls" is

```
Theorem.
(IMPLIES (AND (SUBRP FN)
              (V&C T (CONS FN ARGS) A))
         (EQUAL
           (CAR (V&C T (CONS FN ARGS) A))
           (APPLY-SUBR FN
                       (STRIP-CARS
                        (V&C 'LIST ARGS A))))).
```

Once again, when this rule is applied to a quoted term the unfamiliar function symbols APPLY-SUBR, STRIP-CARS and (V&C 'LIST ...) expand out and the rule permits us to behave just as though we had the

Metatheorem.
For every SUBRP fn, including IF, **if**
[(fn t_1 ... t_n),σ]@val(NE)F, then
v.**[(fn t_1 ... t_n),σ]**
=
(fn *v*.**[t_1,σ] ...** *v*.**[t_n,σ]).**

A cost-manipulation theorem is shown below. It is the rule that relates the cost of a defined call of a non-SUBRP to the cost of the body.

```
Theorem.
(IMPLIES
  (AND (NOT (SUBRP FN))
       (V&C T (CONS FN ARGS) A))
  (GREATERP
    (CDR (V&C T (CONS FN ARGS) A))
    (CDR (V&C T (BODY FN)
              (PAIRLIST (FORMALS FN)
                        (STRIP-CARS
                          (V&C 'LIST ARGS A)))))).
```

All of the theorems shown above may be proved in a straightforward way from the axiom defining V&C and the definitions of its subfunctions.

Using the above rules we can prove the following metatheorem, which is the first step towards the most important metatheorem in our implementation of the modified theorem prover. We are interested in a syntactically defined class of terms with the property that, if t is in the class, then [t]@val(NE)F and *v*.[t] = t.

Call a term *primitive* if it is composed entirely of variable symbols, constants, QUOTEd constants, and applications of SUBRPs to such terms. (LISTP X) and (CONS (CAR X) 'NIL) are examples of primitive terms. Observe that:

[(LISTP X)] @val(NE) F

v.**[(LISTP X)]** = **(LISTP X)**

[(CONS (CAR X) 'NIL)] @val(NE) F

v.**[(CONS (CAR X) 'NIL)]** = **(CONS (CAR X) NIL).**

It is easy to prove, by induction on the structure of primitive terms, that if `t` is primitive `[t]`@val (NE)`F` and *v*.`[t] = t`. The key observation is that a call of a `SUBRP 'fn` is defined if the arguments are and that the value of the call is the application of the function `fn` to the values of the arguments.

We would like to widen the syntactic class to include user defined functions. At this point, we will show how to include all of the admissible functions that do not involve `V&C` and its dependents. We eventually expand the class still further to include some uses of `V&C`. We call the expanded class the *tame* terms.

Using the techniques illustrated above, we can prove:

Theorem. APPEND-X-Y-IS-TAME:
[(APPEND X Y)]@val(NE)F
@val(A)
v.**[(APPEND X Y)]** = **(APPEND X Y).**

The proof is by induction on `X` by `CDR` and follows exactly the outline of the proof for `APP-IS-PARTIALLY-APPEND`. The proof is left to the reader.

Note that the lemma, as stated, does not tell us about arbitrary `APPEND` expressions, just `'(APPEND X Y)`. It is useful to "lift" the lemma to arbitrary `APPEND` expressions. Suppose we have `APPEND-X-Y-IS-TAME` and wish to prove:

```
Theorem.  APPEND-IS-TAME:
(IMPLIES
   (AND (V&C T U A)
        (V&C T V A))
   (AND (V&C T (LIST 'APPEND U V) A)
        (EQUAL
          (CAR (V&C T (LIST 'APPEND U V) A))
          (APPEND (CAR (V&C T U A))
                  (CAR (V&C T V A))))))),
```

which might be abbreviated:

Metatheorem.
If **[u]@val(NE)F** and **[v]@val(NE)F** then
[(APPEND u v)]@val(NE)F
and
v.**[(APPEND u v)] = (APPEND** *v*.**[u]** *v*.**[v]).**

Note that this theorem lets us do for `APPEND` exactly what we can do for `SUBRP`s: Provided the argument expressions are defined, a call of `APPEND` is defined and has as its value the `APPEND` of the values of its arguments. Intuitively, this property means we could include `APPEND` among the function symbols permitted in "primitive" terms.

Proof of `APPEND-IS-TAME`. The proof is immediate from `APPEND-X-Y-IS-TAME` and the previously shown `EQ-ARGS-GIVE-EQ-VALUES`, by instantiating the variables in the latter lemma as follows:

```
    ARGS1:      '(X Y)
    VA1:        (LIST (CONS 'X X) (CONS 'Y Y))
    ARGS2:      (LIST U V)
    VA2:        A
    FN:         'APPEND
Q.E.D.
```

Observe that the proof of APPEND-IS-TAME from APPEND-X-Y-IS-TAME is not concerned with APPEND, just V&C. That is, we could similarly "lift" analogous theorems about other function symbols.

An analogous theorem can be proved about each admissible function symbol that does not rely (at any level) upon V&C. In particular, from the properties of the "primitives" and the "tameness" of the subfunctions of an admissible function, we can prove the "tameness" of each admissible function. We complete our detailed discussion of proofs about partial functions by illustrating this claim.

The function we will focus on is REVERSE:

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

We will prove

```
    Theorem.  REVERSE-X-IS-TAME:
    [(REVERSE X)]@val(NE)F
  @val(A)
    v.[(REVERSE X)] = (REVERSE X).
```

Proof. We induct on X. The base case, when (LISTP X) = F, is trivial.

In the induction step we assume (LISTP X) = T and the induction hypothesis (which we simplify, as before, by moving the substitution {<X, (CDR X)>} inside the quoted expression using EQ-ARGS-GIVE-EQ-VALUES):

```
    Induction Hypothesis.
    [(REVERSE (CDR X))]@val(NE)F
  @val(A)
    v.[(REVERSE (CDR X))] = (REVERSE (CDR X)).
```

We consider the definedness condition first.

```
    [(REVERSE X)]@val(NE)F
  @val(IFF)
    [(IF (LISTP X)
         (APPEND (REVERSE (CDR X))
                 (CONS (CAR X) 'NIL))
         'NIL)]@val(NE)F
  @val(IFF)
```

```
[(APPEND (REVERSE (CDR X))
         (CONS (CAR X) 'NIL))]@val(NE)F
```

Since

```
[(REVERSE (CDR X))]@val(NE)F
```

by the induction hypothesis, and

```
[(CONS (CAR X) 'NIL)]@val(NE)F
```

since the expression is primitive, we get, from APPEND-IS-TAME, that

```
[(APPEND (REVERSE (CDR X))
         (CONS (CAR X) 'NIL))]@val(NE)F
```

and hence [(REVERSE X)]@val(NE)F. Furthermore,

```
v.[(REVERSE X)]
   =
v.[(APPEND (REVERSE (CDR X))
           (CONS (CAR X) 'NIL))]
```

which, by APPEND-IS-TAME,

```
   =
(APPEND v.[(REVERSE (CDR X))]
        v.[(CONS (CAR X) 'NIL)])
   =
(APPEND (REVERSE (CDR X))
        (CONS (CAR X) NIL))
   =
(REVERSE X)
```

Q.E.D.

The other theorems of the previous section have similar proofs. For example in proving that (F91 X) is defined and has as its value (G91 X), the proof is by an induction in which one assumes two instances of the conjecture, one for X replaced by (PLUS X 11) and the other for X replaced by (G91 (PLUS X 11)), under the hypothesis that X@val(LTE)100. The measure justifying this induction is (DIFFERENCE 101 X). The proof is straightforward by the techniques developed.

The unquantified version of our theorem prover can construct these proofs from the axiom for V&C and the definitions shown. It is necessary for the user to guide the theorem prover by the appropriate choice of lemmas to prove first.

6. EVAL and APPLY

Recall our interest in EVAL: we use it in the definition of FOR to evaluate the conditional and body expressions. At first sight, an appropriate definition of (EVAL 't σ) is *v*.[t,σ], i.e.,

Impression:
(EVAL X A) = (CAR (V&C T X A)).

However, recall that in our proof of

```
Theorem.  SUM-DISTRIBUTES-OVER-PLUS:
(FOR I L P 'SUM (LIST 'PLUS G H) A)
   =
(PLUS (FOR I L P 'SUM G A)
      (FOR I L P 'SUM H A)).
```

we assumed the following fact about EVAL:

```
Lemma.  EVAL-DISTRIBUTES-OVER-PLUS:
(EVAL (LIST 'PLUS X Y) A)
   =
(PLUS (EVAL X A) (EVAL Y A)).
```

But is it the case that *v*.[(PLUS x y)] = (PLUS *v*.[x] *v*.[y])? It certainly is if x and y are defined. But if x is not defined then [(PLUS x y)] is undefined. Thus, the left-hand value expression above becomes *v*.F, which happens to be 0. But the right-hand side above is (PLUS *v*.[x] *v*.[y]), which is (PLUS 0 *v*.[y]), not 0.

We could correct the situation by defining (CAR x), i.e., *v*.x, to be @val(BTM) on non-lists and define PLUS and all other functions to return @val(BTM) when one of their arguments is @val (BTM). But we then lose the familiar elementary properties of many functions For example, (TIMES 0 X) = 0 would no longer be a theorem. In addition, then all our proofs about such elementary functions, even proofs not involving partial functions, would have to consider @val (BTM). We believe this is too high a price to pay for convenient quantifier manipulation.

But having EVAL-DISTRIBUTES-OVER-PLUS be an unconditional equality gives us powerful and easily used rules like the unconditional SUM-DISTRIBUTES-OVER-PLUS. We achieve this by defining EVAL in a more complicated way than merely the CAR of V&C.

```
Definition.
(EVAL FLG X A)
   =
(IF (EQUAL FLG 'LIST)
    (IF (NLISTP X)
        NIL
        (CONS (EVAL T (CAR X) A)
              (EVAL 'LIST (CDR X) A)))
 (IF (LITATOM X)  (CDR (ASSOC X A))
 (IF (NLISTP X)   X
 (IF (EQUAL (CAR X) 'QUOTE) (CADR X)
     (APPLY (CAR X)
            (EVAL 'LIST (CDR X) A))))),
```

where APPLY is as defined below.

The FLG argument plays the same role in EVAL as it does in V&C. We henceforth ignore it, sometimes even writing (EVAL t a) for (EVAL T t a). Observe that for every function symbol fn we have:

```
(EVAL (LIST 'fn X1 ... Xn) A)
   =
(APPLY 'fn (LIST (EVAL X1 A) ... (EVAL Xn A))).
```

Without even knowing the definition of APPLY this insures several very elegant theorems about EVAL, for example:

Theorem. Irrelevant bindings can be deleted:
```
(IMPLIES (NOT (MEMBER V (FREE-VARS X)))
         (EQUAL (EVAL X (CONS (CONS V VAL) A))
                (EVAL X A)))
```

Theorem. Substitution of equal EVALs:
```
(IMPLIES (EQUAL (EVAL X A) (EVAL Y A))
         (EQUAL (EVAL (SUBSTITUTE Y X Z) A)
                (EVAL Z A)))
```

where FREE-VARS returns the free variables in the quotation of a term and (SUBSTITUTE Y X Z) substitutes Y for X in Z. These theorems can be proved by the unquantified version of the theorem prover from the definition of EVAL without any knowledge of APPLY. In particular, the definition of EVAL makes it clear that the value of an s-expression denoting a function application is entirely determined by the function applied and the (recursively obtained) values of the argument s-expressions.

We wish it to be the case that

(APPLY 'PLUS (LIST X Y)) = **(PLUS X Y)**

We can arrange this for 'PLUS and for every other total function in the logic by defining:

```
Definition.
(APPLY FN ARGS)
   =
(CAR (V&C-APPLY FN (PAIRLIST ARGS 0)))
```

For example, by APPEND-X-Y-IS-TAME, we get

```
  (APPLY 'APPEND (LIST X Y))
= v.APPEND@val(FOPEN)<X,0>,<Y,0>@val(FCLOSE)
= v.[(APPEND X Y)]
= (APPEND X Y).
```

For partial functions introduced with "@val(ARROW)" and then proved total, such as the unusual reverse function RV, we have analogous simple theorems relating the result of APPLY to the application of the corresponding total function. For example,

```
  (APPLY 'RV (LIST X))
= v.RV(<X,0>)
= v.[(RV X)]
= (REVERSE X).
```

But we do not automatically get, for all function symbols fn,

(APPLY 'fn (LIST X1 ... Xn)) = **(fn X1 ... Xn).**

For example, it is not the case that:

(APPLY 'V&C (LIST FLG X VA)) = **(V&C FLG X VA).**

7. The Definition of the Quantifier Function FOR

Having defined EVAL and APPLY, we then introduce FOR as shown in the Introduction.

The two main subfunctions of FOR, in addition to EVAL, are defined as follows:

Definition.

```
(QUANTIFIER-INITIAL-VALUE OP)
   =
(CDR (ASSOC OP '((ADD-TO-SET . NIL)
                 (ALWAYS . *1*TRUE)
                 (APPEND . NIL)
                 (COLLECT . NIL)
                 (COUNT . 0)
                 (DO-RETURN . NIL)
                 (EXISTS . *1*FALSE)
                 (MAX . 0)
                 (SUM . 0)
                 (MULTIPLY . 1)
                 (UNION . NIL))))
```

Definition.

```
(QUANTIFIER-OPERATION OP X Y)
   =
(IF (EQUAL OP 'ADD-TO-SET) (ADD-TO-SET X Y)
(IF (EQUAL OP 'ALWAYS)     (AND X Y)
(IF (EQUAL OP 'APPEND)     (APPEND X Y)
(IF (EQUAL OP 'COLLECT)    (CONS X Y)
(IF (EQUAL OP 'COUNT)      (IF X (ADD1 Y) Y)
(IF (EQUAL OP 'DO-RETURN)  X
(IF (EQUAL OP 'EXISTS)     (OR X Y)
(IF (EQUAL OP 'MAX)        (MAX X Y)
(IF (EQUAL OP 'SUM)        (PLUS X Y)
(IF (EQUAL OP 'MULTIPLY)   (TIMES X Y)
(IF (EQUAL OP 'UNION)      (UNION X Y)
                            0)))))))))))
```

8. Theorems about Quantifiers

In this section we show many simple theorems about FOR that illustrate the power of the notation. We start with theorems that can be proved from the definition of FOR without knowledge of EVAL.

Theorem.

```
(FOR X (APPEND A B) COND 'COLLECT BODY ALIST)
   =
(APPEND (FOR X A COND 'COLLECT BODY ALIST)
        (FOR X B COND 'COLLECT BODY ALIST))
```

Theorem.

```
(FOR X (APPEND A B) COND 'COUNT BODY ALIST)
   =
(PLUS (FOR X A COND 'COUNT BODY ALIST)
      (FOR X B COND 'COUNT BODY ALIST))
```

```
Theorem.
(FOR X (APPEND A B) COND 'ADD-TO-SET BODY ALIST)
   =
(UNION (FOR X A COND 'ADD-TO-SET BODY ALIST)
       (FOR X B COND 'ADD-TO-SET BODY ALIST))


Theorem.
(FOR X (APPEND A B) COND 'DO-RETURN BODY ALIST)
   =
(IF (FOR X A (LIST 'QUOTE T) 'EXISTS COND ALIST)
    (FOR X A COND 'DO-RETURN BODY ALIST)
    (FOR X B COND 'DO-RETURN BODY ALIST))

Theorem.
(IMPLIES
  (MEMBER OP '(ALWAYS MAX EXISTS SUM
               APPEND MULTIPLY UNION))
  (EQUAL
   (FOR X (APPEND A B) COND OP BODY ALIST)
   (QUANTIFIER-OPERATION OP
      (FOR X A COND OP BODY ALIST)
      (FOR X B COND OP BODY ALIST))))
```

The next group of theorems are schematic in nature. Note however that we do not need second order variables to state these theorems. Note also that they are simple equalities and are not encumbered by hypotheses about the well-definedness of the expressions to which they are applied. Thus, these theorems are easily used in simplification. Their proofs require knowledge of EVAL and APPLY.

```
Theorem.
(FOR X R COND 'SUM (LIST 'PLUS G H) ALIST)
   =
(PLUS (FOR X R COND 'SUM G ALIST)
      (FOR X R COND 'SUM H ALIST))

Theorem.
(FOR X R COND 'MULTIPLY
              (LIST 'TIMES G H) ALIST)
   =
(TIMES (FOR X R COND 'MULTIPLY G ALIST)
       (FOR X R COND 'MULTIPLY H ALIST))

Theorem.
(FOR X R COND 'ALWAYS (LIST 'AND G H) ALIST)
   =
(AND (FOR X R COND 'ALWAYS G ALIST)
     (FOR X R COND 'ALWAYS H ALIST))

Theorem.
(FOR X R COND 'EXISTS (LIST 'OR G H) ALIST)
   =
(OR (FOR X R COND 'EXISTS G ALIST)
    (FOR X R COND 'EXISTS H ALIST))
```

The following theorem may be read "if the bound variable does not occur freely in the body

then the body is constant."

```
Theorem.
(IMPLIES
   (NOT (MEMBER X (FREE-VARS BODY)))
   (EQUAL (FOR X R COND OP BODY ALIST)
          (FOR X R COND OP
            (LIST 'QUOTE (EVAL BODY ALIST))
            ALIST)))
```

A similar theorem can be proved about the conditional expression.

There is a class of theorems about constant bodies. For example:

```
Theorem.
(FOR X R COND 'SUM (LIST 'QUOTE BODY) ALIST)
   =
(TIMES BODY
       (FOR X R (LIST 'QUOTE T)
            'COUNT COND ALIST))
```

Finally, we can prove theorems that allow us to manipulate the range of a quantifier. We illustrate such a theorem when we discuss the proof of the Binomial Theorem.

9. The Modified Theorem Prover

Perhaps the major appeal of the logic we have presented is that our existing mechanical theorem prover is sound for it. Furthermore, that theorem prover is capable of proving the foregoing theorems from the axioms and definitions for V&C, EVAL, and FOR. However, the theorem prover's performance can be improved greatly by building in some of the properties of these three function symbols. We discuss in this section several of the improvements we have made.

Most of our improvements are based on the metatheoretic notion of a "tame" term. For every tame term `t` we have the metatheorem *v.*`[t]` = t. A term is *tame* if it is a variable, a constant, the application of a "total" function to tame terms, a term of the form `(V&C T 't alist)` or `(EVAL T 't alist)` where `t` and `alist` are tame, or a term of the form `(FOR v r 'cond op 'body alist)` where each of `v`, `r`, `cond`, `op`, `body` and `alist` are tame.

We classify each function symbol as to whether it is total or not at the time it is introduced, as a function of the previously introduced functions. Intuitively, `fn` is total if its body is tame, which means, roughly, that every function called in the body is total. However, the body may involve recursive calls of `fn`. Do we assume `fn` is total when we determine whether its body is tame? At first glance the answer seems to be "yes, because we have proved, during the admission of the function, that every recursive call of `fn` decreases a measure of the arguments in a well-founded sense." But consider calls of `fn` occurring inside of quoted expressions given to EVAL. Those calls have not been checked. For example, assuming `fn` total and then asking if its body is tame would result in the following function being considered total:

```
Definition.
(RUSSELL) = (NOT (EVAL '(RUSSELL) NIL)),
```

since, if RUSSELL is considered total then the EVAL expression above is tame. Therefore, in the

definition of "total" we do not use the notion of "tame" and instead use the notion of "super-tame" which means that fn is considered total outside of EVAL expressions but not inside. Here are the precise definitions of "total" and "super-tame."

All the primitives except V&C, V&C-APPLY, EVAL, APPLY and FOR are *total*. A function defined by (fn x_1 ... x_n)=body is *total* iff body is super-tame with respect to fn.

A term is *super-tame* with respect to fn iff it is a tame term (under the assumption that fn is not total) or it is a call of a total function or fn on super-tame terms.

Every function definable in our old logic is total. In addition, functions in the new logic that involve V&C, EVAL and FOR are total provided the interpreted arguments are the quotations of tame terms.

The fundamental theorem about tame terms is

Metatheorem.
If t is a tame term and σ is a semi-concrete alist corresponding to a substitution τ on the variables of t

then
[t,σ]@val(NE)F
and
v.[t,σ] = t/τ.

When we discussed proofs of theorems about partial functions we illustrated the proof of the metatheorem. We proved that REVERSE expressions have the tameness property above, given the tameness property for APPEND. The proof was by induction by CDR on the argument of REVERSE. More generally, any newly admitted total function can be proved to have the tameness property, given the tameness property of all the previously admitted total functions. The proof is by induction on the measure justifying the newly admitted function. The only problematic part of the proof is that we included among the tame terms applications of V&C to the quotations of tame terms. Proving the tameness property for such applications of V&C itself is simple given the following fundamental property of V&C: If (V&C T X A) is non-F then [(V&C T X A)] is non-F and has (V&C T X A) as its value. The property can be proved by induction on the cost component of (V&C T X A). It is, in addition, the case that if (V&C T X A) is F then so is [(V&C T X A)].

We use the metatheorem in several ways. An obvious one is that whenever the theorem prover encounters a term of the form:

(CAR (V&C T 't σ))

where t is a tame term and σ is a semi-concrete alist corresponding to a substitution τ on the variables of t, it is replaced by t/τ.

We have similar simplification routines for V&C-APPLY (on total functions) EVAL, and APPLY (on total functions). The most complicated simplifier is for the FOR function.

When the theorem prover encounters a term of the form:

```
(FOR (QUOTE v) r
     (QUOTE cond)
     op
     (QUOTE body)
     alist)
```

where v is a variable symbol, cond and body are tame terms, and alist a semi-concrete alist corresponding to a substitution τ on the variables in cond and body we: instantiate cond and body with τ (deleting any pair of the form <v, t>), rewrite the instantiated terms to obtain cond′ and body′, and then, provided the results are tame terms, return:

```
(FOR (QUOTE v) r
     (QUOTE cond')
     op
     (QUOTE body')
     alist')
```

where alist′ is a standard alist on the variables in cond′ and body′. We take care, when rewriting cond and body to rename v if there are hypotheses about v. In addition, we then assume the hypothesis (MEMBER v r) when rewriting cond and we assume that and cond′ when rewriting body.[7]

In all we added about 10 pages of Lisp code to the theorem prover to support the simplification of expressions involving V&C and its cousins. However, the reader is reminded that it was unnecessary to change or even inspect any existing code; we merely added new simplifiers for the new function symbols.

10. Proof of the Binomial Theorem

In this section we prove the Binomial Theorem:

$$(a+b)^n = \sum_{i=0}^{n} \binom{n}{i} a^i b^{n-i};$$

When using FOR (as opposed to proving schematic theorems about FOR) we have found the following abbreviation convention helpful. Let *cond* and *body* be terms and let σ be a standard alist on the variables in *cond* and *body*. Then we write

(for *v* **in** *r* **when** *cond op body***)**

as an abbreviation for

(FOR ′*v r* ′*cond* ′*op* ′*body* σ**).**

When *cond* is T we often omit it and the when "key-word."

In our notation, the Binomial Theorem is:

Theorem.

---

[7]Recall the previously mentioned litany of well-known mathematicians who got the rules wrong when dealing with the instantiation of higher-order variables. The metatheorems that establish the rules for simplifying inside the body of FORs are exactly the places one would expect such mistakes to be made. Despite our caution we goofed here the first time we implemented the simplification. We neglected the possibility that the user-supplied alist of the FOR might contain a binding for the indicial variable -- a binding that is completely irrelevant under the axioms but which had effect in our implementation. Our error was caught by a Matt Kaufmann, a colleague at the University of Texas.

```
(EQUAL (EXP (PLUS A B) N)
       (for I in (FROM-TO 0 N) sum
         (TIMES (BC N I)
                (EXP A I)
                (EXP B (DIFFERENCE N I)))))
```

where `(FROM-TO I J)` is the list of the natural numbers from `I` to `J`, `(TIMES i j k)` is an abbreviation for `(TIMES i (TIMES j k))` and the binomial coefficient is defined recursively as:

```
Definition.
(BC N M)
   =
(IF (ZEROP M)
    1
    (IF (LESSP N M)
        0
        (PLUS (BC (SUB1 N) M)
              (BC (SUB1 N) (SUB1 M)))))
```

Proof of the Binomial Theorem. We induct on `N`. The base case, when `N` is `0`, is trivial; both sides reduce to 1.

In the induction step we know `N`>0 and we assume:

```
Induction Hypothesis.
(EQUAL (EXP (PLUS A B) (SUB1 N))
       (for I in (FROM-TO 0 (SUB1 N))
         sum
         (TIMES (BC (SUB1 N) I)
                (EXP A I)
                (EXP B
                   (DIFFERENCE (SUB1 N) I)))))
```

As noted previously, the substitution of `(SUB1 N)` for `N` technically changes only the alist of the `FOR`, not the quoted body expression above. However, as we have done in our previously shown proofs, we can drive that substitution in, since both the body and the substituted terms are tame. In our implementation this transformation is carried out by the previously mentioned `FOR` simplifier, when it instantiates the quoted body of the `FOR`, rewrites the result, and quotes it again.

We now prove the induction step by deriving the left-hand side of the Binomial Theorem from the right-hand side, assuming the inductive hypothesis above and `N`>0. We will use standard arithmetic notation for `PLUS`, `TIMES`, `EXP`, and `DIFFERENCE`. After the derivation we comment upon each step. The derivation is somewhat long because we show each step.

```
(for I in (FROM-TO 0 N)
     sum (BC N I)*A^I*B^(N-I))
   =                                        [1]
(for I in (CONS 0 (FROM-TO 1 N))
     sum (BC N I)*A^I*B^(N-I))
   =                                        [2]
(BC N 0)*A^0*B^(N-0)
    + (for I in (FROM-TO 1 N)
         sum (BC N I)*A^I*B^(N-I))
   =                                        [3]
```

```
B^N + (for I in (FROM-TO 1 N)
          sum (BC N I)*A^I*B^(N-I))

   =                                                    [4]
B^N + (for I in (FROM-TO 1 N)
          sum [(BC N-1 I)+(BC N-1 I-1)]
                *
              A^I*B^(N-I))
   =                                                    [5]
B^N + (for I in (FROM-TO 1 N)
          sum (BC N-1 I)*A^I*B^(N-I)
                +
              (BC N-1 I-1)*A^I*B^(N-I))
   =                                                    [6]
B^N + (for I in (FROM-TO 1 N)
          sum (BC N-1 I)*A^I*B^(N-I))
   + (for I in (FROM-TO 1 N)
          sum (BC N-1 I-1)*A^I*B^(N-I))
   =                                                    [7]
(for I in (FROM-TO 0 N)
     sum (BC N-1 I)*A^I*B^(N-I))
  +
(for I in (FROM-TO 1 N)
     sum (BC N-1 I-1)*A^I*B^(N-I))
   =                                                    [8]
(for I in (APPEND (FROM-TO 0 N-1) (LIST N))
     sum (BC N-1 I)*A^I*B^(N-I))
  +
(for I in (FROM-TO 1 N)
     sum (BC N-1 I-1)*A^I*B^(N-I))
   =                                                    [9]
(for I in (FROM-TO 0 N-1)
     sum (BC N-1 I)*A^I*B^(N-I))
  +
(for I in (LIST N)
     sum (BC N-1 I)*A^I*B^(N-I))
  +
(for I in (FROM-TO 1 N)
     sum (BC N-1 I-1)*A^I*B^(N-I))
   =                                                    [10]
(for I in (FROM-TO 0 N-1)
     sum (BC N-1 I)*A^I*B^(N-I))
  +
(for I in (FROM-TO 1 N)
     sum (BC N-1 I-1)*A^I*B^(N-I))
   =                                                    [11]
(for I in (FROM-TO 0 N-1)
     sum (BC N-1 I)*A^I*B*B^((N-1)-I))
  +
(for I in (FROM-TO 1 N)
     sum (BC N-1 I-1)*A^I*B^(N-I))
   =                                                    [12]
B*(for I in (FROM-TO 0 N-1)
       sum (BC N-1 I)*A^I*B^((N-1)-I))
  +
(for I in (FROM-TO 1 N)
     sum (BC N-1 I-1)*A^I*B^(N-I))
```

```
   =                                                          [13]
B*(for I in (FROM-TO 0 N-1)
       sum (BC N-1 I)*A^I*B^(N-1)-I)
  +
(for I in (FROM-TO 0 N-1)
     sum (BC N-1 I)*A^I+1*B^N-(I+1))
   =                                                          [14]
B*(for I in (FROM-TO 0 N-1)
       sum (BC N-1 I)*A^I*B^(N-1)-I)
  +
(for I in (FROM-TO 0 N-1)
     sum (BC N-1 I)*A*A^I*B^(N-1)-I)
   =                                                          [15]
B*(for I in (FROM-TO 0 N-1)
       sum (BC N-1 I)*A^I*B^(N-1)-I)
  +
A*(for I in (FROM-TO 0 N-1)
       sum (BC N-1 I)*A^I*B^(N-1)-I)
   =                                                          [16]
(A+B)*(for I in (FROM-TO 0 N-1)
           sum (BC N-1 I)*A^I*B^(N-1)-I)
   =                                                          [17]
(A+B)*(A+B)^N-1
   =                                                          [18]
(A+B)^N
Q.E.D.
```

Step 1 is the expansion of `(FROM 0 N)`.

Step 2 is the expansion of `FOR`. The first term in the resulting formula is the body of the `FOR` with `I` replaced by `0`. That term is produced by the `EVAL` simplifier since the body is tame.

In step 3 the instantiated body is simplified to B^N, using simple arithmetic.

Step 4 is the expansion of `(BC N I)` inside the body of the `FOR`. This is done by the `FOR` simplifier, when it rewrites the body under the hypothesis that `I` is a member of the range. The definition of `BC` has two "base" cases, one where `N` is `0` and the other where `I` exceeds `N`. We are not in the former because `N>0`. We are not in the latter because `I` is a member of `(FROM-TO 1 N)`. Thus, the `BC` expression expands to the sum of two recursive calls.

Step 5 is further simplification inside the body of the `FOR`. We distribute the multiplication over the addition of the two `BC` calls.

Step 6 is an application of the quantifier rewrite rule that the `sum` of a `PLUS` is the `PLUS` of the `sums`.

In step 7 we extend the range of the first `for` from `(FROM-TO 1 N)` to `(FROM-TO 0 N)` by observing that at `I=0` the extended `for` sums in the B^N term added explicitly to the earlier `for`.

Step 8 is the expansion of `(FROM-TO 0 N)` at the high end.

Step 9 distributes `for` over an `APPEND`ed range.

Step 10 drops the middle `for` of the previous formula because it is equal `0`. The simplification is done by expanding the definition of `for` and using the `EVAL` simplifier on the instantiated body; when `I=N` the body of the `for` is `0` because of the definition of `BC`.

Step 11 is arithmetic simplification inside the body of the first quantifier.

Step 12 is an appeal to a `FOR` lemma that permits factors not containing the bound variable to be moved outside of `sums`. Observe that the first `for` in the resulting formula is the right hand side of the induction hypothesis. We are not yet ready to use the induction hypothesis however; we wish to manipulate the second `for` in formula 13 to make the same term appear there.

In step 13 we apply a `FOR` lemma that shifts the range down by `1` but increments each occurrence of the bound variable by `1`. The theorem can be stated:

```
Theorem.
(IMPLIES (AND (LITATOM V) (NOT (ZEROP J)))
  (EQUAL (FOR V (FROM-TO J K) COND OP BODY ALIST)
         (FOR V (FROM-TO (SUB1 J) (SUB1 K))
              (SUBSTITUTE (LIST 'ADD1 V) V COND)
              OP
              (SUBSTITUTE (LIST 'ADD1 V) V BODY)
              ALIST)))
```

Step 14 is arithmetic simplification inside the body of the second `for`.

Step 15 brings out a constant factor again, here an `A`. Note that the two `fors` in the resulting formula are identical: they are the right hand side of the induction hypothesis.

In step 16 we apply the distribution law of multiplication over addition.

In step 17 we appeal to the induction hypothesis.

In step 18 we use the definition of exponentiation.

Our mechanical proof uses the same reasoning as that shown above, but the steps are somewhat different because the theorem prover reduces both the left and right hand sides to the same formula.

The reader is encouraged to note that even though the proof involves several interesting quantifier manipulation techniques, the most commonly used manipulations are the expansion of recursive definitions and the simplification of list processing and arithmetic expressions both inside of and outside of the quantifiers.

11. Conclusion

We have described an extension to our computational logic and its supporting theorem prover that provides the power of partial functions and quantification over finite domains.

The extension consists of the following steps, starting with the logic described in [ACL]

1. adopt the notational conventions of [meta];

2. add definitions for the functions ASSOC, PAIRLIST, MEMBER, FIX-COST, STRIP-CARS and SUM-CDRS;
3. add axioms to define the functions SUBRP, APPLY-SUBR, FORMALS and BODY -- these unproblematic functions could be defined as simple tables for a given definitional/shell extension of the logic;
4. add an axiom characterizing the function V&C, an interpreter for the logic which returns the value and cost of any expression in the logic or F if the expression has no interpretation -- V&C is not a computable function but is uniquely characterized by the axiom given; and
5. define a variety of useful functions in terms of V&C, namely, V&C-APPLY, EVAL, APPLY, and FOR.

The most attractive feature of our extension is that it does not alter the term structure of our language, the axioms already present, or the rules of inference. There are two advantages to this. First, the statements and proofs of theorems not involving partial functions or quantified expressions are unaffected by the provision of these features. Second, the existing mechanical theorem prover is sound for the extended logic; changes made in support of the extension were concerned entirely with simplifiers for handling the new function symbols, not consideration of the correctness of existing code.

These considerations are important because a logic or mechanical theorem prover that supports quantification or partial functions is merely an academic exercise unless it provides extensive support for the primitive theories of arithmetic, sequences, trees, etc. Quantifiers and partial functions are used to discuss the operations and objects in those theories. Even proofs involving extensive quantifier manipulation, such as the Binomial Theorem, are usually dominated by the quantifier-free manipulations at the core. We believe it is a mistake to sacrifice the simplicity of the core theories for the sake of quantifiers or partial functions.

That said, however, it is important that the provisions for new features not be so protective of the primitives that it is awkward to use the new features. We have addressed this concern in this paper by using the new features to state and prove theorems that could not previously be stated in our logic.

We have shown how the termination properties of many partial recursive functions could be stated in terms of V&C and proved formally in the extended logic. We are unaware of any other mechanical proof of, say, the correctness of the unusual reverse function RV.

We have shown how schematic quantifier manipulation laws can be stated, proved and used in the logic. We used such laws in the proof of the Binomial Theorem. We are unaware of any other mechanical proof of this theorem.

We have described several simplification techniques that are useful in manipulating V&C, EVAL and FOR expressions. We have implemented these techniques in an experimental extension to our mechanical theorem prover. We have demonstrated these techniques at work in the proof of the Binomial Theorem.

In summary, we believe we have produced a practical extension of our logic and theorem

prover that supports partial functions and bounded quantification.

References

Table of Contents