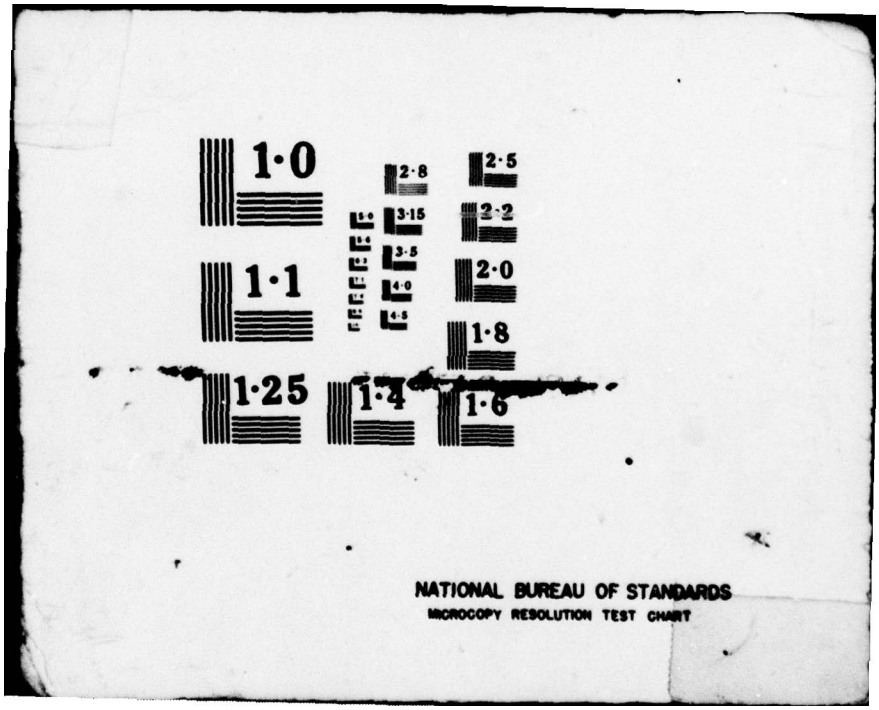
AD-A071 421 STANFORD UNIV CALIF DEPT OF COMPUTER SCIENCE F/G 9/2
RECURSIVE PROGRAMS AS FUNCTIONS IN A FIRST ORDER THEORY.(U)
MAR 79 R CARTWRIGHT, J MCCARTHY MDA903-76-C-0206
UNCLASSIFIED STAN-CS-79-717 NL


1·0
1·1
1·25
2·8
3·15
3·5
4·0
4·5
1·4
2·5
2·2
2·0
1·8
1·6
NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

(12) LEVEL II

Stanford Artificial Intelligence Laboratory
Memo AIM-324

March 1979

Computer Science Department
Report No. STAN-CS-79-717

RECURSIVE PROGRAMS AS FUNCTIONS IN A FIRST ORDER THEORY

by

Robert Cartwright and John McCarthy

Research sponsored by

Advanced Research Projects Agency
and
National Science Foundation

COMPUTER SCIENCE DEPARTMENT
Stanford University

Stanford Artificial Intelligence Laboratory
Memo AIM-324

March 1979

Computer Science Department
Report No. STAN-CS-79-717

RECURSIVE PROGRAMS AS FUNCTIONS IN A FIRST ORDER THEORY

by

Robert Cartwright and John McCarthy

Pure Lisp style recursive function programs are represented in a new way by sentences and schemata of first order logic. This permits easy and natural proofs of extensional properties of such programs by methods that generalize structural induction. It also systematizes known methods such as *recursion induction, subgoal induction, inductive assertions* by interpreting them as first order axiom schemata. We discuss the metatheorems justifying the representation and techniques for proving facts about specific programs. We also give a simpler version of the Goedel-Kleene way of representing computable functions by first order sentences. This paper is to be published in *Proceedings of The International Conference on Mathematical Studies of Information Processing*, edited by S. Takasu and published by Springer-Verlag.

Robert Cartwright's current address is Department of Computer Science, Upson Hall, Cornell University, Ithaca, NY 14853. John McCarthy's current address is Department of Computer Science, Stanford University, Stanford, CA 94305.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under contract MDA903-76-C-0206 and by the National Science Foundation under grants NSF MCS 78-00524 and MCS 78-05850. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University, Cornell University, or any agency of the U. S. Government.

1. Introduction and Motivation.

This paper advances two aspects of the study of the properties of computer programs - the scientifically motivated search for general theorems that permit deducing properties of programs and the engineering problem of replacing debugging by computer-assisted computer-checked proofs that programs have desired properties. Both tasks require mathematics, but the second also requires keeping a non-mathematical goal in mind - getting short completely formal proofs that are easy to write and check by computer.

A pure Lisp style recursive program P defines a partial function f_P. By adjoining an undefined element $\perp$ (read "bottom") to the data domains, f_P may be extended to a total function which we denote by the same symbol. In (Cartwright 1976), it was shown that f_P satisfies a *functional equation*, which is a sentence in a first order theory T_P. Besides the functional equation, T_P contains symbols for the basic functions, predicates and constants of the data domain, axioms for the data domain and its extension by $\perp$, and additional function symbols for the functions defined by recursive programs. (Cartwright 1976) also showed how the functional equation can be used to prove facts about the program by reasoning within T_P, including the fact that f_P is total, i.e. doesn't take the value $\perp$ except when $\perp$ is an argument.

When f_P is total, and sometimes when it isn't, it is completely characterized by the functional equation. Otherwise, the characterization can be completed by a *minimization schema* (McCarthy 1978 and this paper) or alternatively by a *complete recursive function* as first defined in (Cartwright 1978). Moreover, we show how to find a representation of f_P by a sentence of the form $(\forall x)(y = f_P(x) \equiv A(x))$ where $A(x)$ is a wff of T_P not involving f_P.

Now assume that T_P contains functions sufficient for axiomatizing basic syntax, e.g. Lisp or elementary number theory, and let S be a sentence of T_P involving only f_P and the basic functions of the data domain. Then (Cartwright and McCarthy 1979) shows how to construct a sentence S' involving only the basic functions of the data domain such that we can prove in first order logic that $S \equiv S'$. Therefore, the fact that total correctness is not axiomatizable in first order logic is just a matter of the Gödelian incompleteness of the data domain, and it can be expected that all "ordinary" facts about programs will be provable just as all "ordinary" facts of elementary number theory are provable in spite of its incompleteness.

This paper is primarily concerned with proving such "ordinary" facts about recursive function programs with a view to developing practical techniques for program verification using interactive theorem provers. As such it should be compared with other ways of using logic in program proving.

Axiomatizing programs as functions compares favorably with Floyd-Hoare methods in several respects. First it permits stating and proving facts that cannot even be stated in Floyd-Hoare formalisms such as equivalence of programs and algebraic relations between the functions defined by programs. It has the advantage compared to the Scott-Strachey formalisms that it uses ordinary first order logic rather than a logic of continuous functions. This permits the use of any mathematical facts that can be expressed in first order logic, including those that are most conveniently expressed in set theory. This is especially important when the statement of program

correctness or its informal proof involve other mathematical objects than those that occur in the program data domain.

After an informal introduction to recursive programs, subsequent sections of this paper discuss the use of conditional expressions and first order lambdas in first order logic, adjoining $\perp$ to the data domains in order to convert partial functions and predicates into total functions, axioms for Lisp and the integers, the representation of recursive programs by functions, inductive methods of proof, the minimization schema, an extended example of a correctness proof, representation of the inductive assertion and subgoal induction methods as axiom schemata, and a convenient way of representing recursively defined functions by non-recursive sentences.

Our methods apply directly to proving only *extensional* properties of programs, e.g. properties of the function defined by the program. Intensional properties such as the number of times an operation like recursion or *cons* is performed are often extensional properties of simply obtained *derived programs*. Some of these properties are also extensional properties of the functional of which the function is the least fixed point.

An adequate background for this paper is contained in (Manna 1974) and more concisely in (Manna, Ness and Vuillemin 1973). The connections of recursive programs with second order logic are given in (Cooper 1969) and (Park 1970). Our notation differs from Manna's in order to use the = sign exactly as in first order logic.

2. Recursive Programs.

We consider recursive programs like

Factorial: $n! \leftarrow$ if n *equal* 0 then 1 else $n \cdot (n-1)!$

which is the well known recursive program for the factorial function. We will use capitalized italic names for programs themselves regarded as texts and the corresponding name initialized with lower case as a name for the function computed by the program, except that as in the case of *Factorial*, we sometimes use an infix or other conventional notation for the function. Mutually recursive sets of function programs will also be considered.

Another example is the Lisp program *Append*. In this paper we will use the Lisp external or publication notation of (McCarthy and Talcott 1979), and we will write $u*v$ for $append[u, v]$. We then have

Append: $u * v \leftarrow$ if n u then v else a u . [d $u * v$].

Here we are using n for *null*, a for *car*, d for *cdr* and an infixed . for *cons*. We omit brackets for functions of one argument. In a more traditional Lisp M-expression notation we would have

$$append[u, v] \leftarrow \text{if } null[u] \text{ then } v \text{ else } cons[car[u], append[cdr[u], v]],$$

and in Maclisp S-expression notation, this would be

```
(DEFUN APPEND (U V)
    (COND ((NULL U) V) (T (CONS (CAR U) (APPEND (CDR U) V))))).
```

Our objective is to prove facts about such recursively defined functions by regarding the recursive function definitions as sentences of first order logic. More accurately, we represent the recursive function definitions by very similar sentences of first order logic. *Factorial* and *Append* are translated into the sentences

1) $(\forall n)(iseint\ n \supset n! = \text{if } n\ equal\ 0 \text{ then } 1 \text{ else } n \times (n-1)!)$

and

2) $(\forall u\ v)(iselist\ u \wedge iselist\ v \supset u * v = \text{if } n\ u \text{ then } v \text{ else a } u\ .\ [d\ u * v])$

respectively. The form of conditional expression if p then a else b used in these sentences is just a function that could as well be written $if(p, a, b)$ so far as the logic is concerned.

The predicates *iseint* and *iselist* respectively restrict their arguments to be extended integers (i.e. the integers extended by $\perp$) and extended lists. When these domains can be taken for granted, we can omit the explicit restrictions and write

3) $(\forall n)(n! = \text{if } n\ equal\ 0 \text{ then } 1 \text{ else } n \times (n-1)!)$

and

4) $(\forall u\ v)(u * v = \text{if } n\ u \text{ then } v \text{ else a } u\ .\ [d\ u * v])$

The sentences (1) and (2) completely characterize the functions defined by the programs *Factorial* and *Append*, so proofs of the properties of these functions can be deduced from these sentences together with axioms characterizing the natural number and Lisp data domains respectively. For example, suppose we wish to prove that * satisfies the equations

5) $(\forall v)(\text{NIL} * v = v)$

and

6) $(\forall u)(u * \text{NIL} = u)$,

i.e. NIL is both a left and right identity for the * operation. (5) is trivially obtained by substituting NIL for u in (1) and using the rules for evaluating conditional expressions which will have been added to the usual rules for first order logic. (6) expresses a more typical program property in that its proof requires a mathematical induction.

This induction is accomplished by substituting

7) $\Phi(u) \equiv (u * \text{NIL} = u)$

in the list induction schema

8) $\Phi(\mathrm{NIL}) \wedge (\forall u)(\mathit{islist}\ u \wedge \neg \mathit{null}\ u \wedge \Phi(\mathrm{d}\ u) \supset \Phi(u)) \supset (\forall u)(\mathit{islist}\ u \supset \Phi(u))$,

and using (2), the axioms for lists, and the rules of inference of first order logic including those for conditional expressions.

Once the formalism has been established, totality can be proved in the same way as other properties of the programs. Thus the totality of $u*v$ is proved by substituting

9) $\Phi(u) \equiv \mathit{islist}[u*v]$

into the schema (8) and using (2), etc. as described above.

The translation of the program into a logical sentences would be trivial to justify if we were always assured that the program terminates for all sets of arguments and thus defines a total function. The innovation is that the translation is possible even without that guarantee at the cheap price of extending the data domain by an undefined element $\perp$, rewriting recursively defined predicate programs as function programs, having two kinds of equality and conditional expression, and providing each predicate with two forms - one a genuine predicate in the logic and the other a function imitating the partial predicate by a function that takes the value $\perp$ when the program for the predicate doesn't terminate. Proofs of termination then take the same form as other inductive proofs. However, additional formalism is required to characterize completely programs that don't always terminate.

The next sections introduce the logical basis of the formalism and axioms and axiom schemata for Lisp.

3. Two Useful Extensions to First Order Logic.

We begin by extending first order logic to include conditional expressions and first order lambda expressions. This allows us to parallel the structure of recursive programs within logical sentences.

We cannot add arbitrary programming constructions to first order logic without risking its useful properties such as completeness or even consistency. Fortunately, these extensions are harmless, because they are not merely conservative; they can even be eliminated from wffs, and they are generally useful. In fact, they are useful for expressing mathematical ideas concisely and understandably quite apart from applications to computer science. The reader is assumed to know about first order logic, conditional expressions and lambda expressions; we explain only their connection.

Remember that the syntax of first order logic is given in the form of inductive rules for the formation of terms and wffs. The rule for forming terms is extended as follows:

If P is a wff and a and b are terms, then *IF P THEN a ELSE b* is a term. Sometimes parentheses must be added to insure unique decomposition. Note that this makes the definitions of term and wff mutually recursive.

The semantics of conditional expression terms is given by a rule for determining their values. Namely, if P is true, then the value of *IF P THEN a ELSE b* is the value of a. Otherwise it is the value of b.

It is also necessary to add rules of inference to the logic concerned with conditional expressions. One could get by with rules permitting the elimination of conditional expressions from sentences and their introduction. These rules are important anyway, because they permit proof of the metatheorem that the main properties of first order logic are unaffected by the addition of conditional expressions. These include completeness, the deduction theorem, and semi-decidability.

In order to state these rules it is convenient to introduce conditional expressions also as a ternary logical connective. A more fastidious exposition would use a different notation for logical conditional expressions, but we will use them so little that we might as well use the same notation, especially since it is not ambiguous. Namely, if P, Q, and R are wffs, then so is *IF P THEN Q ELSE R*. Its semantics is given by considering it as a synonym for $((P \wedge Q) \vee (\neg P \wedge R))$. Elimination of conditional expressions is made possible by the distributive laws

10) $$f(\textit{IF } P \textit{ THEN } a \textit{ ELSE } b) = \textit{IF } P \textit{ THEN } f(a) \textit{ ELSE } f(b)$$

and

11) $$\Phi(\textit{IF } P \textit{ THEN } a \textit{ ELSE } b) \equiv \textit{IF } P \textit{ THEN } \Phi(a) \textit{ ELSE } \Phi(b) \equiv (P \wedge \Phi(a)) \vee (\neg P \wedge \Phi(b))$$

where f and Φ stand for arbitrary function and predicate symbols respectively.

Notice that this addition to the logic has nothing to do with partial functions or the element $\perp$.

While the above rules are sufficient to preserve the completeness of first order logic, proofs are often greatly shortened by using the additional rules introduced in (McCarthy 1963). We mention especially an extended form of the rule for replacing an expression by another expression proved equal to it. Suppose we want to replace the expression c by an expression c' within the conditional expression *IF P THEN a ELSE b*. To replace an occurrence of c within a, we need not prove $c = c'$ but merely $P \supset c = c'$. Likewise if we want to replace an occurrence of c in b, we need only prove $\neg P \supset c = c'$. This principle is further extended in the afore-mentioned paper.

A further useful and eliminable extension to the logic is to allow "first order" lambda expressions as function and predicate expressions. Thus if x is an individual variable, e is a term, and P is a wff, then $(\lambda x)e$ and $(\lambda x)P$ may be used wherever a function symbol or predicate symbol respectively are allowed. Formally, this requires that the syntactic categories of <function

symbol> and <predicate symbol> be generalized to <function expression> and <predicate expression> respectively and that these categories are then defined mutually recursively with terms and wffs.

The only inference rule required is lambda conversion which serves to eliminate or introduce lambda expressions. According to this rule, a wff is equivalent to a wff obtained from it by replacing a sub-wff or sub-term by one obtained from it by lambda conversion. The rules for lambda conversion must include alphabetic changes of bound variables when needed to avoid capture of the free variables in arguments of lambda expressions.

The use of minimization schemata and schemata of induction is facilitated by first order lambda expressions, since the substitution just replaces occurrences of the function variable in the schema by a lambda expression which can subsequently be expanded by lambda conversion. Using lambda expressions somewhat simplifies the rule for substitution in schemata. First order lambda expressions also permit many sentences to be expressed more compactly and may be used to avoid duplicate computations in recursive programs. Thus we can write $[(\lambda x)(x^2 + x)](a + b)$ instead of $(a + b)^2 + (a + b)$. Since all occurrences of first order lambda expressions can be eliminated from wffs by lambda conversion, the metatheorems of first order logic are again preserved. The reason we don't get the full lambda calculus is that the syntactic rules of first order logic prevent a variable from being used in both term and function positions. While we have illustrated the use of lambda expressions with single variable λ's, expressions like $(\lambda x\ y\ z)e$ are also useful and give no trouble. It is also easily seen that lambda conversion within a term preserves its value, and lambda conversion within a wff preserves its truth value.

Actually it seems that even higher order λ's won't get us out of first order logic provided rules of typing are obeyed and we provide no way of quantifying over function variables. Any occurrences of higher order lambda expressions in wffs are eliminable just by carrying out the indicated lambda conversions. For example, we could define

$$transitive = (\lambda R)((\forall X\ Y\ Z)(R(X, Y) \wedge R(Y, Z) \supset R(X, Z))),$$

and any use of *transitive* in a wff would be eliminable using its definition and lambda conversion.

4. Partial Functions and Partial Predicates.

The main difficulty to be overcome in representing recursive programs by logical sentences is that the computation of an arbitrary recursive program cannot be guaranteed to terminate. Consider the recursive program

Runaway: $f(n) \leftarrow f(n) + 1$

over the integers. If we translate *Runaway* into the sentence

12) $(\forall n)(f(n) = f(n) + 1)$

and use the axioms of arithmetic, we get a contradiction.

The way out is to adjoin to our data domains an additional element $\perp$ (read "bottom"), which is taken to be the value of the function when the computation doesn't terminate. In addition we add the axiom

13) $(\forall n)(isint(n) \vee n = \perp)$,

and modify the axioms for arithmetic to refer to elements satisfying *isint*. Then going from *Runaway* to (12) doesn't lead to a contradiction but to the desired result that

14) $(\forall n)(f(n) = \perp)$,

provided we also postulate that

15) $(\forall n)(n + \perp = \perp + n = \perp)$,

which is reasonable given the interpretation of $\perp$ as the value of a computation that doesn't terminate. We will postulate that all of the base functions, except the conditional expression, have $\perp$ as value if any argument is $\perp$. Such functions are called *strict*. Manna (1974) calls them *natural extensions* of the functions defined on the domain without $\perp$.

We have discussed treating partial functions by introducing $\perp$. A function takes the value $\perp$ when the program that computes it doesn't terminate, and it is sometimes convenient to give a function the value $\perp$ in some other cases when we want it to be undefined.

It is convenient to introduce a rather trivial partial ordering relation on our data domain once it has been extended by adjoining $\perp$. Namely, we define the relation $X \sqsubset Y$ by

16) $(\forall X\ Y)(X \sqsubset Y \equiv X = \perp \wedge Y \neq \perp)$.

(Readers of (Manna 1974) should note that the symbol $\equiv$ is being used in its common logical sense of "if and only if"). We also make corresponding definitions of $\sqsupset$, $\sqsubseteq$, and $\sqsupseteq$. The ordering can be extended to functions by defining

17) $f \sqsubseteq g \equiv (\forall X)(f(X) \sqsubseteq g(X))$.

This induced ordering is not so trivial, but we don't use it in this paper, since it gets us out of first order logic. Even though (16) defines a rather trivial ordering, we find that it shortens and clarifies many formulas.

Partial predicates give rise to new problems. The computation of a recursively defined predicate may not terminate, so the same problem arises. However, we can't solve it in the same way without adding an additional undefined truth value to the logic. This would give rise to a partial first order logic in which sentences could be true, false or undefined. Various partial predicate calculi have been studied in (McCarthy 1964), (Bochvar 1938 and 1943) and elsewhere,

but they have the serious disadvantage that arguments by cases become quite long, since three cases always have to be provided for, even when most of the predicates are known to be total. Moreover, existing logic texts, proof-checkers and theorem provers all use total logic. Therefore, it seems better to keep the logic conventional and handle partial predicates as functions.

We introduce a domain Π with three elements T, F and $\perp$ called the domain of extended truth values. In a sorted logic, this may be a separate sort. Otherwise, it may be considered either separately or as part of the main data domain. In Lisp it is convenient to regard T and F as special atoms and to use the same $\perp$ for extended truth values as for extended S-expressions. It is even possible to follow the Lisp implementations that use NIL for F and interpret all other S-expressions as T, although we don't do that in this paper.

It is convenient to define first a form of conditional expression that takes an extended truth value as its first argument, namely

$$\text{if } p \text{ then } a \text{ else } b = IF\ p = \perp\ THEN\ \perp\ ELSE\ IF\ p = T\ THEN\ a\ ELSE\ b.$$

The only difference between then extended conditional expression and the logical conditional expression is that since the extended conditional expression takes an extended truth value as propositional argument, we can provide for the possibility that the computation of that argument fails to terminate. Since the extended conditional expression treats the undefined cases according to their behavior in programs, we use the same notation as previously used for programs.

Extended boolean operators are conveniently defined using the extended conditional expressions. For every predicate or boolean operator, we introduce a corresponding function taking extended truth values as operands and taking an extended truth value as its value. Thus the function *and*, is written with an infix and defined by

$$p\ and\ q = \text{if } p \text{ then } q \text{ else } F$$

The function *and* is distinct from the logical operator $\wedge$ which remains in the logic. To illustrate this, we have the true sentence

$$((p\ and\ q) = T) \equiv (p = T) \wedge (q = T).$$

The parentheses in the above can be omitted without ambiguity given suitable precedence rules. Note that *and* has the non-commutative property of (McCarthy 1963), namely $F\ and\ \perp = F$ while $\perp\ and\ F = \perp$. This corresponds to the fact that it is convenient to compute $p\ and\ q$ by a program that doesn't look at q if p is false but which doesn't terminate if the computation of p doesn't terminate. Symmetry could be restored if the computer time-shared its computations of p and q, but there are too many practical disadvantages to such a system to justify doing it for the sake of mathematical symmetry. Algol 60 requires that both p and q be computed which precludes using boolean opeators as the main connectives of Lisp type recursive definitions of predicates.

Other extended boolean operators are defined by

$$p\ or\ q = \text{if } p \text{ then } T \text{ else } q$$

and

not p = if p then F else T.

We also require an equality function that extends logical equality, namely

X *equal* Y = *IF* $X = \bot \vee Y = \bot$ *THEN* $\bot$ *ELSE IF* $X = Y$ *THEN* T *ELSE* F.

Readers familiar with (Manna 1974) should note that we write = where Manna writes ≡, and we write *equal* where Manna writes =. We have chosen our notation to conform to that of first order logic with equality.

In fact, the key to successful representation of recursive programs in first order logic is the simultaneous use of true equality in the logic in order to make assertions freely and the *equal* function that gives an undefined result for undefined arguments. The latter describes the behavior of an equality test within the program. The two forms of conditional expression are also essential.

The partial ordering $\sqsubset$ is also useful applied to extended truth values.

We summarize this in the following set of axioms:

T1: $(\forall p)$*istv* $p \equiv p = T \vee p = F$)

T2: $(\forall p)$*isetv* $p \equiv$ *istv* $p \vee p = \bot$)

T3: $T \neq F \wedge \neg$*istv* $\bot$

T4: $(\forall p\ X\ Y)$*isetv* $p \supset$
if p then X else Y = *IF* $p = \bot$ *THEN* $\bot$ *ELSE IF* $p = T$ *THEN* X *ELSE* Y)

T5: $(\forall p)$*isetv* $p \supset$ *not* p = if p then F else T)

T6: $(\forall p\ q)$*isetv* $p \wedge$ *isetv* $q \supset p$ *and* q = if p then q else F)

T7: $(\forall p\ q)$*isetv* $p \wedge$ *isetv* $q \supset p$ *or* q = if p then T else q)

T8: $(\forall X\ Y)X$ *equal* Y = *IF* $X = \bot \vee Y = \bot$ *THEN* $\bot$ *ELSE IF* $X = Y$ *THEN* T *ELSE* F

T9: $(\forall p)$*isetv* $p \wedge$ *isetv* $q \supset (p \sqsubset q \equiv p = \bot \wedge (q = T \vee q = F)))$.

5. The Functional Equation of a Recursive Program - Theory.

The familiar recursive program

18) $u * v \leftarrow$ if $n\ u$ then v else $a\ u\ .\ [d\ u * v]$

is a special case of a system of mutually recursive programs which can be written

19) $f_1(x_1, \ldots, x_{m_1}) \leftarrow \tau_1(f_1, \ldots, f_n, x_1, \ldots, x_{m_1})$

$\cdot$

$\cdot$

$\cdot$

$f_n(x_1, \ldots, x_{m_n}) \leftarrow \tau_n(f_1, \ldots, f_n, x_1, \ldots, x_{m_n})$.

Here the τ's are terms in the individual variables x_1, etc. and the function symbols $f_1, \ldots f_n$. All the essential features of such mutual recursive definitions arise when there is only one function, but phenomena arise when there are two or more arguments to the functions that do not arise in the one argument case - two arguments being sufficiently general. Therefore, we write

20) $f(x, y) \leftarrow \tau(f, x, y)$,

which may also be written

21) $f(x, y) \leftarrow \tau[f](x, y)$

when we wish to emphasize that τ maps a partial function f into another partial function $\tau[f]$.

In this paper, we shall mainly consider recursive programs over S-expressions, lists and integers, but we can actually start with an arbitrary collection of base functions and predicates over a collection of domains and define the functions *computable in terms of the base functions*. This is discussed in (McCarthy 1963). In a discussion of the basic ideas, full generality is superfluous, and all the interesting phenomena arise with a single domain - call it D, extended to D^+ by adjoining $\perp$ and with characteristic predicate isD.

Such a program or system of mutually recursive programs can be regarded as defining a partial function in several ways.

1. It can be compiled into a machine language program for some computer using call-by-value. The resulting program is a subroutine that calls itself recursively. Before it is called, the values of the arguments must be computed and stored in suitable conventional registers. This includes its calls to itself. Most Lisp implementations as well as most implementations of other programming languages use call-by-value.

2. It can be compiled into a machine language program for some computer using call-by-name. The resulting program again calls itself recursively. It is called by storing into suitable registers the location of programs for computing the expressions that have been written as its arguments.

Thus $((w.z)*f(u))$ would be compiled into program that would give the program for $u*v$ pointers to program for computing $w.z$ and $f(u)$. The program for $*$ could call these other programs whenever it wanted its arguments. In the case of $u*v$, there is nothing the program can profitably do except call for both of its arguments. However, a program for multiplying two matrices might call its first argument, and, if the first argument turned out to be the zero matrix, not bother to call the second argument.

We can also consider evaluating the function by symbolic computation. Namely, we substitute the arguments of the function $*$ for u and v, and then evaluate the right hand side of the definition. There are many ways to do this evaluation, because there may be more than one occurrence of the function being defined on the right hand side of the definition, but two of them correspond to call-by-name and call-by-value respectively.

3. When evaluating a conditional expression, always evaluate the propositional term first and use it to decide which of the other terms to evaluate first. When evaluating a term formed by composition of functions, if there is only one occurrence of the function being defined on the right hand side, there is no choice to be made, but if there is more than one, expand the leftmost innermost first. If it gives an answer substitute it and continue the process. If it gives further recursion, then proceed with its leftmost innermost, etc. This corresponds to call-by value.

4. If instead of expanding the leftmost innermost occurrence of the function first, we expand the outermost occurrences, we get an evaluation method corresponding to call-by-name.

It should also be proved that evaluation by substitution and evaluation by subroutine both using call-by-value give the same results. The two ways of doing call-by-name should also be proved to give the same results. Such a proof would involve reasoning about the operation of subroutine calls and the saving of temporary storage registers on the stack. We are not aware of a published proof of these statements or even a precise statement of them.

Computing $u*v$ doesn't show the difference between these methods, but consider the function

22) $\quad morris(x, y) \leftarrow$ if x *equal* 0 then 0 else $morris(x - 1,\ morris(x,\ y))$

introduced in (Morris 1968). Evaluating $morris(2, 1)$ by either call-by-value method leads to an infinite computation, because the term $morris(x, y)$ has to be evaluated all over. Call-by-name evaluation, on the other hand, gives the answer 0, because the second argument of *morris* is never called. Vuillemin (1973) shows that whenever call-by-value gives an answer, call-by-name gives the same answer, but sometimes call-by-name gives an answer when call-by-value doesn't. If we force a program to be *strict*, i.e. to demand that all of its arguments are defined, then call-by-name and call-by-value are equi-terminating - to coin a word.

(Manna 1974) also contains proofs of these assertions.

Execution of recursive programs by substitution is inefficient, but provides a good theoretical tool for classifying the more efficient subroutine methods of evaluation.

5. Finally, we can regard (18) and (22) as functional equations for * and *morris* respectively. In general, a functional equation may have many solutions or none. However, it is essentially Kleene's (1952) first recursion theorem, (see Manna 1974, theorem 5-2) that if the right side is *continuous* in the function being defined and in the individual variables, there will be a unique *minimal* solution. This condition is assured if the right hand side is a term built from strict functions and predicates by composition and the formation of extended conditional expressions. Continuity is discussed in (Manna 1974). It is not permitted to use logical conditional expressions without satisfying additional hypotheses, and this restriction prevents true equality or any predicate from direct use. If logical conditional expressions were generally allowed, we could have sentences like

23) $(\forall x)(f(x) = IF\ f(x) = \bot\ THEN\ T\ ELSE\ \bot)$

which are self-contradictory. The corresponding version using extended conditional expressions, namely

24) $(\forall x)(f(x) =$ if $f(x)$ *equal* $\bot$ then T else $\bot)$

is satisfied by $f(x) = \bot$ and is therefore harmless. Logical conditional expressions can be used when we can guarantee that the propositional part is total and in some other cases.

The minimal solution is minimal in the sense that any other solution is greater in the ordering of functions previously given, i.e. if f is the minimal solution and ϕ is another solution, then

25) $(\forall x\ y)(f(x, y) \sqsubseteq \phi(x, y))$.

The minimal solution of the functional equation can therefore be characterized by the schema

26) $(\forall x\ y)(\phi(x, y) = \tau[\phi](x, y)) \supset (\forall x\ y)(f(x, y) \sqsubseteq \phi(x, y))$.

6. Axioms for S-expressions, Lists and Integers.

The collection of axioms Lisp1 allows for the possibility that there are other kinds of entity besides S-expressions, lists and integers. In practical program proving, these will include sets and data structures of various kinds. In consequence of this decision, we need the predicates *issexp*, *islist* and *isint* to pick out S-expressions, lists and integers respectively. Lists are considered to be a particular kind of S-expression, namely S-expressions such that going in the *cdr* direction eventually reaches NIL. It is convenient to have both the predicates *atom* and *ispair* that pick out atomic and non-atomic S-expressions respectively.

Lisp1 is convenient for making proofs and is intended to treat S-expressions, lists and

integers as similarly as possible. Therefore, the axioms are highly redundant. Adjoining $\perp$ to the domains has both conveniences and inconveniences. The main convenience is that the recursive definitions now give total functions. A major inconvenience is that algebraic relations often require qualification, e.g. $0 \times x = 0$ isn't true if $x = \perp$.

Our first axiom gives the algebraic relations of *cons*, *car* and *cdr*.

S1: $(\forall x\ y)(issexp\ x \wedge issexp\ y \supset ispair[x.y] \wedge a[x.y] = x \wedge d[x.y] = y)$

The definition of atoms and pairs:

S2: $(\forall x)((ispair\ x \equiv issexp\ x \wedge \neg atom\ x) \wedge (atom\ x \supset issexp\ x))$

Taking apart an S-expression and putting the parts back together gives back the original expression.

S3: $(\forall x)(ispair\ x \supset issexp\ a\ x \wedge issexp\ d\ x \wedge x = a\ x\ .\ d\ x)$

Lists are included among S-expressions.

S4: $(\forall u)(islist\ u \supset issexp\ u)$

*cons*ing an S-expression onto a list gives a list.

S5: $(\forall x\ u)(issexp\ x \wedge islist\ u \supset islist[x.u])$

NIL is the only atomic list and only NIL satisfies the predicate *null*.

S6: $(\forall u)((islist\ u \wedge atom\ u \equiv u = NIL) \wedge (null\ u \equiv u = NIL))$

The simple structural induction schema for S-expressions:

S7: $(\forall x)(atom\ x \supset \Phi\ x) \wedge (\forall x)(ispair\ x \wedge \Phi\ a\ x \wedge \Phi\ d\ x \supset \Phi\ x) \supset (\forall x)(issexp\ x \supset \Phi\ x)$

The simple structural induction schema for lists:

S8: $\Phi\ NIL \wedge (\forall u)(islist\ u \wedge \neg null\ u \wedge \Phi\ d\ u \supset \Phi\ u) \supset (\forall u)(islist\ u \supset \Phi\ u)$

$x \leq_S y$ means that x is a subexpression of y and is a well-founded partial ordering. It is important for course-of-values induction for S-expressions.

S9: $(\forall x\ y)(issexp\ x \wedge issexp\ y \supset x \leq_S y \equiv x = y \vee \neg atom\ y \wedge (x \leq_S a\ y \vee x \leq_S d\ y))$

Definition of proper subexpression:

S10: $(\forall x\ y)(x <_S y \equiv x \leq_S y \wedge x \neq y)$

The course-of-values structural induction schema for S-expressions:

S11: $(\forall x)(issexp\ x \wedge (\forall y)(issexp\ y \wedge y <_S x \supset \Phi\ y) \supset \Phi\ x) \supset (\forall x)(issexp\ x \supset \Phi\ x)$

$u \leq_L v$ is the natural well-founded partial ordering for lists. It can be read "The list u is a tail of the list v".

S12: $(\forall u\ v)(islist\ u \wedge islist\ v \supset u \leq_L v \equiv u = v \vee \neg null\ v \wedge u \leq_L d\ v)$

u is a proper tail of v.

S13: $(\forall u\ v)(u <_L v \equiv u \leq_L v \wedge u \neq v)$

The course-of-values induction schema for lists. Course-of-values induction schemata are all the same except for the ordering used.

S14: $(\forall u)(islist\ u \wedge (\forall v)(islist\ v \wedge v <_L u \supset \Phi\ v) \supset \Phi\ u) \supset (\forall u)(islist\ u \supset \Phi\ u)$

These axioms for integers are based on the successor and predecessor functions and are analogous to the above axioms for S-expressions. They are equivalent to the usual first order number theory.

The relation between the successor and predecessor functions:

I1: $(\forall n)(isint\ n \supset isint\ succ\ n \wedge succ\ n \neq 0 \wedge pred\ succ\ n = n)$

As a function in the logic, the predecessor must always have a value. However we say something about *pred n* only for non-zero n.

I2: $(\forall n)(isint\ n \wedge n \neq 0 \supset isint\ pred\ n \wedge succ\ pred\ n = n)$

The simple induction schema for integers:

I3: $(\Phi\ 0 \wedge (\forall n)(isint\ n \wedge n \neq 0 \wedge \Phi\ pred\ n \supset \Phi\ n) \supset (\forall\ n)(isint\ n \supset \Phi\ n)$

For course-of-values induction, we need the ordering relations.

I4: $(\forall m\ n)(isint\ m \wedge isint\ n \supset (m \leq n \equiv m = n \vee n \neq 0 \wedge m \leq pred\ n))$

Proper ordering:

I5: $(\forall m\ n)(m < n \equiv m \leq n \wedge m \neq n)$

The course-of-values schema:

I6: $(\forall n)(isint\ n \wedge (\forall m)(isint\ m \wedge m < n \supset \Phi\ m) \supset \Phi\ n) \supset (\forall n)(isint\ n \supset \Phi\ n)$

The recursive definition of addition:

17: $(\forall m\ n)(isint\ m \wedge isint\ n \supset m + n = IF\ n = 0\ THEN\ m\ ELSE\ succ\ m + pred\ n)$

Multiplication:

18: $(\forall m\ n)(isint\ m \wedge isint\ n \supset m \times n = IF\ n = 0\ THEN\ 0\ ELSE\ m + m \times pred\ n)$

The next group of axioms are concerned with extending the domain by adjoining $\perp$. The predicates of the extended domains are *isesexp*, *iselist* and *iseint* respectively.

Extending the S-expressions with $\perp$:

E1: $(\forall x)(isesexp\ x = issexp\ x \vee x = \perp)$

Extending the lists with $\perp$:

E2: $(\forall u)(iselist\ u = islist\ u \vee u = \perp)$

Extending the integers with $\perp$:

E3: $(\forall n)(iseint\ n = isint\ n \vee n = \perp)$

We need a function taking the value T when its argument is an S-expression. It will be used in extended conditional expressions.

E4: $(\forall x)(issexpf\ x = IF\ x = \perp\ THEN\ \perp\ ELSE\ IF\ issexp\ x\ THEN\ T\ ELSE\ F)$

Likewise for lists:

E5: $(\forall u)(islistf\ u = IF\ u = \perp\ THEN\ \perp\ ELSE\ IF\ islist\ x\ THEN\ T\ ELSE\ F)$

Likewise for integers:

E6: $(\forall n)(isintf\ n = IF\ n = \perp\ THEN\ \perp\ ELSE\ IF\ isint\ x\ THEN\ T\ ELSE\ F)$

Extending the integer functions to take $\perp$ as an argument. The extension is *strict*, i.e. the extended values are all $\perp$.

E7: $succ\ \perp = \perp \wedge pred\ \perp = \perp$

Extending the Lisp functions strictly to take $\perp$ as an argument:

E8: $a\ \perp = \perp \wedge d\ \perp = \perp$

The strict extension of *cons*. (Friedman and Wise (1977) propose a non-strict extension).

E9: $(\forall x)(x.\perp = \perp \wedge \perp.x = \perp)$

The functions at and n are defined from the predicates *atom* and *null*.

E10: $(\forall x)(at\ x = IF\ x = \perp\ THEN\ \perp\ ELSE\ IF\ atom\ x\ THEN\ T\ ELSE\ F)$

E11: $(\forall u)(n\ u = IF\ u = \perp\ THEN\ \perp\ ELSE\ IF\ null\ u\ THEN\ T\ ELSE\ F)$

7. Forms of Induction.

All proofs of non-trivial program properties require some form of mathematical induction. Methods of induction can be divided into three classes - induction on data, various forms of computation induction on approximations to the program, and induction on the course of the computation. It is not certain that that these are really distinct; i.e. there may be systematic ways of regarding one as a form of another. In this section, we deal only with induction on data.

Induction on data often takes a form called structural induction in which the data domain consists of objects built up from elementary objects by a fixed finite set of operations. The construction of S-expressions from atoms by *cons* or the construction of the integers from zero by the successor operation are examples.

Induction can take two forms. One form involves the constructors or selectors of the domain directly. Simple list, S-expression, and numerical induction are examples. The second form is a course-of-values induction schema

27) $(\forall x)(isD\ x \wedge (\forall y)(isD\ y \wedge y < x \supset \Phi\ y) \supset \Phi\ x) \supset (\forall x)(isD\ x \supset \Phi\ x)$

based on an ordering relation < defined in terms of the selector functions. Course-of-values schemata were also given for lists, S-expression and natural numbers. Course-of-values often gives a proof with a simpler induction predicate than simple induction.

A simple example is the termination of the list function *alt* defined by

28) *alt u* ← if n *u or* n d *u* then *u* else a *u* . *alt* dd *u*.

Because of the dd, simple induction doesn't work on the obvious predicate

29) $\Phi(u) = islist\ alt\ u$,

but course-of-values induction does work.

In the simple cases we have seen so far, the induction is on one of the variables in the program, but this is not the general case. More generally, the induction is on some function of the

variables, and the domain of this function may be quite different from that of the variables of the progam. Often it can be taken to be the natural numbers, but more generally it can be any partially ordered domain in which all descending chains are finite.

For example S-expression can be replaced by induction on natural numbers by introducing the function *size x* defined by

30) $size\ x \leftarrow$ if *at* x then 1 else $size\ a\ x + size\ d\ x$

Size has the property that $size\ a\ x < size\ x$ and $size\ d\ x < size\ x$. We can prove that a formula $\Phi(x)$ holds for all S-expressions by "induction on the size of x". This is done by proving that the formula Φ' given by

31) $\Phi'(n) \equiv (\forall x)(size\ x = n \supset \Phi(x))$

holds for all numbers using numerical induction. In fact any proof of the formula Φ by S-expression induction can easily be converted to a proof of Φ' by numerical induction and vice versa.

A more exotic example of this is provided by the Takeuchi function (Takeuchi 1978) defined by

32) $tak(m, n, p) \leftarrow$
if m *lesseq* n then n else $tak(tak(m-1, n, p), tak(n-1, p, m), tak(p-1, m, n))$.

The function is total when the arguments are integers and is equal to

33) $tak0(m1, m2, m3) = IF\ m1 \leq m2\ THEN\ m2\ ELSE\ IF\ m2 \leq m3\ THEN\ m3\ ELSE\ m1.$

The most convenient proof that *tak* is total uses the course-of-values schema for integers with

34) $\Phi(n) \equiv (\forall m1\ m2\ m3)(rank(m1, m2, m3) = n \supset tak(m1, m2, m3) = tak0(m1, m2, m3))$,

where

35) $rank(m1, m2, m3) \equiv dtak1(m1-m2, m3-m2)$,

and

36) $$\begin{aligned} dtak1(n1, n2) = {} & IF\ n1 \leq 0\ THEN\ 0 \\ & ELSE\ IF\ n2 \geq 2\ THEN\ m + n(n-1)/2 - 1 \\ & ELSE\ IF\ n \geq 0\ THEN\ m \\ & ELSE\ IF\ n = -1\ THEN\ (m+1)(m+2)/2 - 1 \\ & ELSE\ (m-n)(m-n+1)/2 - m - 1. \end{aligned}$$

This is an example of the more general form of inductive proof. A rank function is defined taking values in some inductively ordered domain (in this case the natural numbers), and the

theorem is proved under the hypothesis that it is true for all lower rank tuples of variables. The term *structural induction* seems no longer applicable to this general case, because it is not an induction on the structure of the data domain of the program, although it requires no new machinery when we are operating within first order logic. Perhaps *structural induction* was a misnomer anyway, since the more general form corresponds to how mathematicians already looked at induction.

The inductively ordered set serving as the domain of the rank function is chosen for convenience, where the object is to get a short and understandable proof. If we only care about whether a proof exists and not how easy it is to write and read, then all the domains considered so far are equivalent to the natural numbers. To get something stronger, we go to induction over transfinite ordinal numbers - explained in most books on axiomatic set theory.

The axiom schema for induction over ordinals is just the usual course-of-values schema written with the ordering over the ordinals, say $\leq_0$. In order to use it, this ordering must be defined, and we must be able to write a rank function from tuplets to ordinals. This requires that we use a notation for ordinals, and any given notation represents only the ordinals less than some bound. Most proofs arising in practice will involve only ordinals less than ω^ω which can be represented as polynomials in ω.

An example requiring induction up to ω^2 is proving the termination of Ackermann's function which has the functional equation

37) $(\forall m\ n)(A(m, n) =$
if m *equal* 0 then $n+1$ else if n *equal* 0 then $A(m-1, 0)$ else $A(m-1, A(m, n-1)))$.

The statement to be proved is

38) $(\forall\alpha)(\alpha < \omega^2 \supset \Phi(\alpha))$,

where

39) $(\forall\alpha)(\Phi(\alpha) \equiv (\forall m\ n)(rank(m, n) = \alpha \supset isint\ A(m, n)))$,

and

40) $(\forall m\ n)(rank(m, n) = \omega m + n)$.

The proof is straightforward, because $\omega(m-1) < \omega m+n$ and $\omega m+(n-1) < \omega m+n$, so we can assume *isint* $A(m-1, 0)$ and *isint* $A(m, n-1)$. From the latter, it follws that $\omega(m-1)+A(m, n-1) < \omega m+n$ which completes the induction step.

8. An Extended Example.

The SAMEFRINGE problem is to write a program that efficiently determines whether two S-expressions have the same fringe, i.e. have the same atoms in the same order. (Some people omit the NILs at the ends of lists, but we will take all atoms). Thus ((A.B).C) and (A.(B.C)) have the same fringe, namely (A B C). The object of the original problem was to program it using a minimum of storage, and it was conjectured that co-routines were necessary to do it neatly. We shall not discuss that matter here - merely the extensional correctness of one proposed solution.

The relevant recursive definitions are

41) *fringe x* ← if *at x* then <x> else *fringe* a *x* * *fringe* d *x*,

We are interested in the condition *fringe x* = *fringe y*.

The function to be proved correct is *samefringe*[*x*, *y*] defined by the simultaneous recursion

42) *samefringe*[*x*, *y*] ← (*x equal y*) *or* [*not at x and not at y and same*[*gopher x*, *gopher y*]],

43) *same*[*x*, *y*] ← (a *x equal* a *y*) *and samefringe*[d *x*, d *y*],

where

44) *gopher x* ← if *at* a *x* then *x* else *gopher* aa *x* . [da *x* . d *x*].

We need to prove that *samefringe* is total and

45) (∀*xy*)(*samefringe*[*x*, *y*] = *T* ≡ *fringe x* = *fringe y*).

The functional equations are

46) (∀*x*)(*fringe x* = if *at x* then <x> else *fringe* a *x* * *fringe* d *x*),

47) (∀*u v*)(*u* * *v* = if n *u* then *v* else a *u* . [d *u* * *v*]).

48) (∀*x y*)(*samefringe*[*x*, *y*] =
x equal y or [*not aat x and not aat y and same*[*gopher x*, *gopher y*]]),

49) (∀*x y*)(*same*[*x*, *y*] = a *x equal* a *y and samefringe*[d *x*, d *y*],

50) (∀*x*)(*gopher x* = if *at* a *x* then *u* else *gopher* aa *x* . [da *x* . d *x*]).

We shall not give full proofs but merely the induction predicates and a few indications of the algebraic transformations. We begin with a lemma about *gopher*.

51) (∀*x y*)(*ispair gopher*[*x.y*] ∧ *atom* a *gopher*[*x.y*] ∧ *fringe gopher*[*x.y*] = *fringe*[*x.y*]).

This lemma can be proved by S-expression structural induction on x using the predicate

52) $\Phi(x) \equiv (\forall y)(ispair\ gopher[x.y] \wedge atom\ a\ gopher[x.y] \wedge fringe\ gopher[x.y] = fringe[x.y])$.

In the course of the proof, we use the associativity of $*$ and the formula $fringe[x.y] = fringe\ x * fringe\ y$. The lemma was expressed using $gopher[x.y]$ in order to avoid considering atomic arguments for *gopher*, but it could have equally well be proved about $gopher\ x$ with the condition $\neg atom\ x$.

For our proof about *samefringe* we need one more lemma about *gopher*, namely

53) $(\forall x\ y)(size\ gopher[x.y] = size[x.y])$.

This can be proved by S-expression induction on x separately or as a part of the above lemma by including $size\ gopher[x.y] = size[x.y]$ as a conjunct in (51) and (52).

The statement about *samefringe* is

54) $(\forall x\ y)(issexp\ samefringe[x,y] \wedge samefringe[x,y] = T \equiv fringe\ x = fringe\ y)$,

and it is most easily proved by induction on $size\ x + size\ y$ using the predicate

55) $\Phi(n) \equiv (\forall x\ y)(n = size\ x + size\ y \supset$
$issexp\ samefringe[x,y] \wedge (samefringe[x,y] = T \equiv fringe\ x = fringe\ y))$.

It can also be proved using the well-foundedness of lexicographic ordering of the list $<x,\ a\ x>$, but then we must decide what lexicographic orderings to include in our axiom system.

Transfinite induction is also useful, and can be illustrated with a variant *samefringe* that does everything in one complicated recursive definition, namely

```
56)  samefringe[x, y] ←
         (x equal y) or
         not at x and not at y and
             if at a x then [if at a y then a x equal a y and samefringe[d x, d y]
                                 else samefringe[x, aa y . [da y . d y]]]
             else samefringe[aa x . [da x . d x], y].
```

The transfinite induction predicate then has the form

57) $\Phi(n) \equiv (\forall x\ y)[n = \omega(size\ x + size\ y) + size\ a\ x + size\ a\ y \supset$
$issexp\ samefringe[x,y] \wedge (samefringe[x,y] = T \equiv fringe\ x = fringe\ y)]$.

We would like to prove that the amount of storage used in the computation of $samefringe[x,y]$ aside from that occupied by x and y, never exceeds the sum of the numbers of *cars* required to reach corresponding atoms in x and y. Unfortunately, we can't even express that

fact, because we are axiomatizing the programs as functions, and the amount of storage used does not depend merely on the function being computed; it depends on the method of computation. We may regard such things as *intensional* properties, but any correspondence with the notion of intensional properties in intensional logic remains to be established. Many such intensional properties of a program are extensional properties of certain "derived programs", and some are even extensional properties of the functional τ.

9. The Minimization Schema.

The functional equation of a program doesn't completely characterize it. For example, the program

58) $f1\ x \leftarrow f1\ x$

leads to the sentence

59) $(\forall x)(f1\ x = f1\ x)$

which provides no information although the function $f1$ is undefined for all x. This is not always the case, since the program

60) $f2\ x \leftarrow (f2\ x).\mathrm{NIL}$

has the functional equation

61) $(\forall x)(f2\ x = (f2\ x).\mathrm{NIL}).$

from which $(\forall x)\neg issexp\ f2(x)$ can be proved by induction.

In order to characterize recursive programs, we need some way of asking for the least defined solution of the functional equation.

Suppose the program is

62) $f(x, y) \leftarrow \tau[f](x, y)$

yielding the functional equation

63) $(\forall x\ y)(f(x, y) = \tau[f](x, y).$

The *minimization schema* is then

64) $(\forall x)(\tau[\phi](x) \sqsubseteq \phi(x)) \supset (\forall x)(f(x) \sqsubseteq \phi(x)).$

In the case of *Append* we have

65) $(\forall u\ v)(\phi(u, v) \supseteq \text{if } n\ u \text{ then } v \text{ else } a\ u\ .\ \phi(d\ u,\ v)) \supset (\forall u\ v)(\phi(u, v) \supseteq u*v)$.

In the schema ϕ is a free function variable of the appropriate number of arguments. The schema is just a translation into first order logic of Park's (1970) theorem.

66) $\phi \supseteq \tau[\phi] \supset \phi \supseteq Y[\tau]$.

Here Y is the least fixed point operator.

[Note that this theorem is a generalization to continuous functionals of the second part of Kleene's first rescursion theorem (Kleene 1952)].

The simplest application of the schema is to show that the $f1$ defined by (58) is never an S-expression. The schema becomes

67) $(\forall x)(\phi\ x \supseteq \phi\ x) \supset (\forall x)(\phi\ x \supseteq f1\ x)$,

and we take

68) $\phi\ x = \perp$.

The left side of (67) is identically true, and, remembering that $\perp$ is not an S-expression, the right side tells us that $f1\ x$ is never an S-expression.

The minimization schema can sometimes be used to show partial correctness. For example, the well known 91-function is defined by the recursive program over the integers

69) $f91\ x \leftarrow$ if x *greater* 100 then $x - 10$ else $f91\ f91(x + 11)$.

The goal is to show that

70) $(\forall x)(f91\ x = IF\ x > 100\ THEN\ x - 10\ ELSE\ 91)$.

We apply the minimization schema with

71) $\phi\ x \leftarrow$ if x *greater* 100 then $x - 10$ else 91,

and it can be shown by an explicit calculation without induction that the premiss of the schema is satisfied, and this shows that $f91$, whenever defined has the desired value.

The method of recursion induction (McCarthy 1963) is also an immediate application of the minimization schema. If we show that two functions satisfy the schema of a recursive program, we show that they both equal the function computed by the program on wherever the function is defined.

The utility of the minimization schema for proving partial correctness or non-termination

depends on our ability to name suitable comparison functions. f1 and f91 were easily treated, because the necessary comparison functions could be given explicitly without recursion. Any extension of the language that provides new tools for naming comparison functions, e.g. going to higher order logic, will improve our ability to use the schema in proofs.

10. Derived Programs and Complete Recursive Programs.

The methods considered so far in this paper concern *extensional* properties of programs, i.e. properties of the function computed by the program. The following are not extensional properties: the number of times a certain function is evaluated in executing the program including as a special case the number of recursions, the maximum depth of recursion, and the maximum amount of storage used. Some of these properties depend on whether the program is executed call-by-name or call-by-value, while others are extensional properties of the functional of the program.

Many of these *intensional* properties of a program are extensional properties of related programs called *derived programs*. For example, the number of *cons* operations done by *Append* can be computed by a program of the same recursive structure, namely

72) $ncappend[u, v] \leftarrow$ if $n\ u$ then 0 else $1 + ncappend[d\ u,\ v]$.

If we define *flat* by

73) $flat[x, u] \leftarrow$ if $at\ x$ then $x.u$ else $flat[a\ x, flat[d\ x\ u]]$,

then the number of recursions done by *flat* is given by

74) $nrflat[x, u] \leftarrow$ if $at\ x$ then 1 else $1 + nrflat[a\ x, flat[d\ x, u]] + nrflat[d\ x, u]$,

noticing that *nrflat* is mutually recursive with *flat* itself. The maximum depth of recursion of the 91-function is given by

75) $df91\ n \leftarrow 1 +$ if n *greater* 100 then 0 else $max(df91(n + 11), df91(f(x + 11)))$.

Morris (1968) discussed a derived function that gives successive approximations of bounded recursion depth to a recursive function by modifying the definition to take a "rationed" number of allowed recursions. For *append* we would have

76) $append1[n, u, v] \leftarrow$
if n *equal* 0 then $\perp$ else if $n\ u$ then v else $a\ u\ .\ append1[n - 1, d\ u, v]$.

Thus $append1[n, u, v]$ computes $u*v$ but with a ration of n recursions. If the computation would require more than n recursions, the value is $\perp$, i.e. is undefined.

We can give a general rule for the rationed recursion function. Suppose that τ is a program for the function $f(x, y)$.

P: $f(x, y) \leftarrow \tau[f](x, y)$

Then

$C(P)$: $g(n, x, y) \leftarrow \tau'[g](n, x, y)$

where

77) $\tau'[\phi] = (\lambda n\ x\ y)$(if n *equal* 0 then $\perp$ else $\tau[(\lambda\ x\ y)\phi(n-1, x, y)]\ (x, y)$)

is a program for the rationed recursion function $g(n, x, y)$. In this case, the functional for the derived function is expressed by a formula in the functional for the original function. This can't always be done.

We can use the rationed recursion function as an alternate to the *minimiztion schema* for completing the characterization of f_P. Namely we have

78) $(\forall x\ y)(isD\ f_P(x,y) \equiv (\exists n)(\text{isD}\ f_{C(P)}(x,y)))$,

and whether $f_{C(P)}(x, y)$ is defined for given arguments is determined by its functional equation, because $C(P)$ is what (Cartwright 1978) calls a complete recursive program.

A recursive program P is called complete if its functional τ_P has only one fixed point f_P. Since the minimization schema is used for distinguishing the least fixed point, it is redundant for complete programs. The idea of complete recursive program was first advanced in (Cartwright 1978) as an alternative to the minimization schema for completing the characterization of the function computed by a program. The idea was to compute the computation sequence of a program P with a related *complete recursive program* $C(P)$ and to show metamathematically that for any program

79) $(\forall x)(f(x) = last\ f_{C(P)}(x)$

where $f_{C(P)}$ is the function computed by $C(P)$, and *last* is a function giving the last element of a list - in this case the list of values of f arising in the computation. Since whether $C(P)$ terminates for given arguments follows from its functional equation, (79) allows us to establish this for P itself. The constructions of (Cartwright 1978) were somewhat involved and differed substantially according to whether the original program was executed call-by-name or call-by-value.

The derived programs that give the number of recursions are complete so that *nrflat* as defined above satisfies

80) $(\forall x\ u)(isint\ nrflat[x, u] \equiv issexp\ flat[x, u])$.

A program for the number of recursions done when a program is evaluated call-by-name can also be given. Thus the number of recursions done in evaluating *morris*[*m*, *n*] call-by-name is given by *cmorris*[*m*, 0, *n*, 0] where

81) $cmorris[m, cm, n, cn] \leftarrow$
$1 + cm +$ if m *equal* 0 then 0 else $cmorris1[m-1, 0, morris[m, n], cmorris[m, 0, n, cn]]$.

The idea is that the arguments *cm* and *cn* are the numbers of recursive calls involved in evaluating *m* and *n* respectively. *morris* and *cmorris* are again equi-terminating.

II. Proof Methods as Axiom Schemata

Representing recursive definitions in first order logic permits us to express some well known methods for proving partial correctness as axiom schemata of first order logic.

For example, suppose we want to prove that if the input x of a function f defined by

82) $f\,x \leftarrow$ if $p\,x$ then x else $f\,h\,x$

satisfies $\Phi(x)$, then if the function terminates, the output $f(x)$ will satisfy $\Psi(x, f(x))$. We appeal to the following *axiom schema of inductive assertions*:

83) $(\forall x)(\Phi(x) \supset q(x, x)) \wedge (\forall x\, y)(q(x, y) \supset$ if $p\,x$ then $\Psi(x, y)$ else $q(x, h\,y))$
$\supset (\forall x)(\Phi(x) \wedge isD\,f\,x \supset \Psi(x, f\,x))$

where $isD\,f\,x$ is the assertion that $f(x)$ is in the nominal range of the function definition, i.e. is an integer or an S-expression as the case may be, and asserts that the computation terminates. In order to use the schema, we must invent a suitable predicate $q(x, y)$, and this is precisely the method of inductive assertions. The schema is valid for all predicates Φ, Ψ, and q, and a similar schema can be written for any collection of mutually recursive definitions that is iterative.

The method of *subgoal induction* for recursive programs was introduced in (Manna and Pnueli 1970), but they didn't give it a name. Morris and Wegbreit (1977) name it, extend it somewhat, and apply it to Algol-like programs. Unlike *inductive assertions*, it isn't limited to iterative definitions. Thus, for the recursive program

84) $f_5\,x \leftarrow$ if $p\,x$ then $h\,x$ else $g1\,f_5\,g2\,x$,

where p is assumed total, we have

85) $(\forall x)(p\,x \supset q(x, h\,x)) \wedge (\forall x\, z)(\neg p(x) \wedge q(g2\,x, z) \supset q(x, g1\,z)) \wedge (\forall x)(\Phi(x) \wedge q(x, z) \supset \Psi(x, z))$
$\supset (\forall x)(\Phi(x) \wedge isD(f(x)) \supset \Psi(x, f(x)))$

We can express these methods as axiom schemata, because we have the predicate *isD* to express termination. The minimization schema itself can be proved by subgoal induction. We need only take $\Phi(x) \equiv$ true and $\Psi(x, y) \equiv (y = \phi(x))$ and $q(x, y) \equiv (y = \phi(x))$.

General rules for going from a recursive program to what amounts to the subgoal induction schema are given in (Manna and Pnueli 1970) and (Morris and Wegbreit 1977); we need only add a conclusion involving the *isD* predicate to the Manna's and Pnueli formula W_P.

However, we can characterize subgoal induction as an axiom schema. Namely, we define $\tau'[q]$ as an extension of τ mapping relations into relations. Thus if

86) $\tau[f](x) = \text{if } p\ x \text{ then } h\ x \text{ else } g1\ f\ g2\ x,$

we have

87) $\tau'[q](x, y) \equiv \text{if } p\ x \text{ then } (y = h\ x) \text{ else } \exists z.(q(g2\ x, z) \wedge y = g1\ z).$

In general we have

88) $(\forall xy)(\tau'[q](x, y) \supset q(x, y)) \supset (\forall x)(isD\ f\ x \supset q(x, f\ x)),$

from which the subgoal induction rule follows immediately given the properties of Φ and Ψ. I am indebted to Wolfgang Polak (oral communication) for help in elucidating this relationship.

WARNING: The rest of this section is somewhat conjectural. There may be bugs.

The *extension* $\tau'[q]$ can be determined as follows: Introduce into the logic the notion of a *multi-term* which is formed in the same way as a term but allows relations written as functions. For the present we won't interpret them but merely give rules for introducing them and subsequently eliminating them again to get an ordinary formula. Thus we will write $q\langle e\rangle$ where e is any term or multi-term. We then form $\tau'[q]$ exactly in the same way $\tau[f]$ was formed. Thus for the 91-function we have

89) $\tau'[q](x) \equiv \text{if } x>100 \text{ then } x-10 \text{ else } q\langle q\langle x+11\rangle\rangle.$

The pointy brackets indicate that we are "applying" a relation. We now evaluate $\tau'[q](x, y)$ formally as follows:

90) $\tau'[q](x, y)$
$\equiv (\text{if } x>100 \text{ then } x-10 \text{ else } q\langle q\langle x+11\rangle\rangle)(y)$
$\equiv \text{if } x>100 \text{ then } y = x-10 \text{ else } q(q\langle x+11\rangle, y)$
$\equiv \text{if } x>100 \text{ then } y = x-10 \text{ else } \exists z.(q(x+11, z) \wedge q(z, y)).$

This last formula has no pointy brackets and is just the formula that would be obtained by Manna and Pnueli or Morris and Wegbreit. The rules are as follows:

(i) $\tau'[q](x)$ is just like $\tau[f](x)$ except that q replaces f and takes its arguments in pointy brackets.

(ii) an ordinary term e applied to y becomes $y = e$.

(iii) $q\langle e\rangle(y)$ becomes $q(e, y)$.

(iv) $P(q\langle e\rangle)$ becomes $\exists z.q(e, z) \wedge P(z)$ when $P(q\langle e\rangle)$ occurs positively in $\tau'[q](x, y)$ and becomes $\forall z.q(e, z) \supset P(z)$ when the occurrence is negatve. It is not evident whether an independent semantics can be given to multi-terms.

12. Representations Using Finite Approximations.

Our second approach to representing recursive programs by first order formulas goes back to Gödel (1931, 1934) who showed that primitive recursive functions could be so represented. (Our knowledge of Gödel's work comes from (Kleene 1952)).

Kleene (1952) calls a partial function f *representable* if there is an arithmetic formula A with free variables x and y such that

91) $(\forall x\ y)(y = f(x)) \equiv A)$,

where an arithmetic formula is built up from integer constants and variables using only addition, multiplication and bounded quantification. Kleene showed that all partial recursive functions are representable. The proof involves Gödel numbering possible computation sequences and showing that the relation between sequences and their elements and the steps of the computation are all *representable*.

In Lisp less machinery is needed, because sequences are Lisp data, and the relation between a sequence and its elements is given by basic Lisp functions and by the $\leq_L$ axiomatized in section 6 by

92) $(\forall u\ v)(u \leq_L v \equiv (u = v) \vee \neg null\ v \wedge u \leq_L d\ v)$.

Starting with $\leq_L$ and the basic Lisp functions and predicates we will define other Lisp predicates without recursion.

First we define the well known Lisp function *assoc* whose usual recursive definition is

93) $assoc[x, w] \leftarrow$ if n w then NIL else if x *equal* aa w then a w else $assoc[x,$ d $w]$

or non-recursively

94) $(\forall y)(y = assoc[x, w] \equiv (\forall u)(u \leq_L w \supset aa\ u \neq x) \wedge y = NIL$
$\vee (\exists u)(u \leq_L w \wedge x = aa\ u \wedge y = a\ u$
$\wedge (\forall v)(v \leq_L w \wedge u <_L v \supset aa\ v \neq x))$

Now suppose that

95) $f\,x \leftarrow \tau[f](x)$

is a recursive program, i.e. τ is a continuous functional. Our non-recursive definition of f uses finite approximations to f, i.e. lists of pairs of $(x\ .\ f(x))$, where each pair can be computed from the functional τ using only the pairs that follow it on the list. Thus we define

96) $ok[\tau](w) \leftarrow$
$\quad$ n w *or*
$\quad$ da w = $\tau[(\lambda x)$(if n *assoc*[x, d w] then $\perp$ else d *assoc*[x, d w])](aa w) *and* $ok[\tau]$(d w),

or non-recursively

97) $(\forall w)ok[\tau](w) \equiv$
$\quad (\forall u)u \leq_L w \supset$
$\quad\quad$ [*null* $u \vee$ da u = $\tau[(\lambda x)$(if n *assoc*[x, d u] then $\perp$ else d *assoc*[x, d u])](aa u)]])

Now we can define $y = f(x)$ in terms of the existence of a suitable w, namely

98) $(\forall x\ y)(y = f(x) \equiv$
$\quad (\exists w)ok[\tau](w) \wedge y = \tau[(\lambda x)$(if n *assoc*[$x$, w] then $\perp$ else d *assoc*[x, w])](x)))

It might be asked whether $\leq_L$ is necessary. Couldn't we represent recursive programs using just *car*, *cdr*, *cons* and *atom*? No, for the following reason. Suppose that the function f is representable using only the basic Lisp functions without $\leq_L$, and consider the sentence

99) $(\forall x)$(*issexp* $f(x)$),

asserting the totality of f. Using the representation, we can write (99) as a sentence involving only the basic Lisp functions and the constant $\perp$. However, Oppen (1978) has shown that these sentences are decideable, and totality isn't.

In case of functions of several variables, (98) corresponds to a call-by-value computation rule while the representations of the previous sections correspond to call-by-name or other "safe" rules. Treating call-by-name similarly requires a definition of *ok* in which some of the tuplets have some missing elements.

Note: Our original intention was to take $\leq_S$ as basic, but curiously, we have not succeeded in defining $\leq_L$ non-recursively in terms of $\leq_S$, although the converse is a consequence of our general construction.

13. Questions of Incompleteness.

Luckham, Park and Paterson (1970) have shown that whether a program schema diverges for every interpretation, whether it diverges for some interpretation, and whether two program schemas are equivalent are all not even partially solvable problems. Manna (1974) has a thorough discussion of these points. In view of these results, what can be expected from our first order representations?

First let us construct a Lisp computation that does not terminate, but whose non-termination cannot be proved from the axioms Lisp1 within first order logic. We need only program a proof-checker for first order logic, set it to generate all possible proofs starting with the axioms Lisp1, and stop when it finds a proof of (NIL ≠ NIL) or some other contradiction. Assuming the axioms are consistent, the program will never find such a proof and will never stop. In fact, proving that the program will never stop is precisely proving that the axioms are consistent. But Gödel's theorem asserts that axiom systems like Lisp1 cannot be proved consistent within themselves. Until recently, all the known cases of sentences of Peano arithmetic unprovable within Peano arithmetic involved such an appeal to Gödel's theorem or similar unsolvability arguments. However, Paris and Harrington (1977) found a form of Ramsey's theorem a well-known combinatorial theorem, that could be proved unprovable in Peano arithmetic. However, their proof of its unprovability involved showing that it implied the consistency of Peano arithmetic.

We can presumably prove Lisp1 consistent just as Gentzen proved arithmetic consistent - by introducing a new axiom schema that allows induction up to the transfinite ordinal ϵ_0. Proving the new system consistent would require induction up to a still higher ordinal, etc.

Since every recursively defined function can be defined explicitly, any sentence involving such functions can be replaced by an equivalent sentence involving only $\leq_L$ and the basic Lisp functions. The theory of $\leq_L$ and these functions has a standard model, the usual S-expressions and many non-standard models. We "construct" non-standard models in the usual way by appealing to the theorem that if every finite subset of a set S of sentences of first order logic has a model, then S has a model. For example, take S = {NIL $\leq_L$ x, (A) $\leq_L$ x, (A A) $\leq_L$ x, , ... indefinitely}. Every finite subset of S has a model; we need only take x to be the longest list of A's occurring in the sentences. Hence there is a model of the Lisp axioms in which x has all lists of A's as subexpressions. No sentence true in the standard model and false in such a model can be proved from the axioms. However, it is necessary to be careful about the meaning of termination of a function. In fact, taking successive *cdr*s of such an x would never terminate, but the sentence whose *standard interpretation* is termination of the computation is provable from Lisp1.

The practical question is: where does the correctness of ordinary programs come in? It seems likely that such statements will be provable with our original system of axioms. It doesn't follow that the system Lisp1 will permit convenient proofs, but probably it will. Some heuristic evidence for this comes from (Cohen 1965). Cohen presents two systems of axiomatized arithmetic Z1 and Z2. Z1 is ordinary Peano arithmetic with an axiom schema of induction, and Z2 is an axiomatization of hereditarily finite sets of integers. Superficially, Z2 is more powerful than Z1, but because the set operations of Z2 can be represented in Z1 as functions on the Gödel numbers

of the sets, it turns out that Z1 is just as powerful once the necessary machinery has been established. Because sets and lists are the basic data of Lisp1, and sets are easily represented, the power of Lisp1 will be approximately that of Z2, and convenient proofs of correctness statements should be possible. Moreover, since Lisp1 is a first order theory, it is easily extended with axioms for sets, and this should help make informal proofs easy to express.

A PUB source of this paper is available on disk at the Stanford Artificial Intelligence Laboratory with the file name FIRST[W79,JMC].

14. References.

Bochvar, D.A. (1938): "On a three-valued logical calculus and its application to the analysis of contradictions", *Recueil Mathematique*, N.S. 4 pp. 287-308.

Bochvar, D.A. (1943): "On the consistency of a three-valued logical calculus", *Recueil Mathematique*, N.S. 12, pp. 353-369.

The above Russian language papers by Bochvar are available in English translation as

Bochvar, D.A. (1972): *Two papers on partial predicate calculus*, Stanford Artificial Intelligence Memo 165, Computer Science Department, Stanford University, Stanford, CA 94305. (Also available from NTIS).

Cartwright, R.S. (1976): *A Practical Formal Semantic Definition and Verification System for Typed Lisp*, Ph.D. Thesis, Computer Science Department, Stanford University, Stanford, California.

Cartwright, R. (1978): *First Order Semantics: A Natural Programming Logic for Recursively Defined Functions*, Cornell University Computer Science Department Technical Report TR78-339, Ithaca, New York.

Cartwright, Robert and John McCarthy (1979): "First Order Programming Logic", paper presented at the sixth annual ACM Symposium on Principles of Programming Languages (POPL), San Antonio, Texas. Available from ACM.

Cohen, Paul (1966): *Set Theory and the Continuum Hypothesis*, W.A. Benjamin Inc.

Cooper, D.C. (1969): "Program Scheme Equivalences and Second-order Logic", in B. Meltzer and D. Michie (eds.), *Machine Intelligence*, Vol. 4, pp. 3-15, Edinburgh University Press, Edinburgh.

Friedman, Daniel and David Wise(1976): "Cons should not Evaluate Its Arguments", in *Proc. 3rd Intl. Colloq. on Automata, Languages and Programming*, Edinburgh Univ. Press, Edinburgh.

Hitchcock, P. and D. Park (1973): "Induction Rules and Proofs of Program Termination, in M. Nivat (ed.), *Automata, Languages and Programming*, pp. 225-251, North-Holland, Amsterdam.

Kleene, S.C. (1952): *Introduction to Metamathematics*, Van Nostrand, New York.

Luckham, D.C., D.M.R.Park, and M.S. Paterson (1970): "On Formalized Computer Programs", *J. CSS*, 4(3): 220-249 (June).

Manna, Zohar and Amir Pnueli (1970): "Formalization of the Properties of Functional Programs", *J. ACM*, 17(3): 555-569.

Manna, Zohar (1974): *Mathematical Theory of Computation*, McGraw-Hill.

Manna, Zohar, Stephen Ness and Jean Vuillemin (1973): "Inductive Methods for Proving Properties of Programs", *Comm. ACM*,16(8): 491-502 (August).

McCarthy, John (1963): "A Basis for a Mathematical Theory of Computation", in P. Braffort and D. Hirschberg (eds.), *Computer Programming and Formal Systems*, pp. 33-70. North-Holland Publishing Company, Amsterdam.

McCarthy, John (1964): *Predicate Calculus with "Undefined" as a Truth Value*, Stanford Artificial Intelligence Memo 1, Computer Science Department, Stanford University.

McCarthy, John (1978): "Representation of Recursive Programs in First Order Logic", in E.K. Blum and S. Takasu (eds.) *Proceedings of The International Conference on Mathematical Studies of Information Processing*, Kyoto. (a preliminary and superseded version of this paper)

McCarthy, John and Carolyn Talcott (1979): *LISP: Programming and Proving*, (in preparation)

Morris, James H.(1968): *Lambda Calculus Models of Programming Languages*. Ph.D. Thesis, M.I.T., Cambridge, Mass.

Morris, James H., and Ben Wegbreit (1977): "Program Verification by Subgoal Induction", *Comm. ACM*,20(4): 209-222 (April).

Oppen, Derek (1978): *Reasoning about Recursively Defined Data Structures*, Stanford Artificial Intelligence Memo 314, Computer Science Department, Stanford University.

Paris, Jeff and Leo Harrington (1977): "A Mathematical Incompleteness in Peano Arithmetic", in Jon Barwise (ed.), *Handbook of Mathematical Logic*, pp. 1133-1142. North-Holland Publishing Company, Amsterdam.

Park, David (1970): Fixpoint Induction and Proofs of Program Properties", in *Machine Intelligence* 5, pp. 59-78, Edinburgh University Press, Edinburgh.

Scott, Dana (1970): *Outline of a Mathematical Theory of Computation*. Programming Research Group Monograph No. 2, Oxford.

Takeuchi, I. (1978): Personal Communication.

Vuillemin, J. (1973): *Proof Techniques for Recursive Programs*. Ph.D. Thesis, Stanford University, Stanford, Calif.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

(14) 1. REPORT NUMBER: STAN-CS-79-717, AIM-324

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

(6) 4. TITLE (and Subtitle): Recursive Programs as Functions in a First Order Theory.

(9) 5. TYPE OF REPORT & PERIOD COVERED: technical rept.,

6. PERFORMING ORG. REPORT NUMBER: AIM-324

(10) 7. AUTHOR(s): Robert Cartwright, John McCarthy

(15) 8. CONTRACT OR GRANT NUMBER(s): MDA903-76-C-0206, NSF-MCS 78-00524 and MCS 78-05850

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Artificial Intelligence Laboratory
Stanford University
Stanford, California 94305

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
ARPA Order 2494

11. CONTROLLING OFFICE NAME AND ADDRESS
Eugene Stubbs
ARPA/PM
1400 Wilson Blvd., Arlington, VA 22209

(12) 34p. (11)

12. REPORT DATE: Mar[redacted]79

13. NUMBER OF PAGES: 32

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)
Philip Surra, ONR Representative
Durand Aeronautics Building, Rm 165
Stanford University
Stanford, CA 94305

15. SECURITY CLASS. (of this report): 15

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Releasable without limitation on dissemination

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT

Pure Lisp style recursive function programs are represented in a new way by sentences and schemata of first order logic. This permits easy and natural proofs of extensional properties of such programs by methods that generalize structural induction. It also systematizes known methods such as *recursion induction*, *subgoal induction*, *inductive assertions* by interpreting them as first order axiom schemata. We discuss the metatheorems justifying the representation and techniques for proving facts about specific programs. We also give a simpler version of the Goedel-Kleene way of representing computable functions by first order sentences.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED 094 120

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)