Courant Computer Science Report # 12

September 1977

Correct-Program Technology/ Extensibility of Verifiers

Two Papers on Program Verification

Martin Davis and J.T. Schwartz

Courant Institute of
Mathematical Sciences

Computer Science Department

New York University

Report No. NSO-12 prepared under Grants No.
NSF-MCS76-24212, NSF-MCS71-02039, and NSF-MCS76-00116
from the National Science Foundation; and
U.S. Energy Research and Development Administration
Contract No. EY-76-C-02-3077*000.

COURANT COMPUTER SCIENCE PUBLICATIONS

Price

COURANT COMPUTER SCIENCE NOTES

Programming Languages and Their Compilers, J. Cocke and J. T. Schwartz, 2nd Revised Version, *April 1970, iii+767 pp.* — $23.00

On Programming: An Interim Report on the SETL Project. J. T. Schwartz, Revised *June 1975, xii+675 pp.* — 20.50

A SETLB Primer. H. Mullish and M. Goldstein, *1973, v+201 pp.* — 6.25

Combinatorial Algorithms. E. G. Whitehead, Jr., *1973, vi+104 pp.* — 3.25

COURANT COMPUTER SCIENCE REPORTS

No. 1 ASL: A Proposed Variant of SETL
Henry Warren, Jr., *1973, 326 pp.*

No. 2 A Metalanguage for Expressing Grammatical Restrictions in Nodal Spans Parsing of Natural Language.
Jerry R. Hobbs, *1974, 266 pp.*

No. 3 Type Determination for Very High Level Languages
Aaron M. Tenenbaum, *1974, 171 pp.*

No. 4 A Comprehensive Survey of Parsing Algorithms for Programming Languages. Phillip Owens. *Forthcoming.*

No. 5 Investigations in the Theory of Descriptive Complexity.
William L. Gewirtz, *1974, 60 pp.*

No. 6 Operating System Specification Using Very High Level Dictions.
Peter Markstein, *1975, 152 pp.*

No. 7 Directions in Artificial Intelligence: Natural Language Processing. Ed. Ralph Grishman, *1975, 107 pp.*

No. 8 A Survey of Syntactic Analysis Procedures for Natural Language.
Ralph Grishman, *1975, 94 pp.*

No. 9 Scene Analysis: A Survey, Carl Weiman, *1975, 62 pp.*

No. 10 A Hierarchical Technique for Mechanical Theorem Proving and Its Application to Programming Language Design.
Norman Rubin, *1976, 172 pp.*

No. 11 Making Computational Sense of Montague's Intensional Logic.
Jerry R. Hobbs and Stanley J. Rosenschein, *1977, 41 pp.*

No. 12 Correct-Program Technology/Extensibility of Verifiers.
Two Papers on Program Verification. Martin Davis and J. T. Schwartz, with an Appendix by E. Deak, *1977, 146 pp.*

A catalog of SETL Newsletters and other SETL-related material is also available. Courant Computer Science Reports are available upon request. Prepayment is required for all Courant Computer Science Notes. Please address all communications to

COURANT INSTITUTE OF MATHEMATICAL SCIENCES
251 Mercer Street
New York, New York 10012

COURANT INSTITUTE OF MATHEMATICAL SCIENCES

Computer Science NSO-12

CORRECT-PROGRAM TECHNOLOGY/EXTENSIBILITY OF VERIFIERS

Two Papers on Program Verification

Martin Davis and J. T. Schwartz

with Appendix by E. Deak

Report No. NSO-12 prepared under Grants No. NSF-MCS76-24212, NSF-MCS71-02039, and NSF-MCS76-00116 from the National Science Foundation; and U. S. Energy Research and Development Administration Contract No. EY-76-C-02-3077*000.

TABLE OF CONTENTS

		Page
On Correct-Program Technology	J. Schwartz.....	1
Appendix	E. Deak.....	110
Metamathematical Extensibility for Theorem Verifiers and Proof Checkers	M. Davis & J. Schwartz...	120

On Correct Program Technology

Jack Schwartz

1. Introduction

By now the potential importance of formal techniques for proving program correctness has been recognized for over two decades, and something between four and seven hundred papers principally devoted to the development of such techniques have been published (see [London, 1970], [1972] for useful reviews of existing literature). A wide variety of formalisms have been proposed; among these, the inductive assertion technique of [Floyd, 1967], and the variant of it proposed and repeatedly extended by Hoare [1969], [1971] [1972] seem most convenient. But in spite of all this work, a truly practical program verification technology has been slow to develop. The essential difficulty not yet overcome is economic: the cost of formal program verification using existing teachniques is still much too high. This objection is best understood if we consider the way in which a fully formal variant of the Floyd technique would operate. In this approach, one is given a program text P, to which an input assumption I and an output assertion are attached. (These assertions are written in any convenient, sufficiently powerful, logical formalism, e.g., predicate calculus supplemented by Zermelo-Frankel set theory. In general, we shall use LF to denote whatever logical formalism is used.) Call a program text thus annotated (i.e., carrying attached assumptions and assertions) a praa. To prove a praa Q correct by the Floyd method, one begins by attaching

additional assertions to its text. Generally speaking, at least one such additional assertion needs to be attached to each loop (also each procedure) in the program text of the praa; in effect, this assumption captures and formalizes the fragment of technique which guided the programmer to write the loop. (As a matter of fact, recent work has shown that at least for some simple loops these assumptions can be supplied automatically, see [Wegbreit, 1974], [Morris and Wegbreit, 1976], [Misra, 1975]. While the techniques used to do this are interesting and ingenious, the approach to be developed in the present paper suggests that the possibility of proceeding in this way is not of primary importance.) The praa Q' developed by adding these additional assertions to Q is then processed by a *verification condition generator* (sometimes also called a *verifying compiler*, though the term *verification compiler* would be better), which by a straightforward process converts Q into a set S of statements of the logical formalism LF. Then to complete the verification one must prove all of the statements of the set S. In principle, these proofs must themselves be verified mechanically, i.e., each proof must be expressed in a fully formal way and certified by a proof verifer program PV able to recognize correct LF-proofs. It is at this stage of a verification that we must expect the greatest expenditure of labor to occur; although in principle the verifier program PV could supply part of the necessary proof itself, the present state and prospects of automatic proof technology make it appear likely that PV will have to be guided quite closely by the supply of numerous very detailed proof steps. (However, energetic development of proof-verifier technology can be expected to reduce the number of formal steps which must be supplied in typical proofs, and to allow these proofs to be supplied in intuitively comfortable forms.) A secondary difficulty is that the substitutions and other formal manipulations applied by a verification condition generator will typically

convert the assertions occuring in a praa, the intuitive content of which may already be somewhat obscure, into still murkier forms, adding to the burden and difficulty of supplying the necessary formal proofs.

Having thus briefly reviewed existing technique, let us now contrast the approach proposed in the present paper, in order to highlight the pragmatic differences that characterize our approach. In the first place, rather than starting with large unannotated program texts and attempting to prove their correctness, we shall prefer to work systematically with praas known to be correct, *stating rules for the manipulation and combination of correct praas*, using which new, necessarily correct praas can be derived. Thus our approach will conform and give formal realization to an insight of Dijkstra: that programs should not so much be proved correct as developed in such a way as to make their correctness evident. In this sense, the class of correct praas is to be compared to the set of universally valid formulae of the logical formalism which underlies them, and the rules for praa combination which we shall state correspond to the formal proof rules of substitution and deduction which allow new universally valid formulae to be deduced from old. In pragmatic terms, we may claim that the approach to be described gives formal expression to the generating steps by which programmers construct programs in the first place, and accordingly can object to the standard Floyd-Hoare approach by noting that, starting always from an externally given program text, it necessarily loses sight of a program's genetic origins. A central claim of the present paper is that by remedying this defect the task of deducing a praa's correctness can be much alleviated.

In our approach then, correct praas can be used to generate new correct praas, in much the same way as new correct algebraic or logical formulae can be gererated by combining old formulae. This suggests the following view of the programming process: that it amounts to a type of formal manipulation roughly analogous to algebraic computation, in which text fragments expressing elementary, essentially irreducible, correct algorithms are combined to yield correct programs adapted to one or to another intended application. (And developed to the point at which program efficiency becomes acceptable.) From this point of view, what has been lacking till now is simply a sufficiently full and formal statement of the rules for manipulation and composition of correct praas; here it should be noted that *these rules necessarily relate to correct praas and not simply to program texts*, which may perhaps help explain why full statement of them has been so long delayed. Observe then that, regarding programming as a type of formal manipulation akin to the algebraic manipulation of (possibly very large) formulae, we can understand why much but not all of it is experienced psychologically as having a routine flavor rather than smacking of the intensely creative. From this same point of view we may say that, just as the generation of reliably correct large algebraic formulae must rest on the use of a formula manipulation program, so the reliable generation of large correct praas must rest on the formal use of a programmed praa manipulator. In both cases, the probability of error becomes overwhelming if one tries to work manually with large texts. Extending this analogy, we may liken the informal procedures which characterize much of today's programming to what algebraic calculation might be if, without making systematic use of formalized rules, one

operated on the basis of informal reasoning and informed intuition; can then liken the process of structured programming advocated by Dijkstra to a still manual process of calculation guided by systematic reference to formal algebraic rules, and can liken the correct-praa technology which we shall describe to the use of a formula manipulation program.

The most elementary fragments to which our rules of modification and combination will usefully apply will be correct praas, generally rather small, each of which expresses some fundamental and essentially indecomposable element of algorithmic technique. Some of these 'root' praas can be very simple, e.g. those which describe the standard techniques for zeroing an array or adding up the components of a vector; others will be more complex, and will e.g. describe the 'treesort' technique or the techniques used to maintain B-trees; others may be relatively profound, and describe, e.g., the essential ideas of Tarjan high-speed technique for determining flow-graph reducibility. To establish the correctness of the 'root' praas corresponding to these algorithms, we have no choice but to use a variant of the general Floyd/Hoare program verification technique. (The formalism that we will use for this is rather close to that of Hoare, but in detail differs significantly from Hoare's formalism; these purely pragmatic differences adapt our formalism to our larger purposes, but also seem to make the proof formalism that we shall describe somewhat more comfortable to use than Hoare's technique, and perhaps also a mite more general.) That the full power of the logical formalism LF should have to be invoked to prove the correctness of each 'root' praa is after all not surprising, since each such praa expresses an essentially

new mathematical fact: in heuristic terms, we may say that on first meeting such a praa, we are bound to find the fact that it works surprising until we have been presented, at least informally, with a mathematical proof that it does. On the other hand, in our approach the correctness of 'composite' praas will follow in relatively routine fashion from the way in which they are built up. Thus a second pragmatic objection that we may raise to the Floyd/Hoare technique as it is ordinarly used is that *it applies, to composite programs, techniques that only need to be applied to elementary programs, thereby rending formal verification over-expensive.*

Concerning the logical formalism LF that we shall use and the level of programming language to be used along with it, the following remarks should be helpful. For expressing logical relationships and formal proofs, it is clearly most advantageous to use a maximally powerful logical language, which dictates the use of something like predicate calculus supplemented by Zermelo-Frankel set theory. Indeed, if one uses any significantly less powerful logical framework, then certain relationships which would have very direct expression in the more powerful language will be unstateable or at any rate stateable only in very crabbed and roundabout ways in the less powerful formalism; moreover, certain proofs which in the more powerful formalism would be short and direct will be rendered long and complex, and may even become impossible. If one mistakenly foreswears the use of set theory, this consideration applies, e.g., to any use of the 'box principle', i.e., that any assignment of n objects to fewer than n boxes must put at least two objects into some one box, as part of a correctness proof. But then, since it is desirable that no limitation inherent in the programming language we use should deprive us of any

expressive or proof-theoretic possibility which our underlying logical formalism would allow us, we shall also want to allow our programming language to manipulate perfectly general set theoretic objects. We shall do this with a vengeance, initially foreswearing all considerations not only of efficiency but even of implementability, and thus working with a formal programming language in which any set-theoretic object, whether finite, infinite, or even transfinite can be 'manipulated'. Note that since our concern is initially not to execute programs but simply to prove the correctness of praas, this manner of proceeding, which may at first sight seem counterintuitive, is not only harmless but desirable and necessary. However, this does not mean that our considerations will apply only to praas written in a language of some unrealistically high level. Quite to the contrary, the methods to be described will facilitate the proofs of correctness even of programs written in assembly language. After all, in developing such a proof one is concerned only to exhibit a minimum-length path of argument from a given set of axiomatic foundations to a final assembly-language program; the fact that transfinite modes of expression have been used along the way to shorten this path is as little harmful as the numerical analyst's use of properties of the transfinite set of real numbers is harmful to the efficiency of some entirely finite root-finding procedure which he can justify by reasoning about these properties. We note that in a very interesting paper [Wegbreit, 1976] has shown how program-verification techniques can be adapted to give formal proofs that algorithms are efficient; even in such proofs, the use of very high level and even potentially transfinite modes of expression can be advantageous, since (we emphasize once more) the efficiency of an algorithm and the implementability or efficiency of

the expressions used to study it are two entirely different things.

Since the programming language that we will use will allow arbitrary set-theoretic objects as data objects, it will be relatively easy for us to deal with any of the data structures commonly encountered in programming practice, e.g., arrays, structures, and even pointer systems which we will be able to represent explicitly using maps.

In the above connection it is worth noting that the present paper concentrates exclusively on the question of 'partial' correctness of praas. That is, we prove only that, as a praa runs, and given that all the assumptions in it are correct, it must follow that each assertion in it will be confirmed whenever control passes through the place in the path to which the assertion attaches. Note that if control never passes through a given place, then *any* assertion at that place is correct *a priori*. Since we concern ourselves only with partial correctness, neither loops which never terminate or operations with illegal operands will create any special problems for us. In particular, we can write iterations (∀x∈s) *code*; end ∀; over infinite sets freely.

The techniques of praa modification and combination central to the approach advocated in the present paper are very much anticipated in [Gerhart, 1975a], [1975b], [1976]. Significant technical differences between our treatment and that of Gerhart are as follows: (a) Basing herself on the Hoare formalism, Gerhart confines her attention to the somewhat special class of praas in which assumptions and assertions appear only at the heads of while loops, and focuses rather more strongly on the 'forward' and 'backward' Floyd verification conditions

than is entirely appropriate for the development of a smooth formalism. (b) Gerhart tends to make use of a logical formalism considerably narrower than the set-theoretic system which we assume. This does not make the possible range of praa transformations stand out in its full generality; in particular, certain important technical issues connected with the set-theoretic operator '∋', which selects an arbitrary element from a set, are missed. (c) Our treatment is rather more polished and systematic than that found in the papers cited. Although these differences cumulatively seem to have significant implications for the usability of a correct-praa system, perhaps it is fairest to say that our approach differs from that of Gerhart more in pragmatic than in theoretical terms.

The techniques sketched below seem to the author to open the way toward a practical technology guaranteeing program correctness; in this, we agree enthusiastically with Gerhart. To bring this technology into existence, one would have to implement a praa-verification/manipulation system like that which we will describe; features which can strengthen this system from the human-factors point of view should be energetically sought out and incorporated into it. Of course, a correct-praa manipulation system should be interactive. Once such a system was established, one group of algorithm developers could enter root praas, with full, formally verified proofs of their correctness into it; the application programmers who were the verification system's main users would then interactively combine these root praas into the larger programs that they required. An intermediate group of 'subsystem developers' might expand important, initially succinct and highly general, root praas into forms particularly convenient for particular classes of applications, and also supply

fleshed out praas and groups of praas as verified code blocks, procedures, and procedure libraries of intermediate size.

In section 2, following, we will develop all the bases of our method. In section 3 we illustrate the use of the method, specifically by proving the correctness of a simple high level algorithm ('sorting by counting'), whose root form we will initially describe by a succint, very high level text, which will then be manipulated systematically and converted by stages into a correct assembly-language version of the same algorithm. The material in section 3 is best viewed as a partial hand simulation of the use and nature of a verification/manipulation system of the sort we envisage.

Note that since the present paper is intended to be a convincing description of such a verification/manipulation system rather than a detailed specification for a particular system of this kind, we shall formalize *only to a degree felt to be helpful and intuitively convincing, but not more*. In particular, full syntactic specifications and inductive definitions, which the authors of papers on program correctness are sometimes zealous to present, will be painted in with no heavier a brush than seems appropriate.

Techniques like those which we will now describe can be used to develop correct parallel programs and also to prove the correctness of logical circuit designs. Note that the problem of proving the correctness of logical circuit designs, especially for logical circuits which store information internally and thus have very large numbers of states, is an important though little-studied problem.

2. Programming Language, Proof Formalism and Proof Rules, Code Transformation Rules

a. Programming Language.

Our programming language SL admits all the objects of (Zermelo-Frankel) set theory into its semantic universe; this includes all sets whether finite or infinite, all mappings, plus integers and the set of all integers in their ordinary set-theoretic sense. We are even happy to admit transfinite ordinals and cardinals, etc., though as a matter of fact in the present paper we shall make no use of these esoteric objects. We allow the minimal basic vocabulary of set theory to be extended in the usual way by the definition of new sets, predicates, and terms. Thus, for example, we will feel free to write oneone(s,t) for the set of all one-to-one mappings from the set s into the set t, etc. It is left to the reader to work out or look up the detailed definitions which reduce this notion and all others like it to the very spartan collection of primitives actually present in the axioms of set theory. Any valid expression of set theory is also a valid expression of our programming language.

Aside from all of this, which we swallow at a gulp, our language is very conventional. Each program in it consists of a *main block*, whose first statement is its *entry*, plus some finite number of auxiliary *functions*. The syntactic form of a function is illustrated by

(1)
```
function fname(par_1,...,par_n);
        function_body
end;
```

Both *fname* and $par_1,\ldots,par_n$ are simple tokens, *fname* is the *name* of the function and $par_1,\ldots,par_n$ are its *parameters*. The *function_body* in (1) is a *block*, that is, a list of *statements*. All the functions appearing in a program are required to have different names. It will be convenient for us to assume that function names are distinguished lexically

from the variable names appearing in a program. We assume that none of the arguments or variables appearing in any function appear in any other function; that is, all variables and arguments are *strictly local* to the function in which they appear.

Statements may be labeled, and with more than one label if desired. The form of a labeled statement is

(2) $L_1:L_2:\ldots:L_n:$ unlabeled_statement .

A given label can precede only one statement, and that only once. We allow unlabeled statements of only a few forms:

(a) Simple assignment statement:

variable = *expression*;

(b) Conditional transfer statement:

if *boolean_expression* then go to *label*;

The unconditional transfer is regarded as an abbreviation for if *true* then go to *label*.

(c) Function-call statement:

variable = *fname*($expn_1,\ldots,expn_n$) ,

where *fname* is the name of a function having n parameters, and $expn_1,\ldots,expn_n$ are a like number of expressions.

(d) Selection operator and selection statement:

the operator ∋*expn* selects an arbitrary element of the set-valued expression *expn*, and aborts if *expn* has the null-set value nℓ. We allow the selection-statement form

variable = ∋*expn* .

(e) Return statement:

return *expression*;

where *expression* is an arbitrary expression.

This completes the list of the statements of our basic language. Subsequently we will find it convenient to allow other more or less conventional language forms and language features (e.g. while loops, if-then-else constructions) as well, but in every case these additional features will be regarded merely as syntactic abbreviations for combinations of the fundamental statements (a)-(e).

All of the statement forms (a)-(e) are given their conventional semantics. Concerning function calls, our semantic assumption is that calls are allowed to be recursive, and that arguments are transmitted 'by value with delayed value return', that is, by incarnating an instance of the function for each call, and setting its parameters equal to the arguments of the call when the call operation begins, and then by setting the target *variable* of the call (c) to the value of the return *expression* (see (e)) when the call operation is terminated by a return statement.

We assume that a statement if C then go to L can never appear in a function unless the label L appears in the same function. A function is assumed never to modify its parameters.

(b) Proof Formalism and Proof Rules

The *variables* of a program Q consist of the variables which occur explicitly within it, plus an indefinite collection of additional variables which we keep in reserve for use during manipulation of the program (more properly, of a praa obtained by adding assumptions and assertions to the program). We regard these extra variables as denoting values which the original form of the program Q neither reads nor writes. The *places* π of a program are all those (syntactic) points in the program text which either immediately precede (the body of) an unlabeled statement, follow (the body of) such a statement, or which either precede or follow a label in the program. The *entry_place* of a program is the place preceding its first statement. A *program proposition* P is any syntactically well-formed predicate formula of set-

theory involving only the variables of the program as free variables. A *proposition-at-a-place* is a pair (P,π) consisting of a program proposition P and a program place π. We shall often choose to use the symbol P^{π} to denote (P,π). A *proposition-at-a-function* is a pair (P,fname) consisting of the name of one of the functions appearing in (the program of) a praa, together with a syntactically well-formed set-theoretic predicate P containing exactly n+1 variables, where n is the number of parameters of the function *fname*; thus P may be written as

$$P = P(res,a_1,\ldots,a_n).$$

By E we shall always denote some finite set of propositions-at-a-place or -at-a-function in a praa R that we are studying. The central formal elements of our proof formalism will be assertions of the form $E \vdash P^{\pi}$ where P^{π} is a proposition-at-a-place or -at-a-function. We will begin by explaining the meaning of such assertions, and then by listing a number of properties of such assertions which follow easily from the semantics which we have assigned to our language SL (and which could readily be proved formally if we bothered to write out a formal definition, in terms of 'state sequences', for this semantics). Then, having listed these properties, we can abandon all direct consideration of the states and semantics of SL, and shall use only the listed properties of the $E \vdash P^{\pi}$ relationship to deduce correctness. However, before doing so, it is appropriate that we should give formal definition to the notion of a praa and of the correctness of a praa.

Definition: (a) A *praa* R is a program Q (of the language SL), together with two sets E, E' of propositions-at-places and -at-functions of Q. The set E is called the *set of assumptions* of R and the set E' is called the *set of assertions* of R.

(b) The praa R is said to be *correct* or to be *valid* if $E \vdash P^{\pi}$ and $E \vdash P_1^{fname}$, for all P^{π} and P_1^{fname} in the set of assertions of R, where E is the set of assumptions of R.

To indicate that a praa R is valid we will often find it convenient to write $\triangleright$ R $\triangleleft$. If the text of a praa is shown explicitly, we shall indicate its assumptions and assertions by writing each of them (as a formula of LF) either in the place to which it belongs or, (in the case of propositions-at-functions) immediately preceding the first line of the function to which it belongs. To distinguish between program text, assumptions, and assertions, we shall prefix assertions by the sign '$\vdash$' and assumptions by the sign '$\models$'. An example of all this notation, and one that will be studied and manipulated extensively in section 3, is the following simple sort-by-counting praa:

```
▷ ⊨set(s) & f ∈ sing_val_maps(s,reals)
     place = nℓ;
     s' = nℓ;
Loop: ⊢ place ∈ one_one_maps(s',integers)
     & (∀y ∈ s') #{z∈s | f(z) < f(y)} < place(y)
          ≤ #{z∈s | f(z) < f(y) ∨ (f(z)=f(y) & z∈s')};
     if s = s' then go to Lout; /* note in what follows */
          x = ∋(s-s'); /* that Ω represents 'undefined' */
          place(x) = #{z∈s|f(z)<f(x)∨(f(z)=f(x)&place(z)≠ Ω)}+1;
          s' = s' ∪ {x};
     go to Loop;
Lout: ⊢ place ∈ one_one_maps(s,integers)
                          & range(place) = {n, 1 ≤ n ≤ #s}
          & (∀x,y ∈ s) f(x) < f(y) ⇒ place(x) < place(y) ◁
```

Having thus explained our intent and illustrated our notational conventions, we shall now proceed to define the logical meaning that we attach to the notions 'proposition-at-a-place' and 'proposition-at-a-function', and will also define the fundamental relationship $E \vdash P^{\pi}$.

Let R be a *praa*, and let Q be the program on which it is based.

(i) By a *state* of Q, we mean (as usual) a mapping which assigns a value (namely any object of set theory) to every variable of Q, and which assigns a place of Q to the special symbol *control_location*. If a state S assigns the place π to the symbol *control_location*, then we say that S is *at* π. By a *computation for* Q (or R) we mean a finite sequence of states of Q which evolve in accordance with the text of Q and the semantic rules of the language SL, and which (if we ignore its final state) contains no more states at places immediately preceding return statements than states at places immediately preceding function call statements. If the initial (resp. final) state of a computation c is at the place π, then we say that c is a computation *from* (resp. *to*) π.

(ii) If s is a state of Q at a place π of Q and P^{π} is a proposition at π, then we say that s *satisfies* P^{π} if the proposition $P(s(v_1),s(v_2),\ldots,s(v_n))$ is true, where $v_1,\ldots,v_n$ are the variables of P^{π}, and $s(v_j)$ is the value which s assigns to the variable v_j. If c is a computation of Q, then we say that c satisfies P^{π} if every state of c which is at π satisfies P^{π}.

(iii) Let c be a computation of Q, and let *f* name a function of Q. Suppose that f has n parameters $v_1,\ldots,v_n$, and let $P_1^f = (P_1(val,a_1,\ldots,a_n))^f$ be a proposition at f. We say that c *satisfies* P_1^f if, for every state s of c which is at the place immediately preceding a call of the function f, and given that s' is the next following state (if any) of c which is at a place immediately following the same call statement *and for which the section of c between s and s' contains an equal number of states before function calls and after such calls* then (writing $v_1,\ldots,v_m$ for all the variables of f and rvf for the value returned by f)

$$P_1(rvf, s(v_1),\ldots,s(v_n))$$

is true. (Note: in what follows we shall sometimes find it convenient to refer to s' as the *next matching* return *state* of s.)

(iv) If E is a set of propositions-at-places and at-functions of Q, and c is a computation of Q, then we say that c *satisfies* E if c satisfies every proposition at-a-place and every proposition-at-a-function of E.

(v) If Q, E, and P^{π} are as above and if π' is a place of Q then we say that $[\pi']E \mid\!- P^{\pi}$ if every computation of Q from π' which satisfies E also satisfies P^{π}.

(vi) If Q, E, and P_1^f are as above, then we say that $[\pi']E \mid\!- P_1^f$ if every computation of Q from π' which satisfies E also satisfies P_1^f.

(vii) The statements $[initial\_place]\ E \mid\!- P^{\pi}$ and $[initial\_place]E \mid\!- P_1^f$ can be abbreviated as $E \mid\!- P^{\pi}$ and $E \mid\!- P_1^f$ respectively.

Now that we have defined the fundamental $[\pi_1]E \mid\!- P$ relationship, we can begin to list its properties. Note that some of the properties we list can be deduced from others listed; to allow this redundancy is convenient even if somewhat inelegant.

Proof Rules:

(a) If $[\pi_1]E \mid\!- P^{\pi}$ and $[\pi_1](E \cup \{P^{\pi}\}) \mid\!- P_1^{\pi_2}$, then $[\pi_1]E \mid\!- P_1^{\pi_2}$; and the same holds if we replace $P_1^{\pi_2}$ by a proposition-at-a-function P_1^f.

(For if every computation to π_2 satisfying E also satisfies P^{π}, and every computation to π_2 satisfying E and P^{π} also satisfies $P_1^{\pi_2}$, then clearly every computation to π_2 satisfying E also satisfies $P_1^{\pi_2}$).

(b) If g names a function of Q, if also $[\pi_1]E \mid\!- P^g$ and $[\pi_1]E \cup \{P^g\} \mid\!- P_1^{\pi}$, then $[\pi_1]E \mid\!- P_1^{\pi}$; and the same holds if we replace P_1^{π} by P_1^f.

(By much the same argument given in support of (a).)

(c) If $E \subseteq E'$ and $[\pi_1]E \vdash P^{\pi}$, then $[\pi_1]E' \vdash P^{\pi}$, and similarly if P^{π} is replaced by P^{f}. (Obvious.)

(d) If P is any universally valid formula of logic, then $[\pi_1]E \vdash P^{\pi}$ for every E, π_1, and π, and similarly if P^{π} is replaced by P^{f}. (Obvious.)

(e) If $[\pi_1]E \vdash P^{\pi}$ and $[\pi_1]E \vdash (P \Rightarrow P_1)^{\pi}$ (where $\Rightarrow$ is the sign of predicate implication) then $E \vdash P_1^{\pi}$, and similarly if π is replaced by f. (Obvious)

(f) If $P^{\pi} \in E$ and $P_1^{f} \in E$, then $[\pi_1]E \vdash P^{\pi}$ and $[\pi_1]E \vdash P_1^{f}$. (Obvious.)

In writing the remaining statements of the present group, we will use the convenient notation β_-, *statement*, β_+ to indicate that β_- is the place immediately before *statement* and that β_+ is the place immediately after *statement*. Moreover, if we wish to single out one of the variables x of a praa R, we shall write the list of all its variables as (*x, other_vars*). Otherwise we shall write this list of variables simply as *vars*.

(g) (Assignment statement proof rule). If β_-, x = expn(x, other_vars), β_+, (where *expn* is any expression depending on the variables *vars* of R), $\pi_1 \neq \beta_+$, and $[\pi_1]E \vdash (P(x, \text{other\_vars}))^{\beta_-}$, then $[\pi_1]E \vdash ((\exists x')(P(x', \text{other\_vars}) \ \& \ x = \text{expn}(x', \text{other\_vars})))^{\beta_+}$.

(For if we consider any computation from π_1 to β_+ satisfying E and let x' be the value which the variable x had immediately before the assignment statement was executed, then clearly P(x', other_vars) & x = expn(x', other_vars) at the end of the computation).

(h) (Transfer statement proof rule) If β_-, if C(vars) then go to L, β_+, where C is any boolean expression depending on *vars*), $\pi_1 \neq \beta_+$, and $[\pi_1]E \vdash (P(\text{vars}))^{\beta_-}$, then $[\pi_1]E \vdash (P(\text{vars}) \ \& \ \neg C(\text{vars}))^{\beta_+}$. (Obvious; this is the condition that the transfer not be taken.)

(i) (Function call proof rule). If f names a function of Q having m parameters, if β_-, x = f(expn$_1$(x,other_vars),...,expn$_m$(x,other_vars)), β_+, where expn$_1$,...,expn$_m$ are expressions, if $[\pi_1]E \vdash (P(x,other\_vars))^{\beta_-}$, $[\pi_1]E \vdash (P_1(val,a_1,...,a_m))^{f}$, and if $\pi_1 \neq \beta_+$, then

$$[\pi_1]E \vdash ((\exists x')(P(x',other\_vars) \;\&\; P_1(x,expn_1(x',other\_vars),\ldots\ expn_m(x',other\_vars)))^{\beta_+}.$$

(Consider a computation c from π_1 to β_+ satisfying E. then the section of c between the last call to f and the next matching <u>return</u> receives the values y_j = expn$_j$(x,other-vars) as parameters, and if *val* denotes the value of the expression appearing in this <u>return</u> statement the relationship $P_1(val,y_1,...,y_m)$ holds. Hence, if we let x' designate the value which this variable x has immediately before the function call at β_-, then both P(x',other_vars) and P_1(x,expn$_1$(x',other_vars),...,expn$_m$(x',other_vars)) are valid at the place β_+, immediately after return.)

(j) (Selection operator proof rule) If β_-, x = ∋exp (x,other_vars), β_+, (where *expn* is any expression depending on the variables of R), $\pi_1 \neq \beta_+$, and $[\pi_1]E \vdash (P(x,other\_vars))^{\beta_-}$, then $[\pi_1]E \vdash (\exists x')(P(x',other\_vars) \;\&\; x \in expn(x',other\_vars))$.

(By much the same justification as that offered for (f), except that in this case we know only that x ∈ expn(x',other_vars) rather than x = expn(x',other_vars)).

(k) (<u>return</u> statements) If β_-, <u>return</u> expn, β_+, and $\pi_1 \neq \beta_+$, then $[\pi_1]E \vdash (\underline{false})^{\beta_+}$. (Since no computation can reach the place β_+).

In order to give comfortable form to the proof rules which apply to the place π immediately following a label L, it will be convenient to assume $\beta_-^{(0)}$,L:, β_+, to let

$\beta_-^{(1)},\ldots,\beta_-^{(m)}$ denote all the places in the program Q which immediately precede a statement of the form *if* C *then go to* L (with the same label L), and, for each of these places $\beta_-^{(j)}$, to let $C^{\beta_-^{(j)},\beta_+}(vars)$ be the boolean expression occurring in the *if*-statement immediately following $\beta_-^{(j)}$ (while $C^{\beta_-^{(0)},\beta_+}$ is simply taken to be the constant expression *true*). Moreover, with each such L we introduce m+1 additional boolean constants, which we shall write as $from_{\beta_-^{(j)}}$, j = 0,...,m. (In heuristic terms, $from_{\beta_-^{(j)}}$ denotes the condition that one has just arrived at the label L from the place $\beta_-^{(j)}$). Using these notations, we can state the proof rules applying to labels as follows:

(ℓ) (With the above notations) If $\pi_1 \neq \beta_+$, then

$$[\pi_1]E \vdash (from_{\beta_-^{(0)}} \vee \ldots \vee from_{\beta_-^{(m)}})^{\beta_+}$$

(m) (with the above notations) If $[\pi_1]E \vdash (P(vars))^{\beta_-^{(j)}}$, then

$$[\pi_1]E \vdash (from_{\beta_-^{(j)}} \Rightarrow (C^{\beta_-^{(j)},\beta_+}(vars) \;\&\; P(vars)))^{\beta_+}$$

(Both of these are obvious if we consider computations to β_+ which reach β_+ either by a step from $\beta_-^{(0)}$ or by a conditional transfer from $\beta_-^{(j)}$, $j \geq 1$).

Function calls bear some resemblance to *go to* operations, and thus it is convenient to use auxiliary boolean variables like those introduced in connection with labels to state the proofrule which applies to function entrances. Specifically, if f names a function in the praa R with program Q, then we let $\beta^{(1)},\ldots,\beta^{(n)}$ denote all the places in Q which immediately precede any call $x = f(expn_1,\ldots,expn_m)$ to f. For each j we let $expn_1^{(j)},\ldots,expn_m^{(j)}$ denote the argument expressions which appear in the call immediately following $\beta^{(j)}$.

For each such f, we introduce n boolean constants, which we shall write as $\text{from}_{\beta(j)}$, j = 1,...,n. We let π be the place immediately $^{\beta}$following the header statement of f, i.e., immediately preceding the first executable statement of f.

(n) (With the above notations) If $\pi_1 \neq \pi$, then

$$[\pi_1]E \vdash (\text{from}_{\beta(1)} \vee \ldots \vee \text{from}_{\beta(n)})^{\pi}$$

(o) (With the above notations) If P is a predicate whose only free variables are the parameters $y_1,\ldots,y_m$ of f, and if

$$[\pi_1]E \vdash (P(\text{expn}_1^{(j)},\ldots,\text{expn}_m^{(j)}))^{\beta(j)} \quad \text{then}$$

$$[\pi_1]E \vdash (\text{from}_j \Rightarrow P(y_1,\ldots,y_m))^{\beta_j} .$$

Note that all non-parameter variables of f must be excluded from P, since when f is called a completely new 'incarnation' of it will be built, and in this incarnation the value of all non-parameter variables will have unknown values.

This could complete our list of elementary proof rules: however, it is convenient to give a somewhat more extensive list, and in particular to give rules which cover the usual syntactic extensions of our basic language. The principal syntactic extensions which we propose to allow are as follows:

(p) (<u>while</u>-loops) As usual, we regard the construction

<u>while</u> C; *code* <u>end while</u>; ...	as an abbreviation for	Loop: if¬C then go to Lout; *code*; go to Loop; Lout: ...

where Loop, Lout are labels which do not appear elsewhere.

The program place immediately following the label 'Loop' is called the *head* of the while-loop. The proof rule applying in this case is as follows: if β_-, while ... end while, β_+, if β is the head of the while loop, if $\pi_1 \neq \beta_+$ and π_1 is not within the while loop, if the loop cannot be entered except through β, and if $[\pi_1]E \vdash P^{\beta}$ and $E \vdash P^{\beta_-}$, then $[\pi_1]E \vdash (\neg C \ \& \ P)^{\beta_+}$. This is an obvious consequence of the if-statement proof rule.

(q) (if-then-else) We regard the construction

if C then
code1
else
code2
end if;
	as an abbreviation for	if ¬C then go to L1,
		code1
		go to L2;
		L1: *code2;*
		L2: ...

where L1, L2 are labels which do not appear elsewhere.
If we introduce notations for places by
β_-, if C then, $\beta_1,\ldots,\beta_2$, else, $\beta_3,\ldots,\beta_4$, end if, β_+,
and assume that $\pi_1 \neq \beta_1$, $\pi_3 \neq \beta_3$, and $\pi_+ \neq \beta_+$, then the following proof rules apply:

(q1) if $[\pi_1]E \vdash P^{\beta_-}$, then $[\pi_1]E \vdash (P \ \& \ C)^{\beta_1}$

(q2) if $[\pi_3]E \vdash P^{\beta_-}$, then $[\pi_3]E \vdash (P \ \& \ \neg C)^{\beta_3}$

(q3) if $[\pi_+]E \vdash P_1^{\beta_2}$ and $[\pi_+]E \vdash P_2^{\beta_4}$,

then $[\pi_+]E \vdash (P_1 \vee P_2)^{\beta_+}$.

All these are straightforward consequences of proof rules stated earlier.

(r) (Indexed assignment) We allow the familiar syntactic form

(3) $\quad \mathrm{var}(expn_1) = expn_2;$

to be used, but regard this as an abbreviation for the simple assignment

(4) var = $\{\langle expn_1, expn_2\rangle\} \cup \{x \in var \mid \neg pair(x) \vee (pair(x) \& \underline{hd}(x) \neq expn_1)\}$;

Here pair(x) is the set-theoretic predicate which is true if and only if x is an ordered pair, and $\underline{hd}$ is the set-theoretic function which sends each ordered pair into its first component. The proof rule for (3), which we shall not bother to write out explicitly, follows at once if we apply rule (g) to (4). Note in this same connection that *var{expn}* is allowed as an abbreviation for $\{\underline{tl}(x),\ x \in var \mid pair(x)\ \&\ \underline{hd}(x) = expn\}$, where $\underline{tl}$ is the set-theoretic function which sends each ordered pair into its second component, and that *var(expn)* abbreviates $\exists(var\ \{expn\})$.

(s) (Multiple assignment and superselection operator) As we will see in the following subsection, the selection operator $\exists s$ is of particular significance, since at the place immediately following an assignment $x = \exists s$ we know nothing about x other than the fact that it is a member of s. Thus if we replace the statement $x = \exists s$ by any correct, single entry, single exit praa R_1 which changes no variable of R other than x and for which the assertion $(x \in s)^{\beta_+}$ is available at the exit place β_+ of R_1, then the praa R containing the statement $x = \exists s$ remains correct. This statement is an essential key to construction of correct praas by combination; in exploiting it, we will often want to make use of a code sequence

$$temp = \exists\{\langle u_1,\ldots,u_n\rangle \mid C(u_1,\ldots,u_n,other\_vars)\};$$
$$x_1 = temp(1);\ \ldots;\ x_n = temp(n);$$

here, *temp* is assumed to be a 'temporary' variable not occurring elsewhere in a praa R, and temp(j) denotes the j-th component of its n-tuple value. It is convenient to allow this important code sequence to be abbreviated by the syntactic form

(6) $\langle x_1,\ldots,x_n\rangle = \langle u_1,\ldots,u_n\rangle$ <u>where</u> $C(u_1,\ldots,u_n,\text{vars})$.

The proof rule for this statement is an immediate consequence of the selection and assignment rules given above, and can be stated as follows:

Let β_- (resp. β_+) be the place which immediately precedes (resp. follows) a 'superselection' statement of the form (6). Then if $E \vdash (P(x_1,\ldots,x_n,\text{other\_vars}))^{\beta_-}$, it follows that

$$E \vdash ((\exists u_1,\ldots,u_n)\ P(u_1,\ldots,u_n,\text{other\_vars}) \ \&\ C(u_1,\ldots,u_n,\ \text{other\_vars}))^{\beta_+}.$$

(t) (Shadow-Variable Principle) 'Shadow' variables are variables occurring neither in the initial nor in the final form of a praa, but introduced into intermediate forms of it (during manipulation) to facilitate praa proofs that would otherwise be difficult or even impossible. The rules which govern the introduction and removal of shadow variables play an essential role in verification; generally speaking, proof rules like those given in the present section are incomplete without supplementary shadow-variable rules, but become complete as soon as a few simple shadow-variable rules are introduced. For this reason we have felt it necessary to mention the shadow-variable issue here, even though detailed statement of the rules connected with shadow variabels is postponed to the following subsection ('Code Transformation Rules') into which it fits more naturally.

A general objection to the preceding rules is that they do not provide any sufficiently powerful method for eliminating assumptions from a praa. For this reason, the two following principles are of fundamental importance.

Lemma (Induction principle for propositions-at-a-place). Let π be a place in a praa R with variables *vars*. Let E be a set of propositions of R, and let P be a proposition at π. Let $\beta_-^{(0)}$, L, π, so that the place π immediately follows a label L, and let $\beta_-^{(1)},\ldots,\beta_-^{(m)}$ denote all the places in the program Q which immediately precede a statement of the form if C then go to L (with the same label L). For each of these places $\beta_-^{(j)}$, let $C^{\beta_-^{(j)},\pi}(vars)$ be the boolean expression occurring in the conditional transfer statement immediately following $\beta_-^{(j)}$ (while $C^{\beta_-^{(0)},\pi}$ is the constant expression true). Then, if $\pi_1 \neq \pi$ and $[\pi_1]E \cup \{P^\pi\} \vdash (C^{\beta_-^{(j)},\pi} \Rightarrow P)^{\beta_-^{(j)}}$ for all j = 0,...,m, it follows that $[\pi_1]E \vdash P^\pi$.

Lemma (Induction principle for propositions-at-a-function) Let f name a function in a praa R having variables *vars*. Let E be a set of propositions of R, and let P^f be a proposition at f. Let π be the place immediately following the function statement which heads f (so that π immediately precedes the first 'executable' statement of f). Suppose that the places in f which immediately precede return statements are $\rho_1,\ldots,\rho_m$, and let $expn_1,\ldots,expn_m$ denote the expressions which appear in the corresponding return statements. Let the parameters of f be $x_1,\ldots,x_n$. Then if $[\pi]E \cup \{P^f\} \vdash (P(expn_j,x_1,\ldots,x_n))^{\rho_j}$ for j = 1,...,m, it follows that $[\pi_1]E \vdash P^f$ for any place π_1.

Proof of the first induction principle: Suppose the contrary. Let c be a computation from π_1 to π satisfying E but not P^π, and suppose that c is of minimum length among all computations having this property. Truncate the last step of c, thus obtaining a computation c' one step shorter, which is to one of the places $\beta_-^{(j)}$. Because of the minimality of c, c' satisfies P^π, and thus since $[\pi_1]E \cup \{P^\pi\} \vdash (C^{\beta_-^{(j)},\pi} \Rightarrow P)^{\beta_-^{(j)}}$ P clearly holds for the final state of c, a contradiction which proves our assertion. Q.E.D.

Proof of the second induction principle: Suppose the contrary. Let c be a computation from $[\pi_1]$ to a place immediately following a call of f. Suppose that c satisfies E but not P^f, and suppose also that c is of minimum length among all computations having this property. Then a final section c_0 of c must go from π to an invocation of f, and from there through f to the place immediately following this invocation and have the property that within c_0 there occur as many states at places preceding call statements as at places following call statements. Let $x_1',\ldots,x_n'$ be the values assigned to the variables $x_1,\ldots,x_n$ by the initial state of c_0. Truncate the last step of c_0, thus obtaining a computation c' one step shorter, which is to one of the places ρ_j. Because of the minimality of c, c' satisfies P^f. Since $[\pi]E \cup \{P^f\} \vdash (P(expn_j,x_1,\ldots,x_n))^{\rho_j}$, it then follows that c_0 satisfies P^f. If c_- is the part of c which precedes c_0, then it is clear from the minimality of c that c_- satisfies P^f. Hence altogether c satisfies P^f, a contradiction which proves our assertion. Q.E.D.

To use the proof rules described in the preceding pages to prove the correctness of a root praa R, we will generally proceed in the following way. Initially, R will be a program text annotated with assumptions but with no assertions. Clearly, any praa containing assumptions but no assertions is correct. The proof rules will then be used to deduce various assertions; any particularly significant 'output assertion' playing a significant role in the use of R should be included among the assertions deduced. Then finally, the first and second induction lemmas stated just above will be used to remove superfluous assumptions; in many cases, the only surviving assumption will be an input assumption characterizing the data objects on which R will work properly.

In writing out the text of a praa, it will be useful to distinguish between assumptions which are to remain in the final fully 'proved' version of the praa and assumptions

introduced temporarily but to be removed as the praa is manipulated. In order to make this distinction vivid, we shall mark assumptions of the latter, purely temporary, class with the prefixed sign $\models$ rather than $\models\!\!=$; and then each use of one of the two inductive lemmas just stated transforms some occurrence of the sign $\models$ into an occurrence of $\vdash$. Note that the temporary assumptions which appear in our formalism often correspond closely to the 'loop invariants' of the Floyd and Hoare formalisms.

We can use the 'counting sort' praa shown earlier to illustrate the general workings of our proof formalism. To prove the counting sort praa correct, we begin by striking out the output assertion which it contains, and by converting the assertion appearing at the place immediately following the label 'Loop' to an assumption. This gives a praa which, since it contains no assertions, is certainly correct. From this applying the proof rules stated in the preceding pages to add assertions, we obtain the following, still correct, praa. (Note that relatively elementary set-theoretic reasoning has been applied to deduce one assertion from another at various fixed places in the praa shown below; in a fully formalized system, justification of this reasoning to the satisfaction of a logical proof verifier would be the most onerous part of our total verification procedure. We do not show the detailed steps of this reasoning, but the reader who wishes to master our proof technique is strongly urged to reconstruct it.)

$\models$ set(s) & f $\in$ sing_val_maps(s,reals)

place = nℓ;

s' = nℓ;

$\vdash$ place $\in$ one_one_maps(s',integers)

& ($\forall$y$\in$s') #{z$\in$s | f(z)<f(y)} < place(y)

$\leq$ #{z$\in$s | f(z)<f(y) $\vee$ (f(z)=f(y) & z$\in$s')};

Loop: $\models$ place $\in$ one_one_maps(s',integers)

& ($\forall$y$\in$s') #{z$\in$s | f(z)<f(y)} < place(y)

$\leq$ #{z$\in$s | f(z)<f(y) $\vee$ (f(z)=f(y) & z$\in$s')};

if s = s' then go to Lout;

x = $\ni$(s-s'); $\vdash$ x$\in$s & x$\notin$s' & x$\notin$domain(place)

($\forall$y$\in$s')((f(x)$\geq$f(y) $\Rightarrow$ #{z$\in$s | f(z)<f(x)$\vee$(f(z)=f(x) & z$\in$s')}$\geq$place(y))

&(f(x)<f(y) $\Rightarrow$ #{z$\in$s | f(z)<f(x)$\vee$(f(z)=f(x) & z$\in$s')}<place(y)-1))

$\vdash$ place $\in$ one_one_maps(s',integers) & x$\notin$domain(place)

& (#{z$\in$s | f(z)<f(x)$\vee$f(z)=f(x) & place(z)$\neq\Omega$}+1)

$\notin$ range(place)

place(x) = #{z$\in$s | f(z) < f(x) $\vee$ f(z)=f(x) & place(z)$\neq\Omega$}+1;

$\vdash$ place $\in$ one_one_maps(s' $\cup$ {x},integers)

& ($\forall$y$\in$s' $\cup$ {x}) #{z$\in$s | f(z) < f(y)} < place(y)

$\leq$ #{z$\in$s | f(z)<f(y) $\vee$ f(z)=f(y) & z$\in$(s'$\cup$ {x})};

s' = s $\cup$ {x};

$\vdash$ place $\in$ one_one_maps(s',integers)

& ($\forall$y$\in$s') #{z$\in$s | f(z)<f(y)} < place(y)

$\leq$ #{z$\in$s | f(z) < f(y) $\vee$ f(z) = f(y) & z$\in$s'};

go to Loop;

Lout: $\vdash$ place $\in$ one_one_maps(s,integers)

& ($\forall$y$\in$s) #{z$\in$s | f(z) < f(y)} < place(y) $\leq$ #{z$\in$s | f(z) $\leq$ f(y)}

$\vdash$ ($\forall$x,y $\in$ s) f(x) < f(y) $\Rightarrow$ place(x) < place(y)

& domain(place) = {n, 1 $\leq$n $\leq$ #s}

Once this praa is reached, the first inductive lemma can be used to degrade the assumption at 'Loop' to an assertion; then by dropping all but this loop assertion and the assertions at 'Lout' we put the praa into its desired form.

Note that much of the logical detail appearing in the praa written just above is irregular enough in form to evade the grip of automatic simplification routines working at any level of power easy to develop at the present time. This is the central difficulty of current program-verification technology. The transformational techniques which we shall now begin to explain aim to make it unnecessary to supply nearly this much detail except when proving the correctness of a root praa.

c. Code Transformation Rules

We shall now begin to state the rules, central to our proposed approach, which allow the manipulation and combination of correct praas. We begin with a class of rules which, since they assume that specific assertions are available at particular points, can only be stated for praas. Subsequently, general auxiliary rules which could as well be stated for programs (rather than praas) will be listed. Throughout this section, we suppose that R is a valid praa, that P^{π} is one of the assertions-at-a-place in R, that f names a function appearing in R, and that P_1^f is one of the assertions-at-a-function in R.

Substitution and Other Fundamental praa Rules

(1.a) (Equality substitution) If P is $(e_1=e_2)^{\pi}$, then e_1 can be replaced by e_2 in any statement at π.

(1.b) (Member substitution) If P^{π} is $(e_1 \in e_2)^{\pi}$, then $\ni e_2$ can be replaced by e_1 in any statement at π.

(The praa in which $\ni e_2$ occurs is known to be (partially) correct no matter what element is chosen from e_2 when e_2 is evaluated; hence it remains correct if the specific element e_1 is selected.)

(1.c) (Subset substitution) If P^{π} is $(e_1 \subseteq e_2)^{\pi}$, then $\ni e_2$ can be replaced by $\ni e_1$ in any statement at π. (Justified in much the same way as (b).)

(1.d) (Dead code removal/insertion) If $(false)^{\pi}$ holds for a set of places π at which there are no assumptions (so that no computation satisfying all assumptions can reach any of these places) then the statements immediately following these places can be removed. Conversely, any syntactically admissible code, with arbitrary assumptions and assertions, can be inserted at a place at which $(false)^{\pi}$ holds, provided that label duplication is avoided.

(1.e) (Block substitution rule) Let R_1 be a correct praa whose function names are disjoint from those of R, suppose that the variables appearing in functions of R_1 are all distinct from

the variables appearing in R, and suppose that all the variables $x_1,\ldots,x_m$ common to R_1 and R appear either in the 'main block' (i.e. 'main program') of R, or that all these variables appear in a single function f of R. (In the former case, we shall say that R_1 is *insertable into the main block* of R; in the latter, that R_1 is *insertable into the function* f of R.) Let π be the entry place of R_1, and let π_1 be some other place in R_1. Suppose that the only variables among $x_1,\ldots,x_m$ modified by R_1 are $x_1,\ldots,x_n$ (where of course $n \leq m$), and let $x_1',\ldots,x_m'$ be a group of m additional variables of R_1 which do not appear either in R or R_1. Let $C = C(x_1,\ldots,x_n,x_1',\ldots,x_m')$ be a logical formula whose only free variables are those indicated. Let E_1 be the assumptions of R_1. Then if for R_1 we have

$$(7)\quad [\pi]E_1 \cup \{(x_1=x_1' \,\&\ldots\&\, x_m=x_m')^{\pi}\} \vdash (C(x_1,\ldots,x_n,x_1',\ldots,x_m'))^{\pi_1},$$

we say that *the R-effect of* R_1 *to the place* π_1 *is governed by* C.

Next, in addition to the above assumption, suppose that all the labels in R_1 are distinct from the labels appearing in R, with the exception of precisely k labels $L_1,\ldots,L_k$, all of which appear in R either in the main block of R (if R_1 is insertable into the main block of R) or in a particular function f of R (if R_1 is insertable into f). Suppose that the context in which each of these labels appears in R_1 is L_j: go to L_j; (i.e., that the statement in R_1 following each of these labels is a 'stop'). Suppose that immediately after the entry place π of R_1 a label L appears. For each $j = 1,\ldots,k$, let $C_j(x_1,\ldots,x_n, x_1',\ldots,x_m')$ govern the R-effect of R_1 to the place $\beta_+^{(j)}$ immediately following the label L_j. Suppose that in R each of the labels L_j, $j = 1,\ldots,k$ appears in the context

$$(8)\quad \beta_-^{(j)}, \langle x_1,\ldots,x_n\rangle = \langle u_1,\ldots,u_n\rangle \ \underline{\text{where}}\ C_j(u_1,\ldots,u_n,x_1,\ldots,x_m);\ L_j:$$

i.e., follows a where statement of the indicated form, and that if E designates the assumptions of R we have

(9) $E \vdash ((\forall u_1,\ldots,u_n)\ (C'_j(u_1,\ldots,u_n, x_1,\ldots,x_m)$

$$\Rightarrow C_j(u_1,\ldots,u_n,x_1,\ldots,x_m)))^{\beta^{(j)}}$$

for $j = 1,\ldots,k$. Suppose finally that if $1 \leq i \neq j \leq k$ we have

(10) $E \vdash (\forall u_1,\ldots,u_n)(\neg C'_j(u_1,\ldots,u_n,x_1,\ldots,x_m))^{\beta^{(i)}}$.

Then if we fuse R and R_1 together by modifying them in the following way, the praa R' which results is still correct:

(i) Replace each of the k where statements (8) appearing in R by the statement go to L;

(ii) Add the functions of R_1, with all their assumptions and assertions, to the collection of functions of R;

(iii) If R_1 is insertable into the main block of R, then insert the main block of R_1 at any place in R immediately following an unconditional go to statement. If R_1 is insertable into the function f of R, then insert the main block of R_1 at any place in f immediately following an unconditional go to or a return statement. In either case, the text of R_1 should be inserted into R along with all the assumptions and assertions present in it, and the labels L_j, $j = 1,\ldots,k$ originally present in the text of R_1 should be suppressed, leaving only one copy of these labels in R', namely those copies originally present in R.

To convince ourselves of the correctness of the praa R' to which this rather complex and general substitution rule leads us, we can reason as follows. Consider a computation c' which starts at the entry place of R' and satisfies all the assumptions of R'. We can decompose c' into a sequence of subsections $c_1,\tilde{c}_1,c_2,\tilde{c}_2,\ldots,c_n,\tilde{c}_n$, where each c_j is a computation in R and each $\tilde{c}_j$ is a computation in R_1; $\tilde{c}_j$ always starts at the entry place of R_1 (and if $j < n$) goes to some

place $\beta_+^{(j)}$ immediately preceding one of the go to L_j statements in R_1. Consider one of the subcomputations $\tilde{c}_j$, $j < n$, let $v_1' \ldots v_m'$ be the values assigned to the variables $x_1,\ldots,x_m$ by the first state of $\tilde{c}_j$, and let $v_1 \ldots v_m$ be the values assigned to these variables by the final state of $\tilde{c}_j$. Clearly $v_k' = v_k$ for $n < k \leq m$, and equally clearly all the states of $\tilde{c}_j$ assign the same value to any variable not mentioned in R_1. What we must show is that if the state of c' immediately preceding $\tilde{c}_j$ is at the place $\beta_-^{(\ell)}$, then $\tilde{c}_j$ goes to the palce $\beta_+^{(\ell)}$; and that the values $v_1',\ldots,v_n'$ are values that could be assigned to $x_1,\ldots,x_n$ by the where statement (8), which is the same as requiring that $C_\ell(v_1,\ldots,v_n,v_1',\ldots,v_m')$. Since we assume that the R-effect of R_1 to the place $\beta_+^{(\ell)}$ is governed by $C_\ell'(x_1,\ldots,x_n,x_1',\ldots,x_m')$, and taking note of (9), we see that we have only to prove that the final state of $\tilde{c}_j$ is at $\beta_+^{(\ell)}$. Suppose this to be false, so that the final state of $\tilde{c}_j$ is at some $\beta_+^{(i)}$ for which $i \neq j$. Then we would have $C_i'(v_1,\ldots,v_n,v_1',\ldots,v_m')$, so that the proposition $(\exists v_1 \ldots v_n)\ C_i'(v_1,\ldots,v_n, x_1,\ldots,x_m)$ would hold for the state at $\beta_-^{(\ell)}$ which immediately precedes the first state of $\tilde{c}_j$. By (10), this is impossible. Note in connection with all this that since each $\tilde{c}_j$ is a valid computation of R_1, the computation c satisfies all the assertions and assumptions of R_1, in addition to all the assertions and assumptions of R.

When the block substitution rule (e) is applied, the next few rules will often be useful.

(1.f) Let $x_1',\ldots x_n'$ be variables of R not otherwise appearing in R. Let

(11) $\beta_-,\langle x_1,\ldots,x_n\rangle = \langle u_1,\ldots,u_n\rangle$ where $P_1(u_1,\ldots,u_m,\text{vars})$ & $P_2(u_{m+1},\ldots,u_n,\text{vars}),\beta_-$

Then the where statement in (11) can be replaced to give

(12) $\beta_-,\langle x_1',\ldots,x_m'\rangle = \langle u_1,\ldots,u_m\rangle$ where $P_1(u_1,\ldots,u_m,\text{vars})$;
$\langle x_{m+1}',\ldots,x_n'\rangle = \langle u_{m+1},\ldots,u_n\rangle$ where $P_2(u_{m+1},\ldots,u_n,\text{vars})$;
$x_1 = x_1'$; $x_2 = x_2'$; ...; $x_n = x_n'$;,β_+

The new places introduced into R by this substitution should initially carry no assumptions or assertions.

Conversely, if R contains the code text (12), if the variables $x_1', \ldots, x_m'$ appear nowhere else in R, and if no assumptions or assertions are present in (12) except perhaps at the places β_- and β_+, then (12) can be replaced by (11), in which case any assumptions and assertions present at β_- or β_+ in (12) should be carried over to the corresponding places in (11). (This rule can be obtained by combining rule (1.b) above with the dead variable and redundant assignment rules to be stated shortly.)

(1.g) Any occurrence of the expression $\ni\{expn\}$ in a correct praa R can be replaced by an occurrence of *expn*, and conversely.

The following proof rule follows by the argument given in justification of (1.e):

(1.h) (Subblock transformation principle) Let R be a praa, and let B be a block of code in R having only one entry. Let η immediately precede the first statement of B, let π be a place in B, and let π_1 be a place not in B. Let R' be the praa consisting of B and of all the functions which might be called directly or indirectly from a statement of B. Let $v_1, \ldots, v_m$ be the set of all variables of B which are not dead at η, let $v_1', \ldots, v_m'$ be variables of R which appear nowhere else in R. Let *vars* be the variables of R, E be the assumptions of R, let E' be the subset of these assumptions which are at places in R', and let $P = P(v_1', \ldots, v_m', vars)$ be a predicate having exactly the indicated free variables. Then if in R' we have

$$[\eta]\ E' \cup \{(v_1=v_1' \ \&\ldots\&\ v_m=v_m')^{\eta}\} \vdash (P(v_1', \ldots, v_m', vars))^{\pi}$$

and if in R we have $[\pi_1]E \vdash (P_1(v_1, \ldots, v_n))^{\eta}$ for a predicate P_1 with the indicated free variables, then in R we have

$$[\pi_1]\ E \vdash ((\exists v_1', \ldots, v_m')\ P_1(v_1', \ldots, v_m') \ \&\ P(v_1', \ldots, v_m', vars))^{\pi}.$$

Auxiliary General Transformation Rules.

Next we shall state a class of rules which are general in that they can be applied both to praas and to programs. However, we shall be careful to state the forms of these rules which apply to praas. We continue to assume that R is a valid praa, and of course all the transformations we describe will preserve praa validity.

Group 1: Label-related rules.

(2.a) A label L not otherwise appearing can be inserted anywhere in R, thus introducing a new place π immediately following L, with no assumptions or assertions at π.

(2.b) If there is no if ... then go to L statement in R referencing a given label L, and β_-,L:,β_+, then we can move all the assertions and assumptions at β_+ to β_-, and remove L, thus eliminating the place β_+.

(2.c) The statement β_-, if *false* then go to L,β_+ can be eliminated, provided that we move all assumptions and assertions at β_+ to β_- and then eliminate β_+. Similarly, a statement of this form can be inserted anywhere in a praa, thus introducing a new place π immediately following it, with no assumptions or assertions at π.

(2.d) In the special context β_-, if C then go to L; L:,β_+, we can delete the if-statement, provided that all the assumptions and assertions at β_- are moved to the place following the if-statement and that the place β_- is eliminated. Conversely, if L is a label not otherwise occurring, we can introduce the combination

if C then go to L; L:

at any point in R, thus introducing two new places, initially with no assumptions or assertions at either of these places.

(2.e) (Will-go-to rule) If π, if C then go to L, β_+; if the statement following the label L is if C_1 then go to L1, if an assertion P is available at π and if P & C $\Rightarrow$ C1, then the statement if C then go to L may be replaced by if C then go to L1, and conversely.

(2.f)(Wont-go-to-rule) If π, if C then go to L,β_+; if the context following L is if C_1 then go to L1; L2: if the assertion P is available at π, and if P & C $\Rightarrow \neg C1$, then the statement if C then go to L may be replaced by if C then go to L2, and conversely.

(2.g) (Go-to splitting) Given any boolean expression C_1, the statement if C then go to L can be replaced by the pair of statements if C & C_1 then go to L; if C & $\neg C_1$ then go to L . This introduces a new place between the two if-statements, and initially there will be no assumptions or assertions at this place.

(2.h) (Go-to combination) The contiguous pair of statements

$$\beta_-,\ \text{if } C_1 \text{ then go to } L;\ \text{if } C_2 \text{ then go to } L,\beta_+$$

can be replaced by the single statement

$$\text{if } C_1 \vee C_2 \text{ then go to } L;$$

provided that all assumptions (resp. assertions) at the place thereby abolished are moved to the place β_- (resp. β_+).

The go-to rules that we have just stated can be seen to have a certain interesting completeness property: essentially, all other rules involving go-to's only can be obtained from these rules. Nevertheless, other more 'compound' go-to rules are worth stating as conveniences. A useful rule of this type is

(2.i) (Interchange of successive if-statements) If β_-, if C_1 then go to L1; if C_2 then go to L_2,β_+ , then the two successive if-statements can be replaced by the combination if $C_2 \wedge \neg C1$ then go to L2; if C1 then go to L1; provided that all assumptions and assertions P^{π} at the place π between β_- and β_+ are converted to assumptions and assertions $(\neg C_1 \Rightarrow P)^{\beta_+}$ at β_-.

To derive rule (2.i) from the preceding rules, we can proceed as follows: Given

```
        if C1 then go to L1;
        if C2 then go to L2;
```

use rule (2.d) and then rule (2.g) to get

```
        if C2 ∧ ¬C1 then go to L;
        if C2 ∧ C1  then go to L;
    L:  if C1 then go to L1;
        if C2 then go to L2;
```

where L is a label not otherwise occurring. The second line is superfluous by (2.d); (2.e) can be used to replace the occurrence of L in the first line by L1; the label L can then be dropped; and since the assertion $\vdash (\neg C2)^{\pi}$ i.e. $\vdash (C2 = \mathit{false})^{\pi}$, can be deduced at the place π immediately preceding the final if statement, equality substitution can be used to replace this statement by if *false* then go to L2, which (2.c) allows us to remove.

The transformation of if C then go to L into
if ¬C then go to L'; go to L; L':
where L' is a label not otherwise occurring, is a special case of (2.i).

The enthusiastic reader can amuse herself by using the go to related rules stated above to justify the following well known while loop transformation, in which B1 and B2 designate arbitrary praa subblocks; transform

```
while C;                          while C;
    if C1 then                        while C & C1;
       B1                                B1
    else                              end while;
       B2                             while C & ¬C1;
    end if;                              B2
end while;          into              end while;
                                  end while;
```

(This transformation introduces new places into a praa; neither assumptions nor assertions will initially be present at these new places.)

(2.j) (Isomorphic code rule) Consider two disjoint subsections s_1, s_2 of a praa R, each beginning with a label. Suppose that both are contained within the same function of R (resp. the 'main' program of R), and that both consist of unbroken contiguous sequences of program statements running up to a final go-to or return statement. Then s_1 and s_2 are said to be *isomorphic* if the statements, assumptions, and assertions of s_1 can be converted into the corresponding statements, assumptions, and assertions of s_2 simply by replacing each occurrence of any label attached to a statement of s_1 by the label appearing in the corresponding place in s_2.

Rule: If s_1 and s_2 are isomorphic, then any statement of the form if C then go to L1, where L1 is a label of s_1, can be replaced by if C then go to L2, where L2 is the corresponding label of s_2.

Using this rule and the dead code insertion rule (1.d), we can readily derive the following rule, which is often convenient.

(2.k) (Isolated block duplication) A subsection of a praa is said to be *isolated* if it consists of an unbroken contiguous sequence of program statements running from an immediately preceding go-to or return statement, up to and including a final go-to or return statement. A place π in a praa is said to be *safe* if the assertion $(false)^{\pi}$ is available (so that execution can never reach π).

Rule: A duplicate of any isolated subsection s of a praa R may be inserted at any safe place in R, provided that a subsection belonging to a given function (resp. to the main block of R) is inserted only in a place belonging to the same function (resp. the main block), provided that each

occurrence of all the labels L attaching to statements in the duplicated section are changed to occurrences of new labels L' nct otherwise appearing in R, and provided that all the assumptions and assertions at places within the section being duplicated are also duplicated. When s is duplicated, any statement of the form if C then go to L may be changed to if C then go to L', where L is a label in s and L' is the label in the duplicate section which corresponds to L.

Note that if we apply rule (2.k) in a manner which makes all the code in s dead, then it reduces to a rule isolated code sections to be moved to any safe place in a praa.

The following rules, which are easy consequences of (2.j) and (2.k), are sometimes useful.

(2.ℓ) (Go-to-pulling) If β_-, go to L, β_+, and if the context following the label L is L: *statement*; L':, then β_-, go to L can be replaced by β_-, *statement*; go to L', β_+; provided that every assertion and assumption present at the places before and after *statement* in its initial context is carried over to the appropriate place before or after the new copy of *statement*.

(2.m) (Go-to pushing) If β_-, *statement*; go to L', β_+; if the context preceding the label L' is L: *statement*; L':; if precisely the same assumptions and assertions appear at the place β_- as at the place following the label L, and if precisely the same assumptions and assertions appear at the place preceding the statement go to L' as at the place preceding the label L', then β_-, *statement*; go to L', β_+ can be replaced by β_-, go to L, β_+. When this change is made, the assumptions at β_- can be dropped, and the assertions at β_- kept.

The next few rules state various obvious facts having to do with the use of labels.

(2.n) (Assertion movement) In the context β_-, L_1:L_2:, β_+ any assertion at the place π between L_1 and L_2 can be moved back to β_- and any assumption at π can be moved forward to β_+.

(2.o) (Label permutation) If there are no assumptions or assertions at the place between L_1 and L_2, then L_1:L_2 can be replaced by L_2:L_1, any statement *if* C *then go to* L_1; can be replaced by *if* C *then go to* L_2, and conversely.

Group 2: Rules relating to assignment statements.

(3.a) An assignment statement of the form x = x can be inserted anywhere in R, thus introducing a new place π immediately following it, with no assumptions or assertions at π. Conversely, if β_-, x = x, β_+ then the statement x = x can be deleted, provided that we move all the assumptions and assertions at β_- to β_+ and eliminate β_-.

(3.b) Definition. A variable x of R is said to be *dead* at a place π in a praa if there is no path from π through the program Q of R which reaches a use of x (either in a program statement or in an assertion or assumption) without first passing through some assignment whose target variable is x.

If π is a place at which the variable x is dead, then an assignment statement x = *expn* or x = $\ni$*expn* can be inserted immediately before π, thus introducing a new place π_- immediately preceding the assignment statement, with no assumptions or assertions at π_-. Conversely, if π_-, x = *expn*, π_+ and x is dead at π_+, then the statement x = *expn* can be deleted, provided that we move all the assumptions and assertions at π_- to π_+ and delete π_-.

(3.c) (Shadow variable rule) A family F of variables appearing in a praa R is said to be a *shadow variable* family if none of the variables of F appears either in the controlling boolean expression C of any statement *if* C *then go to* L, in any assumption or assertion of R, or in any expression assigned to a variable not in F.

Rule: If all assignments to the variables of a shadow variable family are deleted from R, then R remains correct.

(3.d) (Renaming rule) The labels and the variables occurring in the statements, assumptions, and assertions of a praa can be renamed in any fashion, provided that every occurrence

of every variable and label receives the same new name, and provided that no two variables or labels having originally distinct names receive the same name.

(3.e) (Dead variable re-use) Suppose that R is a correct praa, that x and x_1 are two variables of R which do not occur either in two different functions of R or one in a function and the other in the main block of R, that B is a subregion of R, that x is dead at every place in B, and that x_1 is dead at every place of exit from B, i.e. every place which is either followed by

(i) a statement if C then go to L, where the place following L is not in B;

(ii) a statement return e;

(iii) a statement if C then go to L, with C different from the constant *true*, and with the place following this statement not in B;

(iv) a statement or label s of any other form, and with the place following s not in B.

(We define the notion '*place of entry into* B ' similarly, but do not give the details of this definition, leaving it to the reader to work them out.) Then if we replace every occurrence of x in B by an occurrence of x_1, and then insert an assignment $x = x_1$ at every place of entry into B, R remains correct.

Note: This rule can be derived by using rule (3.b) to insert the assignments $x = x_1$ at every place of entry into B and assignments x = expn immediately prior to each assignment x_1 = expn with target variable x_1. If this is done, then $(x = x_1)^{\pi}$ will hold for each place π of B, so that the equality substitution rule stated previously allows each occurrence of x_1 in an expression, assumption, or assertion in B to be replaced by an occurrence of x. After this replacement the assignments to x_1 occurring in B become dead and rule (3.b) can be used to delete them.

Group 3. Rules relating to function calls.

(4.a) Any group G of functions which is never called either from the main block of a praa R or from any other function not in G can be removed, provided that all the assertions and assumptions at these functions and at places within them are removed at the same time.

(4.b) (Moving code on-line) Any function call β_-, x = f(expn$_1$,...,expn$_m$),β_+ can be replaced by the block of code constructed as follows:

(i) Let $y_1,\ldots,y_n$ be the variables, $y_1,\ldots,y_m$ be the parameters, and $L_1,\ldots,L_k$ be the labels occurring in the statements of f. Let $y',y_1',\ldots,y_n'$ be a set of variables, and $L',L_1',\ldots,L_k'$ be a set of labels, not otherwise occurring.

(ii) Take the text of f, and using it build up an 'online variant' of f as follows: replace the header line of f by the set of assignments

$$y_1' = \text{expn}_1;\ \ldots,\ y_m' = \text{expn}_m\ .$$

Replace the trailer line ('end') of f by

$$L':\ \models P_1(y',y_1',\ldots,y_m')$$
$$\vdash P_2(y',y_1',\ldots,y_m')$$

where P_1 (resp. P_2) is the conjunction of all assumptions-at-f (resp. assertions-at-f) (each of which is a proposition of the form $P(y',y_1,\ldots,y_m)$.

In the remaining text of f, replace every occurrence of the variable y_j (including occurrences in assumptions and assertions) by an occurrence of y_j', j = 1,...,n; and replace every occurrence of the label L_j by an occurrence of L_j', j = 1,...,k. Then replace every ' return expn ' statement by

y' = expn; go to L';

(iii) Having in this way built up the online variant *olvt* of f, replace the function call $x = f(expn_1,\ldots,expn_m)$ by

olvt; x = y' .

(4.c) (Moving code off-line) Let R be a correct praa, and let B be a subsection of R all of which belongs either to a single function of R or to the main block of R. Suppose that B consists of a contiguous group of statements (with attached assumptions and assertions), running up to but not including a label L^e. Suppose that no statement
<u>if</u> C <u>then</u> <u>go</u> <u>to</u> L outside B involves a label L present inside B, and that no statement of this form inside B involves any label L outside B other than the label L^e. Let the variables appearing in the statements of B be $y_1,\ldots,y_m$, and suppose that of these variables the subcollection $y_1,\ldots,y_n$ are those not dead at the (necessarily unique) entry place of B, while the subcollection $y_k,\ldots,y_\ell$ are those modified by some statement of B. Let f be a function name distinct from the names of all functions in R, and introduce a new function with this name, as follows:

(i) Let the labels attached to statements of B be $L_1,\ldots,L_p$. Introduce new variables $y',y'_1,\ldots,y'_m$, $y''_1,\ldots,y''_m$ and labels $L',L'_1,\ldots,L'_p$ distinct from all the variables and labels occurring in R, and substitute $y'_1,\ldots,y'$, $L',L'_1,\ldots,L'_p$ for $y_1,\ldots,y_m$, $L^e,L_1,L_2,\ldots,L_p$ respectively in all the statements, assumptions, and assertions of B, thereby obtaining a new text B'.

(ii) Choose $n' \geq n$, $k' \leq k$, $\ell' \geq \ell$, and surround B' by the following header and trailer lines:

<u>function</u> $f(y''_1,\ldots,y''_{n'})$;
$y'_1 = y''_1;\ldots;y'_n = y''_n;$ B'
L': <u>return</u> $\langle y_{k'},\ldots,y_{\ell'}\rangle$;
<u>end</u>;

(iii) Add f to the collection of functions of R, introduce a new variable t not otherwise occurring, and replace B by the code fragment

$$t = f(y_1,\ldots,y_n'); \; y_k' = t(1); \; \ldots, \; y_\ell' = y(k'-\ell'+1);$$

Initially, no assertions or assumptions should be present at any of the places internal to this code fragment.

The following rule, which allows one function call to be substituted for another, will often be useful in connection with applications of rule (c).

(4.d) (Isomorphic function rule) Let R be a correct praa, and let f and f' be two functions of R. Suppose that there exists a 1-1 mapping which interchanges the variables and labels of f with the variables and labels of f' in a manner which converts the parameters of f to the parameters of f', the sequence of statements constituting the code text of f into the text of f', the assumptions and assertions present at the various places of f to the corresponding assumptions and assertions present at the places of f', and vice-versa. Suppose that this same interchange converts the set of assumptions-at-f and assertions-at-f into the set of assumptions- and assertions-at-f', and vice-versa. Then any function statement $x = f(expn_1,\ldots,expn_m)$ appearing in R can be changed to $x = f'(expn_1,\ldots,expn_n)$, and vice-versa.

(4.e) (Function call insertion) In order to state this transformation rule, it is convenient to make a preliminary definition. Let R be a praa with assumptions $\mathcal{E}$. Let f,g be functions of R, and let π be a place in g. Let P^g and P_1^π be predicates at g and at π respectively.

Definition. We write $[f]\mathcal{E} \mid\!- P^g$ (resp. $[f]\mathcal{E} \mid\!- P_1^\pi$) if the assertion P^g (resp. P^π) holds in the praa R' obtained by modifying R in the following way: letting $x,x_1,\ldots,x_n$ be variables of R not occurring elsewhere, replace the main block of R by the statement $x = f(x_1,\ldots,x_n)$ (i.e., by a single, direct call to f).

Rule: Suppose that R, E, and f are as above, and that for every assertion-at-a-function P^g and every assertion-at-a-place P_1^π for which the place π is within some function g of R we have $[f]E \vdash P^g$ and $[f]E \vdash P_1^\pi$. (If this is the case, then we shall say that f *allows entry*.) Let π' be any place within R, and let x' be a variable not otherwise occurring in R. Then if we insert the function call

$$x' = f(expn_1,\ldots,expn_m);$$

at π', R remains correct. (No assertion or assumption should initially be present at the place following this newly inserted statement.)

Note that the conditions $[f]E \vdash P^g$ and $[f]E \vdash P_1^\pi$ depend only on the set of functions of R, not on the main block of R.

Application of transformation (4.e) will often be facilitated if the following simple proofrule is used first.

(4.f) Let R be a praa, let f, g be functions in R, π a place in g. Suppose that no sequence of function calls can lead directly or indirectly from f to g. Let E be the set of assumptions of R. Then we have $[f]E \vdash P^g$ and $[f]E \vdash P_1^\pi$ for any predicates P^g at g and P_1^π at π respectively.

It will often be appropriate to apply rule (e) in the context

(13) $\beta_-, \langle x_1,\ldots,x_n\rangle = \langle u_1,\ldots,u_n\rangle$ <u>where</u> $C(u_1,\ldots,u_n,vars),\beta_+$

thus replacing the <u>where</u> statement by one or more function calls. The following rule, easily derived from (4.e), facilitates this kind of application.

(4.g) Let R be a valid praa with assumptions E. Suppose that f is a function of R having m parameters, and that f allows entry. Suppose that $expn_1,..,expn_m$ are expressions in the variables *vars* of R, and that in R there exist assertions

P^f at f and $P_1^{\beta_-}$ at β_- such that

$$(14)\quad E \vdash ((\forall r)\ P_1(vars)\ \&\ P(r, expn_1,\ldots,expn_m) \Rightarrow C(r(1),\ldots,r(n),vars))^{\beta_-}$$

Then if x' is a variable not otherwise occurring in R, the where statement in (13) can be replaced by a function call and a sequence of assignments, giving

$$\beta_-, x' = f(expn_1,\ldots,expn_m);\ x_1 = x'(1);\ \ldots,\ x_n = x'(n);,\ \beta_+$$

Initially, no assertions or assumptions should be present at any of the places introduced when this replacement is made.

To justify this transformation, we can reason as follows: By rule (4.e), a function call $x' = f(expn_1,\ldots,expn_m)$ can be inserted immediately following β_-. This introduces a new place π immediately following the function call. By the function call proofrule (i) of Section (2.b) above, the assertion

$$\vdash (P_1(vars)\ \&\ P(x', expn_1,\ldots,expn_m))^{\pi}$$

holds at π. Hence, using (14), it follows that

$$\vdash (C(x'(1),\ldots,x'(n),vars))^{\pi}$$

also holds at π. Rule (1.b) now tells us that the where statement in (13) can be replaced by the set of assignments $x_1 = x'(1);\ \ldots,\ x_n = x'(n);$ which completes the proof that transformation rule (4.g) preserves validity. Q.E.D.

A general remark on praa transformation rules.

Each of the transformation rules stated in the present section describes a procedure which manipulates the text of praas and which has the property that if its input is a correct praa then its output is also a correct praa. We can think of these procedures as collectively constituting the internal procedure library of an implemented praa-verification/ modification system VM whose essential property is that it will never accept a praa unless it is correct, and never modify a praa in a manner spoiling correctness. Given the initial procedure library of VM, other procedures which preserve correctness can of course be devised by compounding the procedures of VM. However, there exists a more general, and sometimes more efficient and convenient method for extending the set of rules for correct-praa transformation which a system like VM will admit. Namely, a procedure which recognizes the boolean formula CORR(t) expressing the statement 't is the text of a correct praa' can be made part of VM; then, whenever VM has been used to prove the correctness of a praa R with the single assumption $\models$ CORR(t) at its entry place and carrying the assertion $\vdash$ CORR(t) at its exit place, the procedure defined by R can be accepted into VM. We call this 'meta-rule', which allows us to expand VM's internal procedure library, *the metamathematical extensibility rule*. Various technical issues connected with this rule and with related meta-rules applying to proof-checker programs will be discussed in a subsequent paper.

3. Praa-Manipulation Illustrated.

a. Successive Refinement.

In the present section, we shall illustrate the use of the rules for correct-praa modification and combination stated in the preceding section, specifically by transforming the sort-by-counting praa considered in sections 1 and 2 into a series of more and more efficient forms expressed at steadily lower language levels, until finally we develop a correct praa embodying the same algorithm but expressed entirely in an assembly language. However, before doing so, it is appropriate both to comment on the intent of the sequence of examples to be given, and also to give general descriptions of a number of useful praa manipulation techniques.

The praa manipulations described below are best regarded as scenarios depicting interactions with a hypothetical, implementable but of course still unimplemented, interactive, 'correctness-preserving editor'. Such a system will permit its user to make various changes in an initially given praa. The most elementary of these will be usable without constraint, but in the case of less elementary manipulations, justifying assertions will be demanded.
These assertions are groupable into two pragmatically distinguishable classes: on the one hand, those whose proofs are simple and stereotyped enough to be generated rapidly by a program analysis system not much more complex than program analyzers/optimizers like those currently under development; on the other, assertions with proofs complicated enough for recourse to a full-fledged proof-verification program to be necessary. Our aim in taking the approach we do is to ensure that all, or at any rate most, of the proofs that need to be supplied in support of praa manipulation belong to the first (superficial) rather than the second (deeper) class. The illustrations to be given will hopefully create the impression that this is indeed the case; however, the reader should be warned that it is all too easy for an author to create this impression by casting formal arguments into natural language

forms which mask the irritating complications that arise when one must really communicate with an implemented system.

Wishing to emphasize the fact that all the praa transformations that we carry out are fully justified by the rules stated in the preceding sections, we shall pursue these transformations to quite a low level of detail, indeed to a level which could be left to a compiler, especially to a very high level compiler able to accept general directives concerning transformations to be applied and to refuse service when acceptable forms of the assertions needed to justify these transformations are not available. Of course, when we can do so it is even better to rely on an automated transformation system of this type than to use a praa manipulation system directly, since a system able to generate a class of program transformations will be able *ipso facto* to assume full responsibility for justifying these transformations.

Certain commonly occurring and quite general transformation techniques are of particular importance, and it is well to describe them before proceeding to any more specific discussion. To begin with, note the obvious fact that if we strengthen the assumptions of a praa R by adding new assumptions and by adding extra clauses P_1 which convert particular initial assumptions $\models P^{\pi}$ to stronger forms $\models (P \ \& \ P_1)^{\pi}$, correctness will be preserved. The clauses P_1 can involve free variables, i.e. new set-theoretic objects, which do not appear in the original form of R. Now it may be that the added clauses P_1 imply the clauses P originally present; then $P_1 \Rightarrow P \ \& \ P_1$, so that the hypotheses $\models (P \ \& \ P_1)^{\pi}$ can be replaced by $\models P_1^{\pi}$, and the resulting praa will still remain correct. The newly added assumptions P_1 may imply identities and inclusions not available in the original praa R, and these identities and inclusions can allow certain of the expressions originally present in the code text and assertions of R to be replaced by other expressions which involve the new objects, i.e. those appearing initially in the clauses P_1, rather than the variables

originally present in R. If these old variables are not assignment-targets in R then this replacement can be carried out systematically, and we may even succeed in eliminating certain of the old variables completely from all the statements and assertions of R. When this happens, the only remaining occurrences of these variables x will be in the assumption clauses P_1 themselves. If such a point is reached, and assuming now that the only assumptions in which x appears are assumptions at the entry place π of R, we can hope to apply the following

Lemma (Variable Dropout). If a variable x of R appears in the assumptions $\models P^{\pi}$ of a praa R at only the entry place π of R, and never appears either in the program text or the assertions of R, then R will remain correct if each predicate $P = P(x, other\_vars)$ appearing in such an assumption P^{π} is replaced by $(\exists u)\ P(u, other\_vars)$.

Proof: Let R' be the praa obtained by modifying the assumptions $\models P$ in the manner indicated, and let c' be a computation satisfying the assumptions of R. Let the 'other_vars' appearing in our hypothesis be $x_1,\ldots,x_n$, let $o_1,\ldots,o_n$ be the values assigned to these variables by the first state of c', and let o be any set-theoretic object satisfying $P(o,o_1,\ldots o_n)$. Then if we modify each state s' of c' by constructing a new state s for which $s(x_j) = s'(x_j)$ for all $j = 1,\ldots,n$ and $s(x) = o$, we obtain an R-computation c which clearly satisfies all the assumptions of R. Thus c must satisfy all the assertions of R, which clearly implies that c' satisfies all the assertions of R'. Q.E.D.

One common way of applying the technique just outlined will be to select certain variables $x_1,\ldots,x_n$ which appear in R but which R does not modify, then to introduce m new variables $r_1,\ldots,r_m$ and n set-theoretic functions $\phi_j(r_1,\ldots,r_m)$, $j = 1,\ldots,n$, and to supplement the initial assumptions of R by the clauses $x_j = \phi_j(r_1,\ldots,r_m)$, $j = 1,\ldots,n$. Then clearly

every occurrence of x_j in a statement, assumption, or assertion can be replaced by an occurrence of $\phi_j(r_1,\ldots,r_m)$; and after this is done the only appearance of any of the variables x_j will be in the clauses $x_j = \phi_j(r_1,\ldots,r_m)$ themselves. But then the variable dropout lemma stated above allows these clauses to be replaced by $(\exists\, x)(x = \phi_j(r_1,\ldots,r_m))$. Since $\phi_j(r_1,\ldots,r_m) = \phi_j(r_1,\ldots,r_m)$ this existentially quantified equality is universally true, and can therefore be dropped as an assumption. The overall effect of this is simply to substitute $\phi_j(r_1,\ldots,r_m)$ for x_j at each of its occurrences. This technique, which stands in exact analogy to the technique of generating new predicate theorems by substituting arbitrary terms for the free variables of old terms, may be called the method of *substitution* in a praa.

What may be called the technique of *representation* goes beyond the substitution method that we have just described. This technique can be described as follows. Consider some sub-collection $x_1,\ldots,x_n$ of the variables of a praa R, and consider all the expressions, occurring either in program statements and in assumptions/assertions, in which these variables appear. These expressions will have the form $e(x_1,\ldots,x_n,\text{other\_vars})$. Some of these expressions will appear in assignments of the form

(1) $\quad x_j = \text{expn}(x_1,\ldots,x_n, \text{other\_vars})\ ,\quad j = 1,\ldots,n;$

others will appear in other contexts. We call expressions which appear in a context (1) *active expressions*, and call all other expressions $\text{expn}(x_1,\ldots,x_n,\text{other\_vars})$ *passive expressions*. Suppose that we can find some collection $r_0,r_1,\ldots,r_m$ of auxiliary variables, together with

(i) n set-theoretic functions
$\phi_1(r_1,\ldots,r_m,\text{other\_vars}),\ \ldots,\ \phi_n(r_1,\ldots,r_m,\text{other\_vars})$;

(ii) for each passive expression $e(x_1,\ldots,x_n,\text{other\_vars})$ in which $x_1,\ldots,x_n$ appears, another expression $e'(r_1,\ldots,r_m,\text{other\_vars})$ such that either

(ii.a) $e(\phi_1(r_1,\ldots,r_m,\text{other\_vars}),\ldots$
$\phi_n(r_1,\ldots,r_m,\text{other\_vars}),\text{other\_vars})) =$
$e'(r_1,\ldots,r_n,\text{other\_vars})$ holds identically; or

(ii.b) a proposition $P(x_1,\ldots,x_n,\text{other\_vars})$, available in R immediately before the place of appearance of the expression e, such that the implication

(2) $P(\phi_1(r_1,\ldots,r_m,\text{other\_vars}),\ldots,\phi_n(r_1,\ldots,r_m,\text{other\_vars}),\text{other\_vars})$
$\Rightarrow e(\phi_1(r_1,\ldots,r_m,\text{other\_vars}),\ldots,\phi_n(r_1,\ldots,r_m,\text{other\_vars}),\text{other\_vars}$
$= e'(r_1,\ldots,r_m,\text{other\_vars})$

holds universally.

(iii) For each active expression appearing in a context

(3) $x_j = e(x_1,\ldots,x_m,\text{other\_vars})$,

a set of m set-theoretic expressions $e_j(r_1,\ldots,r_m,\text{other\_vars})$, $j = 1,\ldots,m$, and an assertion $(P(x_1,\ldots,x_m,\text{other\_vars}))^{\pi}$ known to be true at the place π immediately preceding the assignment (3), such that the implications

(4a) $P(e_1(r_1,..r_m,\text{other\_vars}),\ldots,e_m(r_1,..\ r_m,\text{other\_vars}),\text{other\_vars})$
$\Rightarrow \phi_i(e_1(r_1,..r_m,\text{other\_vars}),\ldots,e_m(r_1,..r_m,\text{other\_vars}),\text{other\_vars})$
$= \phi_i(r_1,\ldots,r_m,\text{other\_vars})$ for $1 \leq i \leq n$, $i \neq j$

and

(4b) $P(e_1(r_1,..r_m,\text{other\_vars}),\ldots,e_m(r_1,..r_m,\text{other\_vars}),\text{other\_vars})$
$\Rightarrow \phi_j(e_1(r_1,..r_m,\text{other\_vars}),\ldots,e_m(r_1,\ldots,r_m,\text{other\_vars}),\text{other\_vars}$
$= e(\phi_1(r_1,\ldots,r_m,\text{other\_vars}),\ldots,$
$\phi_n(r_1,\ldots,r_m,\text{other\_vars}),\text{other\_vars}$

hold universally.

If this is the case, we shall say that the functions ϕ_j are *representing functions* which *reduce* the expressions $e(x_1,\ldots,x_n,\text{other\_vars})$ to the expressions $e'(r_1,\ldots,r_m,\text{other\_vars})$ in the *context defined by the assertions* P.

Suppose that variables r_i and representing functions ϕ_j as described above exist. Suppose that we modify R, first by using the shadow-variable rule to introduce assignments

(5) $r_0 = \langle e_1(r_1,\ldots,r_m,\text{other\_vars}),\ldots,e_m(r_1,\ldots,r_n,\text{other\_vars})\rangle;$
$r_1 = r_0(1);\ \ldots;\ r_m = r_0(m)$

immediately before each assignment (3); and after this, by introducing temporary assumptions

(6) $\models x_i = \phi_i(r_1,\ldots,r_m,\text{other\_vars})\ ,\quad 1 \leq i \leq n$

everywhere. Because of the inserted assignments (5) and the identities (4a) and (4b), none of the assignment sequences (3, 5) spoil the assumptions (6); hence if we include (6) among our input assumptions, then at every place other than the entry place of R, the assumptions (6) can become assertions. Using these assertions we can now replace every remaining passive expression

(7) $e(x_1,\ldots,x_n,\text{other\_vars})$

by

(8) $e'(r_1,\ldots,r_m,\text{other\_vars})$.

After this is done, all assignments to $x_1,\ldots,x_n$ can be eliminated by the shadow variable rule, and the input assumptions (6) can be existentially quantified using the variable dropout lemma stated earlier in the present section and then dropped since once quantified they become true.

The overall effect of this procedure is to replace each assignment operation (3) by an assignment sequence (5) or something equivalent to it, and to replace every passive expression (7) by the substituted expression (8). As already

noted, we shall refer to this entire procedure as praa *transformation by representation*, specifically by use of the representing functions $\phi_1,\ldots,\phi_m$, plus the various reduced expressions e', and the various assignment expressions $e_1,\ldots,e_m$.

In most cases, each representing function ϕ_j will depend on only a very few of the newly introduced variables r, and will be independent of all *other_vars*; thus the full expression complexity suggested by formulae (1 - 5) will rarely be faced.

The logical propositions (1), (4a), (4b) needed to justify the transformations described in the last few pages may be called *representation lemmas*. The proof of such lemmas may be considered to be the stuff of a fully formalized variant of the familiar 'data structures' course. Transformation by representation can be organized into a process resembling compilation, roughly as follows. Each member of the collection of justifying identities (2), (4a), (4b) available for representational use can be assigned some designating mnemonic or phrase, e.g.

'represent x_1 by a list r_1 '

'represent x_2 by a B-tree r_2 ' .

An appropriate correctness-preserving compiler can then check the consistency of such declarations and coding hints, and either reject them as hopelessly inconsistent, demand that supporting assertions $P(x_1,\ldots,x_n,\text{other\_vars})$ be attached to the praa R to be transformed, or deduce these assertions itself and proceed immediately to produce an appropriately transformed praa R'.

To wind up this point, note that identities (4a), (4b) which define the representation of indexed assignment operators $x_j(\text{expn}_1) = \text{expn}_2$ will obviously find particularly frequent use.

As the code text of a praa R is manipulated to bring it into more and more efficient forms, there will come a point at which we pass over from the use of set-theoretic objects having pure value semantics to the use of representing objects which can be imbedded easily in a comprehensive memory array having hardware-like logical characteristics, and also to the use of code sequences which for the sake of efficiency reflect the conventions of pointer rather than of value semantics. We shall now say a few words about the typical logical arguments which justify such a transition.

A pointer-oriented data structure S can be regarded as an indefinitely expansible set of objects, the *pointers*, together with a mapping *pto* which sends each pointer into a vector v whose components are either elementary objects (e.g. integers) or further pointers and also together with an auxiliary set E of pointers, which we shall call the *entrances* of the data structure S. Each of the set-theoretic objects which such a structure represents will be defined by some abstractly specified function $\rho(p, pto)$ of pointers $p \in E$ and of the comprehensive mapping *pto*. The functions ρ used in this way will ordinarily have inductive definitions; from the set-theoretic point of view, it is therefore perhaps best to introduce auxiliary mappings of (p,pto) onto sequences f = f(n) defined by inductive and initial conditions

(9) $f(n+1) = G(p,pto,f(n))$ and $f(1) = H(p,pto)$,

where G and H have elementary definitions. For example, if we consider a conventional LISP-like structure in which the vectors v are always pairs having components which are either elementary objects or pointers, then a particularly important sequence connected with each entry p will be that defined by

(10) $f(1) = p$; $f(n+1) =$ if $\neg$is_pointer((pto(f(n))(2)) then nil
else (pto(f(n)))(2) .

Here is_pointer(x) is assumed to be an elementary predicate true for objects which are pointers and false otherwise; and nil is a special pointer used in the familiar LISP manner. If we take G and H to be given and designate the sequence f defined by (9) (or by its special case (10)) as F(p,pto), then the set-theoretic object ρ(p,pto) represented by p and *pto* will often be definable in a straightforward manner using F(p,pto). For example, if we consider a list-like data structure for which (10) is appropriate, and if lists are being used to represent sets, we might very well have

(11) $\rho(p,pto) = \{(pto(f(n)))(1) \mid f = F(p,pto)\ \&\ f(n) \neq nil\}$.

If we proceed in this way, then each inductive relationship between sequences F(p,pto) will yield some useful relationship between the set-theoretic objects defined using these sequences. For example, in the case (10) we have

(12a) $\rho(p,pto) = nl$ if and only if $f(1) = nil$

and

(12b) $p \neq nil \Rightarrow \rho(p,pto) = \rho((pto(p))(2),pto) \cup \{(pto(p))(1)\}$,

which among other things can be used to justify the conversion of

```
∀x ∈ ρ(p,pto);          p' = p;
  code                  while p' ≠ nil;
end ∀;          into      x = pto(p')(1);
                          code
                          p'= pto(p')(2);
                        end while;
```

In cases involving logical relationships too complex to be described conveniently by simple sequences f having definitions like (9), it may be appropriate to consider

multi-sequences f' = f(<n_1,...,n_k>) which depend on *pto* via more complex inductions, e.g.

(13) f(<n_1,...,n_{k-1},n_k+1>) = G(p,pto,k,f(<n_1,...,n_k>))

f(<n_1,...,n_{k-1},1>) = H(p,pto,k)

f(<1>) = H(p,pto,1) .

To manipulate a data structure like S we shall use code sequences *cs* containing assignments pto(p) = expn, and also assignments of the somewhat more special form (pto(p))(j) = expn, which is an abbreviation for

(14) t = pto(p); t(j) = expn; pto(p) = t;

where t is a variable not otherwise occurring. The code sequences *cs* used to manipulate S will always be chosen so as to preserve the validity of certain predicates of the form P(E,pto). These predicates, which may be called the *invariant predicates* of the data structure, will therefore always be available as hypotheses for use in deducing properties both of sequences (9) and (13) and of set-theoretic objects ρ(p,pto) related to such sequences. Moreover, the code sequences *cs* used to modify a data structure will generally be chosen so as to leave the values ρ(p,pto) invariant for all but a finite subcollection $p_1,...,p_n$ of the members of E, and for these p_j one will generally have proved identities

(15) $\rho(p_j,pto') = expn_j(\rho(p_1,pto),...,\rho(p_n,pto))$

which relate each set-theoretic object calculated using a modified mapping *pto'* to the set-theoretic objects ρ calculated using the unmodified *pto* from which pto' was obtained.

Once again we note that representation lemmas which establish properties of sequences f defined by recursions (9), (13), or which assert that particular code sequences preserve important invariant properties of data structures, or which

imply identities (15), only need to be proved once and can then be used to improve the efficiency of a broad variety of praas.

Up to this point we have been describing arguments which can be used to justify the transformation of programs written using set-theoretic constructs into programs in which only vector-and-pointer constructs appear. However, we have assumed that vectors of indefinite length are available; in pragmatic terms, this implies an environment which is fully 'garbage collected'. Now we shall sketch the techniques which can be used to transform praas whose code text has this character into equivalent praas which only involve 'static' arrays. To do this, we single out some finite subcollection $p_1,\ldots,p_n$ of E, introduce associated 'size integers' $k_1,\ldots,k_n$ and 'address integers' $pa_1,\ldots,pa_n$, as well as a comprehensive memory vector MEM, and insert the temporary assumptions

$$(16) \qquad pto(p_j) = MEM(pa_j+1:\ pa_j+k_j)\ , \quad j = 1,\ldots,n,$$

where MEM(a:b) designates the 'slice' of MEM running from its i-th to its j-th components. The shadow variable rule is then used to introduce an assignment

$$(17) \qquad MEM(pn_j + x) = expn$$

immediately before each occurrence of $(pto(p_j))(x) = expn$ in the praa R being manipulated. If these indexed assignments are the only operations in R that can change $pto(p_j)$, if the quantities k_j are never changed and we assume that on entry to R we have $k_j \geq 1$ and $pa_j+k_j \leq pa_{j+1}$ for $j = 1,\ldots,n-1$, and if we can show that every x occurring in an assignment $(pto(p_j))(x) = expn$ satisfies $1 \leq x \leq k_j$, then each assignment (17) will change only $MEM(pa_j+1:pa_j+k_j)$ and not $MEM(pa_2+1:pa_i+k_i)$ for any $i \neq j$. This allows the assumptions (17) to become assertions. Using these assertions, we can replace every use of $pto(p_j)$ by a use of $MEM(pa_j+1:pa_j+k_j)$,

and if this is done systematically it may be possible to eliminate all uses of p_j , and then to eliminate the variables p_j themselves by the shadow variable rule.

So much for generalities; we turn now to an illustrative example. In the discussion which now follows, we will refer repeatedly to the transformation rules stated in the preceding section, sometimes unspecifically, but sometimes specifically by number. Our manipulations will make use of the general techniques sketched in the preceding pages, but in order to emphasize foundations we will sometimes use first principles to justify these manipulations instead of simply referring to a general technique by name. Our example is the sort-by-counting praa developed in the preceding section, but before turning to its detailed manipulation it is appropriate to introduce some additional notation and a few simple auxiliary praas. We take the notation

(18)

```
∀x ∈ s;
   code
end ∀;
```

to abbreviate

(19)

```
s⁻ = nℓ;
while s ≠ s⁻;
   x = ∋(s - s⁻);
   code;
   s⁻ = s⁻ ∪ {x};
end while;
```

Proof rules for (18) can easily be deduced by expanding (18) into (19). The correct praa

(20) $\triangleright \models$ set(s) & g ∈ sing_val_maps(s,booleans)

```
n = 0;
∀y ∈ s;
   if g(y) then
      n = n+1;
   end if;
end ∀;
```

$\vdash$ n = #{x ∈ s | g(x)} $\triangleleft$

which of course describes the standard technique of counting by iteration, uses this notation. The correctness of this familiar praa is easily proved by introducing the obvious auxiliary assumption $\models$ n = #{x ∈ s^- | g(x)} at the head of the while loop which appears when the ∀-loop in (20) is expanded.

By substituting {<y, f(y)<f(x)∨f(y)=f(x)&place(y)neΩ>,y∈s} for g in (20) and eliminating redundant assumptions, we obtain the correct praa

(21) $\triangleright \models$ set(s) & f ∈ sing_val_maps(s,reals) & x ∈ s & place ∈ sing_val_maps

```
n = 0;
∀y ∈ s;
   if f(y) < f(x) ∨ f(y) = f(x) & place(y) ne Ω then
      n = n+1;
   end if;
end ∀;
```

$\vdash$ n = #{y∈s | f(y)<f(x)∨f(y)=f(x)&place(y) ne Ω} $\triangleleft$

Since the praa only modifies variables which do not appear in the counting sort praa R_1 shown on page 15, we can insert it into that praa, and then use the equality substitution rule to obtain the following correct praa, which we call R_2:

```
▷ ⊨ set(s) & f ∈ sing_val_maps(s,reals)
   place = nℓ;
   ∀x ∈ s;
      n = 0;
      ∀y ∈ s;
         if f(y) < f(x) ∨ f(y) = f(x) & place(x) ne Ω then
            n = n+1;
         end if;
      end ∀;
      place(x) = n+1;
   end ∀;
   |– place ∈ one_one_maps(s,integers)
                        & range(place) = {n, 1≤n≤ #s}
                & (∀x,y ∈ s)f(x) < f(y) ⇒ place(x)<place(y) ◁
```

If a set s has the form s = {n, p ≤ n ≤ q}, then any loop of the form (18), i.e. (19), occurring in a correct praa can be transformed as follows: Introduce a new variable j, initially a shadow variable, and introduce the assignment j=p immediately before the while loop of (19) and j=j+1 at the end of this loop; also introduce the assumption $\models s\text{-}s^- = \{n, j \le n \le q\}$ at the head of the loop. With this assumption, the selection statement $x = \ni (s\text{-}s^-)$ can be replaced by the single statement code-block x = j. (Cf. Rule (1.e)). Once this is done, the assumption $\models s - s^- = \{n, j \le n \le q\}$ can be degraded to an assertion, and the test while $s \neq s^-$ can be replaced by while $j \le q$. If no essential uses of s and s^- remain, the Shadow Variable Rule can be applied to remove all assignments to s and s^-. The s will have the assumed form if we strengthen our input assumption to read s = domain(f) & vector(f) & (range(f) ⊆ reals), where vector(f) means that f is a set of pairs, and that domain(f) = {n,1≤n≤#f}.

After this transformation, and resurrecting an assertion not always shown earlier, the preceding praa takes on the form

```
▷ ⊨ vector(f) & range(f) ⊆ reals
  place = nℓ;
  x = 1;
  while x ≤ #f; ⊢ domain(place) = {n, 1 ≤ n < x}
     n = 0;
     y = 1;
     while y ≤ #f;
        if f(y) < f(x) ∨ (f(y) = f(x) & y < x) then
           n = n+1;
        end if;
        y = y + 1;
     end while;
     place(x) = n+1;
     x = x + 1;
  end while;
  ⊢ vector(place) & #place = #f & range(place) = {n,1≤n≤#f}
                  & (1≤∀x,y≤#f)f(x)<f(y)⇒place(x)<place(y)  ◁
```

We shall call this praa R_3. Suppose that we now introduce a new variable *place'*, make the initial assumption ⊨ vector(place') & #place' = #f, and then insert the assignment place'(x) = expn immediately before the assignment place(x) = expn in the praa R_3. After these changes, it is easy to see that the temporary assumption

⊨ (∀z ∈ dom(place)) (place'(z) = place(z))

can be degraded to an assertion. Since y ≤ #f implies y ∈ domain(place'), we have vector(place') & #place' = #f throughout. This allows *place'* to replace *place* in the final assertion of our praa, which in turn allows the removal of *place* by the shadow variable rule. The overall effect of all this is simply to eliminate the second line of R_3, to replace *place* by *place'* in a few other lines,

and to drop the assertion $\vdash$ domain(place) = {n, 1 $\leq$ n < x}, which no longer plays any role. We shall use R_4 to denote the praa which results from these changes to R_3.

Now we start to move to a statically compiled form of our praa. This can be done by introducing a new vector, called MEM (an assumed overall memory array), by assuming initially that k = #f, that f = MEM(fadr+1:fadr+k), where fadr is some integer, and that padr is another integer, where padr $\geq$ fadr+k, so that the value of MEM(fadr+1:fadr+k) is unchanged by any assignment of the form MEM(j) = *expn* with padr+1 $\leq$ j $\leq$ padr+k. We will also make the initial assumption that place' = MEM(padr+1:padr+k), which subsumes our earlier assumptions concerning place'.

To deduce 1 $\leq$ x $\leq$ #f in the last preceding form R_4 of our praa is easy, and thus even if we insert the assignment MEM(padr+x) = n+1 immediately before the statement place(x) = n+1 in R_4, the temporary assumption $\models$ f = MEM(fadr+1:fadr+k) is never spoiled, and can become an assertion. But with this assignment inserted, it is clear that the assumption $\models$ place' = MEM(padr+1:padr+k) is not spoiled either. Thus references to MEM can replace all references to *f* and *place'*, allowing these two variables to be suppressed using the shadow variable rule. This casts our praa into the following form, which we call R_5:

```
    ⊨≡ vector(MEM) & k∈integers & k≥1 & padr∈integers & fadr∈integers
         &  padr≥fadr+k
         & range(MEM(fadr+1:fadr+k)) ⊆ reals
    x = 1;
    while x ≤ k;
       n = 0;
       y = 1;
       while y ≤ k;
          if (MEM(fadr+y) < MEM(fadr+x) ∨
                          MEM(fadr+y) = MEM(fadr+x) & y < x) then
             n = n+1;
```

```
        end if;
        y = y + 1;
    end while;
    MEM(padr + x) = n + 1;
    x = x + 1;
end while;
⊢ vector(MEM) & range(MEM(padr+1:padr+k)) = {n, 1 ≤ n ≤ k}
     & (1 ≤ ∀x,y ≤ k) MEM(fadr+x) < MEM(fadr+y) ⇒
                          MEM(padr+x) < MEM(padr+y)
```

The praa R_5 has a semantic level quite close to that of assembly language. To bring it down to what is recognizably a form of assembly language, we have only to expand the *if* and *while* statements in R_4 into their primitive forms, introduce additional temporary variables to decompose all compound expressions into their elementary parts, and replace all operations not available in assembly language by assembly-language code blocks which have equivalent effect on the variables of R_5. We can regard our target assembly language as a collection of code blocks, of unknown internal structure, which perform elementary operations on a fixed collection of variables, let us say to be specific X1,...,X16; the way in which we write assembly langage operations is illustrated by the typical case Xi = Xj + Xk. We assume that every assembly language operation either succeeds or aborts (perhaps simply by never finishing), and that if it succeeds it has precisely the same effect as some corresponding mathematical operation. By making this simple (although sweeping, and not entirely realistic) assumption, we rule out all those complications of proof which ensue if assembly language operations are explicitly allowed to differ in certain marginal cases from the mathematical operations which they normally represent. (For a study of these important but irritating cases, see [Sites, 1974]).

We also assume that operations MEM(Xi) = Xj and Xj = MEM(Xi) are available at the assembly language level. In order to capture at least some of the typical flavor of assembly language code, we shall assume that no boolean operations are available, and that the only available conditional transfer operations have the usual form

if Xk ≥ 0 then go to L	if Xk < 0 then go to L
if Xk ≤ 0 then go to L	if Xk > 0 then go to L
if Xk = 0 then go to L	if Xk = 0 then go to L .

Since all the rest is quite ordinary, we will consider only the treatment of the statements

```
(20) if MEM(fadr+y) < MEM(fadr+x) v
                (MEM(fadr+y) = MEM(fadr+x) & y < x) then n = n+1;
     end if;
```

appearing in R_5. We shall initially assume that fadr+x and fadr+y have been 'loaded' into two registers which we designate symbolically as Xfx and Xfy; that is, we make the temporary assumption $\models$ Xfx = fadr+x and $\models$ Xfy = fadr+y immediately prior to the code block (20). Then treating others of the special register variables Xj as shadow variables, we introduce assignments

```
Xcy = MEM(Xfy);   Xcx = MEM(Xfx)
```

immediately prior to (20). This establishes equalities which allow us to rewrite (20) as

```
(21)    if Xcy < Xcx v Xcy = Xcx & Xfx < Xfy then n = n+1;

        end if;
```

We now manipulate (21) in various obvious ways using the go to rules (2.a-2.h) repeatedly, give the variable n the new name Xn, and also make use of a few other special register variables Xj, introducing appropriate assignments for those we use, we can rewrite (21) in the following equivalent form, which is of course

typical of what a compiler might produce:

```
      Xcxmy = Xcx - Xcy;
      if Xcxmy   0 then go to Add;
      if Xcxmy ≠ 0 then go to Skip;
      Xcxmy = Xfx - Xfy;
      if Xcxmy ≥ 0 then go to Skip
Add:  Xn = Xn + Xone                    /* where ⊨ Xone = 1 */
Skip: ...
```

By applying the same familiar technique to the remainder of the praa R_5 , we can easily convert it to an equally correct praa, whose assumptions and assertions will retain a high-level, set theoretic form, but all of whose statements belong to assembly language. Details of this are left to the reader.

b. A Class of Root Praas Derivable by Transformation.

Up to the present point we have taken the notion of 'root praa' as fundamental and have described techniques for combination and manipulation of root praas. In the present section, we shall describe a class of root praas which can be derived transformationally from underlying mathematical facts of a particular form, thereby penetrating somewhat more deeply into the questions of program genesis that define the pragmatic issues which a verification technology must face. We begin by noting that, although set theory makes available a language of powerful global dictions, programming rules out direct leaps between global totalities, so that *to devise an algorithm one must rely either on a technique of systematic extension or on a pattern of decomposition.* Tc be specific, suppose that some set-theoretic function (or functions) F depending on a composite set-theoretic object (or objects) x has been defined mathematically, leaving us with the problem of developing an acceptable algorithm for the computation of F(x). First consider algorithms which are based upon the technique of extension. Such algorithms typically introduce some class of auxiliary objects z, together with an auxiliary transformation $z \rightarrow T(x,z)$ which, if applied repeatedly to an initial $z_0 = S(x)$, will eventually produce some z from which F(x) can be calculated directly. Iteration of T may, for example, enlarge some set, extend some mapping, or progressively 'tighten' some condition satisfied by z until the condition defining F(x) comes to be satisfied. For verification of algorithms of this kind, direct use of the praa proof and transformation methods described in preceding sections is normally just about as comfortable a technique as can be devised, though of course the finding of 'minimal' algorithm forms which facilitate the task of verification but from which the conventional versions can be derived by easy transformations is a task requiring careful thought.

In the case of algorithms which proceed by decomposition, a more interesting transformational approach is possible. To calculate F(x) using such an algorithm, we factor F into a product of simpler maps F_j, some of which may decompose x (or perhaps objects y calculated from x) into a collection of simpler 'subparts' $F_j(x) = \langle G_j^{(1)}(x),\ldots,G_j^{(k)}(x)\rangle$. A given factorization may only be valid when a particular logical condition P_1 is satisfied; if P_1 is false but some other relevant condition is satisfied, a different condition may apply. In general therefore a decompositional algorithm for the computation of F will rest upon a relationship of the general form

$$
\begin{aligned}
(1)\qquad F(x) = {} & \text{if } P_1(x) \text{ then } F_1^{(1)}(\ldots F_{n_1}^{(1)}(x)) \\
& \text{else if } P_2(x) \text{ then } F_1^{(2)}(\ldots F_{n_2}^{(2)}(x)) \\
& \ldots \\
& \text{else if } P_k(x) \text{ then } F_1^{(k)}(\ldots F_{n_k}^{(k)}(x))
\end{aligned}
$$

Note that if the set-theoretic function F will only appear in the context $\ni F(x)$, then it is sufficient to have inclusion (of right by left) in (1) rather than identity.

As everybody knows, a relationship of the form (1) can be used to compute F even if F appears in several places on the right-hand side of (1), provided only that the argument values to which F is supplied are in some sense simpler than the initial argument x. When this is the case, the relationship (1) is recursive and can be converted directly into a recursive program for the calculation of F. Now, any recursive program can be converted into an iteration by introduction of an explicit stack of appropriate form. Moreover, at each recursive level we only need to stack information which cannot be recovered conveniently from the information passed down

to the next recursive level. In general, any recursion for which it is possible to keep track of stack contents in a simple way can be converted to an iteration; and as a matter of fact numerous useful recursive schemata having this property have been catalogued and exploited by [Walker and Strong 1972], [Burstall and Darlington 1975]. We list various significant items from their catalog, assuming at first that the function F is single-valued.

(i) Assume that

(2) $$F(x) = \text{if } P_1(x) \text{ then } F(M(F(M \ldots F(N_1(x) \ldots)))) \text{ else if } P_2(x) \text{ then } F(M(F(M \ldots F(N_2(x) \ldots)))) \ldots \text{ else if } P_k(x) \text{ then } F(M(F(M \ldots F(N_k(x) \ldots))))$$

with the map M occurring m_j times if P_j is satisfied. Then (as observed by Walker and Strong) $y = F(x)$ converts (if x is dead) to

(3)
```
        k = 1;
        go to  Lstart;
        while k > 0;
           x = M(x);
Lstart:    while P1(x) v ... v Pk(x);
              k = k+ if P1(x) then m1 else if...else if Pk(x)
                                                 then mk;
              x = if P1(x) then N1(x) else if...else if Pk(x)
                                                 then Nk(x);
           end while;  |- ¬(P1(x) v ... v Pk(x))
           x = F(x);
           k = k-1;
        end while;
        y = x;
```

If all the integers m_j are zero, the outer loop is superfluous, and (3) simplifies to

(4) $\underline{\text{while}}\ P_1(x) \vee \ldots \vee P_k(x);$

$x = \underline{\text{if}}\ P_1(x)\ \underline{\text{then}}\ N_1(x)\ \underline{\text{else}}\ \underline{\text{if}} \ldots$

$\underline{\text{else}}\ \underline{\text{if}}\ P_k(x)\ \underline{\text{then}}\ N_k(x);$

$\underline{\text{end}}\ \underline{\text{while}};$

$y = F(x);\ \vdash \neg(P_1(x) \vee \ldots \vee P_k(x))$

(ii) Assume that

(5) $F(x) = \text{if } P_1(x) \text{ then } M_1(H_1(x), F(N_1(x))) \text{ else if } \ldots$

$\text{else if } P_k(x) \text{ then } M_k(H_k(x), F(N_k(x)))\ ,$

that all the mappings N_j have inverses N_j^{-1}, that the predicates $P_j(N_j^{-1}(x))$ are mutually exclusive, and that, for each x, there exists at most one y in the set generated from x by repeated application of the maps N_j which satisfies $\neg(P_1(y)\ \&\ \ldots\ \&\ P_k(y))$. We call this value V(x). Then y = F(x) converts (if x is dead) to

(6) $xsave = x;$

$x = V(x);\ y = F(x);\ \vdash \neg(P_1(x) \vee \ldots \vee P_k(x))$

$\underline{\text{while}}\ x \neq xsave;$

$x = \underline{\text{if}}\ P_1(N_1^{-1}(x))\ \underline{\text{then}}\ N_1^{-1}(x)\ \underline{\text{else}}\ \underline{\text{if}} \ldots$

$\underline{\text{else}}\ \underline{\text{if}}\ P_k(N_k^{-1}(x))\ \underline{\text{then}}\ N_k^{-1}(x);$

$y = \underline{\text{if}}\ P_1(x)\ \underline{\text{then}}\ M_1(H_1(x), y)\ \underline{\text{else}}\ \underline{\text{if}} \ldots$

$\underline{\text{else}}\ \underline{\text{if}}\ P_k(x)\ \underline{\text{then}}\ M_k(H_k(x), y);$

$\underline{\text{end}}\ \underline{\text{while}};$

This transformation will commonly be applied to cases in which the functions M_j are independent of their first parameters. Application of it will often convert a recursive relationship of the form (5) into an algorithm that works by extension.

Even if the N_j have no inverse, this transformation can still be used if we are willing to search nondeterministically over the sets $N_j^{-1}(x)$.

(iii) Assume that

(7) $F(x) = \text{if } P_1(x) \text{ then } M(H_1(x), G_1(F(N_1(x)))) \text{ else if...}$
$\text{else if } P_k(x) \text{ then } M(H_k(x), G_k(F(N_k(x)))) ,$

and that the two-parameter mapping M is associative. We say that a one-parameter mapping G left-commutes (resp. commutes, resp. right-commutes) with M if $G(M(x,y)) = M(Gx,y)$ (resp. $= M(Gx,Gy)$, resp. $M(x,Gy)$). Suppose that the range $j = 1,\ldots,k$ is divided into three subsets ℓc, c, rc such that G_j left-commutes (resp. commutes, resp. right-commutes) with M for $j \in \ell c$ (resp. $j \in c$, resp. $j \in rc$). Put $crc(j) = \underline{\text{if}}\ j \in \{c, rc\}\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ 0$. Suppose also that all the G_j for which $j \in \{c, rc\}$ commute with each other.

Then $y = F(x)$ converts (if x is dead) to

(8)
```
if ¬(P_1(x) ∨ ... ∨ P_k(x)) then
     y = F(x);   ⊢ ¬(P_1(x) ∨ ... ∨ P_k(x))
else
     n_1 = 0; ...; n_k = 0;
     j = if P_1(x) then 1 else if...else if P_k(x) then k;

     y = H_j(x);
     while P_1(x) ∨ ... ∨ P_k(x);
          newj = if P_1(x) then 1 else...else if P_k(x) then k;
          z = H_newj(x);
          if j ∈ (ℓc∪c) then
               z = G_j(z);
          end if;
```

```
                for m = 1 to k;
                     if m ∈ c then
                          for i = 1 to n_m;
                               z = G_m(z);
                          end for;
                     end if
                end for;
                y = M(y,z); n_j=n_j+crc(j); j=newj; x=N_j(x);
           end while;
           z = G_j(F(x)); ⊢ ¬(P_1(x) ∨ ... ∨ P_n(x))
           for m = 1 to k;
                for i = 1 to n_m;
                     z = G_m(z);
                end for;
           end for;
           y = M(y,z);
```

Frequently encountered special cases of this general schema are $G_j(x) \equiv x$, $j = 1,\ldots,k$, in which case G_j left-commutes with M and the quantities n_j are identically zero, and $M(x,y) \equiv y$, in which case all G_j right-commute with M; also the more special case in which $M(x,y) \equiv y$ and $G_j(x) \equiv G(x)$ for all j.

(iv) Suppose that (7) holds, that the two-parameter mapping M in (iii) is both associative and commutative, and that all the $G_j = G$ are the same and left-commute with M. Suppose also that for each x all the u in the set V(x) defined by the condition $\neg(P_1(u) \vee \ldots \vee P_n(u))$ and by the requirement that u be generable from x by repeated application of the transformations N_j have the same value F(u). Then a code sequence somewhat different from (8) can be used to calculate F(x).

As a matter of fact, as observed by Darlington and Burstall, this alternative code sequence can be used even if M only satisfies the condition $M(x,M(y,z)) = M(y,M(x,z))$, which is somewhat weaker than associativity and commutativity together. In this case, y = F(x) converts (if x is dead) to

(9)
```
y = G(F(∋ V(x)));
j = 0;
while P1(x) ∨ ... ∨ Pk(x);
   z = if P1(x) then H1(x) else if ...
                  else if Pk(x) then Hk(x);
   if j ≠ 0 then
      z = G(z);
   end if;
   j = if P1(x) then 1 else if ...
                  else if Pk(x) then k;
   y = M(z,y);  x = Nj(x);
end while;
```

(v) Assume that

(10) $F(x) = \text{if } P_1(x) \text{ then } M(F(N_1(x)), F(N_1'(x))) \text{ else if } \ldots$
$\text{else if } P_k(x) \text{ then } M(F(N_k(x)),F(N_k'(x)))$,

and that the two-parameter mapping M is associative. Suppose that $N_1,N_1',\ldots,N_k,N_k'$ all have inverses, and that the ranges of these mappings are all disjoint, so that given an x of the form $N_j(z)$ or $N_j'(z)$ the $z = N^{-1}(x)$ from which it was obtained is uniquely defined. (I.e., we can define N^{-1} as the 'combined' inverse of all the mappings $N_1,N_1',\ldots,N_k,N_k'$.) Suppose moreover there exists a boolean function R(x) which is true for all elements of the form $N_j'(z)$ and false for all elements of the form $N_j(z)$. Then y = F(x) can be converted into

```
(11)  x' = x;
      while P_1(x') ∨ ... ∨ P_k(x');
           x' = if P_1(x') then N_1(x') else if ...
                           else if P_k(x') then N_k(x');
      end while;
      y = F(x');  ⊢ ¬(P_1(x') ∨ ... ∨ P_k(x'))
      while x' ≠ x;
           if ¬R(x') then
              x' = N^-1(x');
              x' = if P_1(x') then N_1'(x') else if ...
                              else if P_k(x') then N_k'(x');
              while P_1(x') ∨ ... ∨ P_k(x');
                 x' = if P_1(x') then N_1(x') else if ...
                                 else if P_k(x') then N_k(x');
              end while;
              y = M(y, F(x'));  ⊢ ¬(P_1(x)   ...   P_k(x))
           else
              x' = N^-1(x')
           end if;
      end while;
```

If there exists a left-identity element for the two parameter mapping M the code sequence (11) can be simplified. We leave it to the reader to work out the details of this simplification.

A few additional, but less generally useful cases of recursions which can be converted into iterations are found in the cited works of Walker, Strong, Burstall, and Darlington.

We have already noted that if the set-theoretic function F appearing in the general recursive relationship (1) will be used only in the context $\ni F(x)$, then (1) leads to a recursive algorithm that can be used to calculate F, even if only inclusion and not equality has been proved in (1). This remark carries through to the various iterative constructions into which (1)

can be expanded; all these expansions remain correct if we replace certain of the set-theoretic functions appearing in them by functions having smaller set values ('smaller' in the sense of set-theoretic inclusion), provided that other of the set-theoretic functions map smaller sets into smaller sets. In cases of this sort, simple function composition $F_1(F_2(x))$ will often be replaced by the element-by-element composition function $F_1[F_2[x]] = \{z|(\ u_1,u_2)\ u_1 \in x\ \&\ \langle u_1,u_2\rangle \in F_1\ \&\ \langle u_2,z\rangle \in F_2\}$. Moreover, in dealing with cases of this sort, associativity of a two parameter map M can be replaced by the inclusion $M[x,M[y,z]] \supseteq M[M[x,y],z]$, while commutativity of M and G can be replaced by $G[M[x,y]] \supseteq M[G[x], G[y]]$, etc.

Whenever it is possible to derive a praa R using one of the recursion-removal schemata described in the preceding pages, the derivation is likely to be more advantageous than any other, since in such a case all that needs to be proved is the mathematical fact embodied in the recursive relationship (2). Once this is done, rather complex program structures such as (3), (6), (8), (9), etc., together with all the inductive assumptions needed to prove their correctness, follow immediately. Moreover, we avoid the labor of supplying all the intermediate proof details, instantiations, etc. that would be needed to prove such a program structure correct even after it had been supplied with inductive assumptions.

To convince the reader that the recursion-removal rules displayed above do generate a number of interesting root praas, we will give a few examples.

(a) *Greatest common divisor.* Let x and y be nonnegative integers not both zero, gcd(x,y) their greatest common divisor. Then

gcd(x,y) = if x=0 then y else if x>y then gcd(y,x-y) else gcd(y-x,x)

converts into a familiar iteration, and

```
gcd(x,y) = if odd(x) & odd(y) then
              if x>y then gcd(x-y,y) else gcd(x,y-x)
           else if x=0 then y else if y = 0 then x
           else if even(x+y) then 2*gcd(y/2,x/2)
           else if even(x) then gcd(x/2,y)
           else if even(y) then gcd(x,y/2)
```

leads to a more efficient iteration.

(b) *Merging of sorted arrays.* Let x and y be two sorted arrays, and let merge(x,y) be a sorted array with

rangecount(merge(x,y)) = rangecount(x) + rangeccunt(y) .

Then the relationship

```
merge(x,y) = if x= nℓ then y else if y = nℓ then x
             else if x(1)≤y(1) then x(1) ‖merge(x(2:),y)
             else y(1)‖ merge(x,y(2:))
```

leads, since the concatenation operator ‖ is associative, to a variant of the familiar merging loop. Note that x(2:) is the vector of all components of x, omitting the first.

(c) *Binary search.* Let a be a sorted array of reals, and let $\ell \leq u$ be integers. Define place(a,ℓ,u,x) as the least index i in the range $\ell \leq i \leq u$ such that a(i) = x, if there is any such i; otherwise zero. Then

```
place(a,ℓ,u,x) = if ℓ=u then if a(ℓ)=x then ℓ else 0
                 else if a(⌊ℓ+u/2⌋) ≥ x then place(a,ℓ,⌊ℓ+u/2⌋,x
                 else place(a,⌊ℓ+u/2⌋+1,u,x)
```

converts to the standard binary search.

On the other hand, certain algorithms which work with stacks and which consequently are often expressed recursively are best regarded as having iterative root forms. For example, consider the well known algorithm of Tarjan which takes

an undirected graph G and a node x in it from which every other node can be reached, and which develops a 'depth first spanning tree' T in G. Perhaps the most convenient root form for this algorithm is

```
(12)  nodestack = <x>;  nodes = {x};  T = nl;
      while nodestack ≠ nl;
            |= range(T) ∪ dom(T) ∪ range(nodestack) ⊆ nodes
             & nodestack(1)=x
             & (∀j,k, 1≤j<k<#nodestack)(∃ p∈paths(T,nodestack(j),
                                nodestack(k))
             & (∀z ∈ nodes | (G{z}-nodes) ≠ nl) z∈range(nodestack)
             & (∀z ∈ nodes) ∃!p ∈ paths(T,x,z)
                 /* ∃!p means that there exists a unique p */
            if  x ∈G{nodestack(#nodestack)} | x ∉ nodes then
               nodestack = nodestack || <x>; nodes = nodes ∪ {x};
               T = T with {<nodestack(#nodestack),x>};
            else
               nodestack = nodestack(1:#nodestack-1);
            end if;
      end while;
```

Since successive elements of *nodestack* are T-descendants of each other, and since all the elements of *nodes* which have descendants not in *nodes* belong to the range of *nodestack*, it is not hard to show, by supplementing the loop assumption, that whenever y,z ∈ *nodes* and <y,z> ∈ G it follows that z is either a T-descendant or ancestor of y, or that in T z lies to the left of the path leading to y, which is of course a property fundamental to the use of depth-first spanning trees.

Note that, in spite of its typically recursive use of a stack, (12) cannot be regarded as a true recursive algorithm, since it accesses the global variables *nodestack*, *nodes*, and *T* in a manner forbidden to straightforwardly recursive functions.

4. Additional Comments on praa Manipulation.

a. The block substitution rule revisited; Pramas

The fundamental block substitution Rule (1.e) of Section 2(c) allows us to combine praas by replacing either single where statements or groups of several such statements in a correct praa R by an auxiliary praa R_1 which contains appropriate output assertions. While rather general, this replacement principle is not quite as general as we would like it to be, since as normally written a praa may contain fewer where statements than would be ideal for broad application of Rule (1.e). In the next few pages we will (again following Gerhart) address this point, specifically by modifying the proof formalism described in Sections 2(a,b) above in such a way as to facilitate insertion and deletion of where statements. For this, the following definition is appropriate.

Definition. (a) A *prama* ('program with assumptions and maximal assertions') R is a program Q (of the language SL), together with three sets E, E', M. As in a praa, E and E' are are both sets of propositions-at-places and -at-functions of Q; the set E (resp. E') is called the set of assumptions of R (resp. the set of assertions) of R. The elements of M are pairs (Λ, P^{α}), where P^{α} is a proposition-at-a-place or -at-a-function of R, and where Λ is a list of variables of R. We impose the restriction that if α is a place in a function f, then only variables which do not appear in any function other than f and which are not arguments of f can appear in the list Λ. (If α is a place in the main code block of R, then only variables not appearing in any function can appear in Λ). The propositions P^{α} appearing in pairs (Λ, P^{α}) belonging to M are called *maximal assertions* of R, and Λ is called the *modifiable variable list* associated with the maximal assertion P^{α}.

We impose the restriction that no prama can contain more than one maximal assertion at any place or function, and that if it contains a maximal assertion P at a given place or function,

then all other assertions Q at the same place or function which involve any of the variables of P are implied by P.

In heuristic terms, each maximal assertion P states both that the assertion P^{α} is true whenever control reaches the place α (or whenever the function α is exectued), *and that no predicate stronger than P is known at* α. More specifically:

(b) A prama R is *correct* or *valid* if conversion of all its maximal assertions to ordinary assertions turns R into a correct praa, and if the following additional conditions are also satisfied:

(b.1) Let $(\Lambda, P^{\alpha}) \in M$ and let α be a place in R. Then R must remain correct if we insert the <u>where</u> statement

(1) $\langle x_1,\ldots,x_n\rangle = \langle u_1,\ldots,u_n\rangle$ <u>where</u> $P(u_1,\ldots,u_n,\text{other\_vars})$

immediately following the place α. Here $x_1,\ldots,x_n$ is the list Λ of variables, and the predicate P is assumed to have the form $P(x_1,\ldots,x_n,\text{other\_vars})$.

(b.2) Let $(\Lambda, P^{\alpha}) \in M$, and let α be a function f in R. Then the list Λ must consist of the single variable x and R must remain correct if we replace any function call

(2) $$x = f(\text{expn}_1,\ldots,\text{expn}_m)$$

in R by the <u>where</u> statement

(3) $$x = u \text{ } \underline{\text{where}} \text{ } P(u,\text{expn}_1,\ldots,\text{expn}_m) \ .$$

(Note that the form of the proposition-at-a-function f is necessarily $P(u,y_1,\ldots,y_m)$, where m is the number of parameters of f.)

If $\Lambda = (x_1,\ldots,x_n)$, and E is the set of assumptions of the prama R, we will sometimes indicate the presence of the maximal assertion (Λ, P^{α}) by writing $E(x_1,\ldots,x_n) \vdash\!\!\vdash P^{\alpha}$. Similarly, in writing out the text of a prama, we shall distinguish maximal assertions by prefixing them with the sign $(x_1,\ldots,x_n)\vdash\!\!\vdash$, or, in case $x_1,\ldots,x_n$ is the collection of all variables which can be changed at a given place or function, simply by prefixing the sign $\vdash\!\!\vdash$.

Now we begin to state a set of proof and transformation rules for pramas, culminating in a discussion of block substitution in the prama case.

Proof Rules:

(a) The proof rules (a), (b), (c), (d), (e), (f) stated in Section 2(b) above and also the two Induction Principles (for propositions-at-a-place and for propositions-at-a-function) stated there carry over from praas to pramas. In applying these rules, we can use any maximal assertion $(\Lambda) \vdash\!\!\!\vdash P^{\pi}$ or $(\Lambda) \vdash\!\!\!\vdash P^{f}$ as a simple assertion (e.g. we can use $(\Lambda) \vdash\!\!\!\vdash P^{\pi}$ as $\vdash P^{\pi}$, etc.)

(b) Proof rules (g), (h), (i), (j), (k), (ℓ), (m), (n) and (o) stated in Section 2(b) above carry over from praas to pramas, provided that in applying any of these rules to deduce proposition P^{π} at the place π in the prama R we insist that R contain no maximal assertion (Λ,Q) at π whose list Λ of variables involves any of the variables of P. In applying these rules we can use any maximal assertion as a simple assertion (in the manner explained in the preceding paragraph).

We assume throughout the next few pages that R is a correct prama, that β, β_-, β_+, π, π_-, π_+, etc. are places in R, that f, g etc. are functions in R, that the variables appearing in R are $x_1,\ldots,x_m$, and that Λ is the list $x_1,\ldots,x_n$ of variables. We will sometimes call the variables of the list Λ *active* variables of a maximal assertion $(\Lambda) \vdash\!\!\!\vdash P^{\pi}$ or $(\Lambda) \vdash\!\!\!\vdash P^{f}$, and call all other variables of P *passive* variables of such an assertion.

(c) A prama remains correct if the list Λ of variables attached to any of its maximal assertions-at-a-place or -at-a-function is replaced by a smaller list of variables. In particular, a prama remains correct if any of its maximal assertions is replaced by a corresponding simple assertion.

(d) (Maximal Assertion Strengthening) Let R contain the maximal assertion $(\Lambda) \vdash\!\!\!\vdash P^{\pi}$ at the place π, and suppose that if the maximal assertion were reduced to an ordinary assertion the additional assertion $\vdash P_1^{\pi}$ at π could be deduced, where we assume that the implication $(\forall x_1,\ldots,x_n)(P_1 \Rightarrow P)$ holds identically. Then $(\Lambda) \vdash\!\!\!\vdash P^{\pi}$ can be replaced by $(\Lambda) \vdash\!\!\!\vdash P_1^{\pi}$.

(e) (Union Rule for Maximal Assertions) Suppose that maximal assertions separated only by a label appear in R, as follows:

$$(\Lambda) \vdash\!\!\!\vdash P^{\beta_-}, \quad L\colon (\Lambda_1) \vdash\!\!\!\vdash P_1^{\beta_+}$$

Suppose that every assertion of R at the place β_- immediately preceding the label L which is nct independent of the variables of Λ_1 is implied by the predicate P_1. Let the variables appearing in the list Λ (resp. Λ_1) be $x_1,\ldots,x_n$ (resp. $x_k\ldots,x_\ell$), and let the list of all variables of R be $x_1,\ldots,x_m$. Then if the predicate P is independent of the variables $x_{n+1},\ldots,x_\ell$, the maximal assertion $(\Lambda) \vdash\!\!\!\vdash P^{\beta_-}$ can be replaced by $(x_1,\ldots,x_\ell) \vdash\!\!\!\vdash (P \,\&\, P_1)^{\beta_-}$.

Proof: It is clear that the assertion $\vdash (P \,\&\, P_1)^{\beta_-}$ holds at β_-. Since $P_1(x_1,\ldots,x_m)$ implies every assertion of R which is at β_- and which is not independent of $x_k\ldots,x_\ell$, and since each assertion at β_- which is not independent of $x_1,\ldots,x_n$ is implied by $P(x_1,\ldots,x_m)$, it is clear that every assertion of R which is at β_- and which is not independent of $x_1,\ldots,x_\ell$ is implied by $P \,\&\, P_1$. Hence insertion of the where statement

$$(4)\quad \langle x_1,\ldots,x_\ell\rangle = \langle u_1,\ldots,u_\ell\rangle \text{ where } P(u_1,\ldots,u_\ell,x_{\ell+1},\ldots,x_m) \,\&\, P_1(u_1,\ldots,u_\ell,x_{\ell+1},\ldots x_m$$

at β_- does not invalidate any of the R-assertions at β_-. Suppose that the statement is inserted at β_-, and also that the where statement

(5) $\langle x_k,\ldots,x_\ell\rangle = \langle u_k,\ldots,u_\ell\rangle$ where

$$P(x_1,\ldots,x_{k-1},u_k,\ldots,u_\ell,x_{\ell+1},\ldots,x_n)$$

is inserted immediately before the place β_+ in R. Let R be the prama that results from these insertions, and let c' be a computation for the code text R'. Then since $P(u_1,\ldots,u_\ell,x_{\ell+1},\ldots,x_m)$ is independent of the variables $u_{n+1},\ldots,u_\ell$, it is clear that any values assigned to the variables $x_1,\ldots,x_n$ by (4) could also have been assigned by the where statement

(6) $\langle x_1,\ldots,x_n\rangle = \langle u_1,\ldots,u_n\rangle$ where $P(u_1,\ldots,u_n,x_{n+1},\ldots,x_m)$,

and hence that any values assigned to $x_1,\ldots,x_\ell$ by (4) could also have been assigned by the two successive statements (5), (6). Thus, in the presence of (5), the only components of the computation c' which can differ from the corresponding components of a computation c for R must be components which are either at β_- or at β_+. Thus c' must satisfy all the assertions of R at places other than β_- and β_+. But since any values assigned to $x_1,\ldots,x_\ell$ by (4) could also have been assigned by (5), (6) in combination, c must satisfy all the assertions of R at β_+, and since we have already seen the same to be true at β_-, our proof rule follows. Q.E.D.

(f) (Splitting Rule for Maximal Assertions) Suppose that β_-, L:, β_+, that a maximal assertion of the form

(7) $$(x_1,\ldots,x_\ell) \vdash\!\vdash (P \,\&\, P_1)^{\beta_-}$$

appears at the place β_-, and that no maximal assertion appears at β_+. Suppose that $n \leq \ell$, that the predicate P is independent of the variables $x_{n+1},\ldots,x_\ell$, and that the assertion $\vdash P_1^{\beta_+}$ can be deduced at β_+. Then the maximal assertion $(x_1,\ldots,x_\ell) \vdash\!\vdash P_1^{\beta_+}$ can be added to R, with preservation of correctness.

Proof: As seen in the proof of (e) above, any values assigned to $x_1,\ldots,x_\ell$ by the where statement (6) can be assigned to these variables by (7). Thus insertion of (6) into the text of R at β_+ will not cause any assertion of R to become false.

(g) (Reduction Rule) Suppose that $(\Lambda) \vdash\!\vdash (P \,\&\, P_1)^\beta$ is a maximal assertion of R and that P_1 is independent of the variables appearing in the list Λ. Then $(\Lambda) \vdash\!\vdash (P \,\&\, P_1)^\beta$ can be replaced by $(\Lambda) \vdash\!\vdash P^\beta$, with preservation of correctness.

Proof: Consider two pramas R_1, R_2 derived from R, the first obtained by inserting the statement (6) with $\ell = n$ immediately after the place β, the second by inserting (4) instead. We shall simply show that both admit the same sets of computations. Obviously the set of computations which R_1 admits is no smaller than the set of computations which R_1 admits. Now let c be a computation of minimal length admitted by R_2 but not by R_1. Clearly the semi-final component of c must be at the place β. If we truncate the final component of c, we obtain a c' which is a valid computation both for R_1 and R_2. Hence it must satisfy all the assertions of R_1, and in particular the final component of c' must satisfy P_1. But then since P_1 is independent of all the variables of Λ any final values which can be assigned to the variables $x_1,\ldots,x_n$ by (6) can also be assigned by (4), so that c must be a computation which R_1 admits, a contradiction which proves our rule. Q.E.D.

The next rules to be stated justify motion of maximal assertions backward through the code text of a prama. In addition to the standing assumptions described above, we assume throughout the next few pages that *no maximal assumption of R is initially at the place* β_-.

(h) (Assignment to an active variable) If β_-, $x_1 = expn(x_1,\ldots,x_m)$, β_+, if $(\Lambda) \vdash\!\!\!\vdash (P(x_1,\ldots,x_m))^{\beta_+}$, and if $P(expn(x_1,\ldots,x_m),x_2,\ldots,x_m)$ implies every assertion of R which is at the place β_- and which is not independent of the variables of Λ, then the maximal assertion $(\Lambda) \vdash\!\!\!\vdash (P(expn(x_1,\ldots,x_m),x_2,\ldots,x_m))^{\beta_-}$ can be added to R with preservation of correctness.

We shall give a detailed proof of this rule since it is the first of its type that we state; detailed proofs will not be given for subsequent rules of this general kind, since rather similar arguments can be used in all cases.

Proof: Since $\vdash P^{\beta_+}$, it is clear that $\vdash (P(expn(x_1,\ldots,x_m), x_2,\ldots,x_m))^{\beta_-}$. Moreover, since $P(expn(x_1,\ldots,x_m),x_2,\ldots,x_m)$ implies every assertion of R which is at β_- and which is not independent of $x_1,\ldots,x_n$, it is clear that insertion of the where statement

(8) $\langle x_1,\ldots,x_n\rangle = \langle u_1,\ldots,u_n\rangle$ where $P(expn(u_1,\ldots,u_n,other\_vars),u_2,\ldots,u_n,other\_vars)$

immediately before the place β_- does not invalidate any of the R-assertions at β_-. It is also clear that insertion of (8) at β_- does not invalidate the assertion $\vdash P^{\beta_+}$. Suppose that the where statement

(9) $\langle x_1,\ldots,x_n\rangle = \langle u_1,\ldots,u_n\rangle$ where $P(u_1,\ldots,u_n,other\_vars)$

is inserted immediately before the place β_+ in R. It is clear that in the presence of (9) at β_+ the members of the set of computations for the code text R' containing (8) differ from

the the computations legal in the absence of (8) only for computation components which are at either β_- or β_+. Thus these computations satisfy the assertions of R' at all places other than β_- or β_+ , and, since we have seen that all assertions at β_- and β_+ are satisfied also, it follows that R' is correct, which proves the transformation rule that has been stated.

(i) (Assignment to a passive variable) If $\beta_-,\ x_{n+1} = \mathrm{expn}(x_1,\ldots,x_m),\ \beta_+$, if $(\Lambda) \vdash\!\!\!\vdash (P(x_1,\ldots,x_m))^{\beta_+}$, if $P(x_1,\ldots,x_n,\ \mathrm{expn}(x_1,\ldots,x_m),\ x_{n+2},\ldots,x_m)$ implies every assertion of R which is at the place β_- and which is not independent of the variables of Λ, and if $\mathrm{expn}(x_1,\ldots,x_m)$ is independent of the variables of Λ, then the maximal assertion

$$(10)\quad (\Lambda) \vdash\!\!\!\vdash (P(x_1,\ldots,x_n,\ \mathrm{expn}(x_1,\ldots,x_m),\ x_{n+2},\ldots,x_m))^{\beta_-}$$

can be added to R, with preservation of correctness.

(The proof is like that of (h); we must assume that $\mathrm{expn}(x_1,\ldots,x_m)$ is independent of the variables of Λ since otherwise insertion of the <u>where</u> statement (8) at β_- might influence the course of computation past the place β_+ . Note that if we are prepared to drop all variables on which $\mathrm{expn}(x_1,\ldots,x_n)$ depends from Λ, the condition ' $\mathrm{expn}(x_1,\ldots x_m)$ is independent of the variables of Λ ' can always be made to hold.)

(j) (Selection assignment to an active variable) If $\beta_-,\ x_1 = \ni\mathrm{expn}(x_1,\ldots,x_m),\beta_+$, if $(\Lambda) \vdash\!\!\!\vdash (P(x_1,\ldots,x_m))^{\beta_+}$, and if $(\forall u \in \mathrm{expn}(x_1,\ldots,x_m))\ P(u,x_2,\ldots,x_m)$ implies every assertion of R which is at the place β_- and which is not independent of the variables of Λ, then the maximal assertion $(\Lambda) \vdash\!\!\!\vdash ((\forall u \in \mathrm{expn}(x_1,\ldots,x_m))\ P(u,x_2,\ldots,x_m))^{\beta_-}$ can be added to R, with preservation of correctness. (Proof like that of (h).)

(k) (Selection assignment to a passive variable) If $\beta_-,\ x_{n+1} = \ni\,\mathrm{expn}(x_1,\ldots,x_m),\beta_+$, if $(\Lambda) \vdash\!\!\!\vdash (P(x_1,\ldots,x_m)^{\beta_+}$, if $(\forall u \in \mathrm{expn}(x_1,\ldots,x_m))\ P(x_1,\ldots,x_n,\ u,x_{n+2},\ldots,x_m)$ implies every assertion of R which is at the place β_- and which is not independent of the variables of Λ, and

if $expn(x_1,\ldots,x_m)$ is independent of the variables of Λ, then the maximal assertion

(11) $(\Lambda) \vdash\!\vdash ((\forall u \in expn(x_1,\ldots,x_m))\ P(x_1,\ldots,x_n,u,x_{n+2},\ldots x_m))^{\beta_-}$

can be added to R, with preservation of correctness.

(ℓ) (Conditional Transfer Rule) Let β_-, <u>if</u> C <u>then</u> <u>go</u> <u>to</u> L, β_+, and suppose that the place π follows the label L. Suppose that $(\Lambda) \vdash\!\vdash P^{\beta+}$, and that $(\Lambda) \vdash\!\vdash P_1^{\pi}$. Suppose that the proposition $(C \Rightarrow P_1)\ \&\ (\neg C \Rightarrow P)$ implies every assertion of R which is at the place β_- and which is not independent of the variables of Λ. Then the maximal assertion $(\Lambda) \vdash\!\vdash ((C \Rightarrow P_1)\ \&\ (\neg C \Rightarrow P))^{\beta_-}$ can be added to R, with preservation of correctness.

(The proof can be adapted from that of (h); details are left to the reader.)

Note that if the condition C in <u>if</u> C <u>then</u> <u>go</u> <u>to</u> L is identically *true*, then we can always take the predicate P to be *false*, in which case the maximal assertion $(\Lambda) \vdash\!\vdash ((C \Rightarrow P_1)\ \&\ (\neg C \Rightarrow P))^{\beta_-}$ reduces immediately to $(\Lambda) \vdash\!\vdash P_1^{\beta-}$.

(m) (Label Rule) Suppose that β_-, L, β_+, that $(\Lambda) \vdash\!\vdash P^{\beta+}$, and that the proposition P implies every assertion of R which is at the place β_- and which is not independent of the variables of Λ. Then $(\Lambda) \vdash\!\vdash P^{\beta_-}$ can be added to R, with preservation of correctness. (This is rather obvious.)

(n) (Function call rule) Let f be a function of R which allows entry in the sense of Rule (4.e) of Section (2.c); let β_-, $x_1 = f(expn_1,\ldots,expn_k)$, β_+, and let $(\Lambda) \vdash\!\vdash P^{\beta+}$. Suppose that the maximal assertion $(z) \vdash\!\vdash (C\ (z,y_1,\ldots,y_k))^f$ is available at f, and that the proposition

(12) $(\forall z)(C(z, expn_1,\ldots,expn_k) \Rightarrow P(vars))$

implies every assertion of R which is at the place β_- and which is not independent of the variables of Λ. Then

(13) $(\Lambda) \vdash\!\!\!\vdash ((\forall z)(C(z, expn_1,\ldots,expn_k) \Rightarrow P(vars)))^{\beta_-}$

can be added to R, with preservation of correctness.

Proof: Since R remains correct if the function call is replaced by

(14) $x_1 = u$ where $C(u, expn_1,\ldots,expn_k)$,

rule (j) above implies that the assertion $\vdash ((\forall u)(C(u, expn_1,\ldots,expn_m) \Rightarrow P(vars)))^{\beta_-}$ must hold at β_-. Next suppose that we assume that this replacement has been made and that we further modify R by inserting the statement

(15) $$<x_1,\ldots,x_n> = <u_1,\ldots,u_n> \text{ where}$$
$$(\forall z)(C(z,expn_1(u_1,\ldots u_n,x_{n+1},\ldots x_n),\ldots expn_k(u_1,\ldots u_n,x_{n+1}\ldots x_n)))$$

at β_-. Using rule (j) again, we see that these modifications preserve the correctness of R. Since the function f allows entry in the sense of Rule (4.e) of Section (2.c), all assertions encountered during any computation c which begins at the entry place of f and continues until return from f will be satisfied no matter what the initial state of c is. Thus, even if the where statement (14) is replaced by the function call $x_1 = f(expn_1,\ldots,expn_k)$ after the insertion of (15), R remains correct. Q.E.D.

(o) (Function call rule for a passive result variable) Let f be a function of R, let β_-, $x_{n+1} = f(expn_1,\ldots expn_k)$, β_+, and let $(\Lambda) \vdash\!\!\!\vdash P^{\beta+}$. Suppose that the maximal assertion $(z) \vdash\!\!\!\vdash (P_1(z,y_1,\ldots,y_k))^f$ is available at f, and that the proposition (12) implies every assertion of R which is at the place β_- and which is not independent of the variables of Λ. Suppose that all the argument expressions $expn_1,\ldots,expn_k$ are independent of the variables of Λ. Suppose also that neither f nor any function g that can be called directly or indirectly from f contains an assertion that references

any variable in Λ. Then the maximal assertion (13) can be added to R, with preservation of correctness.

The proof is readily adapted from that of (n); details are left to the reader.

Observe that the condition that all $expn_j$ are independent of the variables of Λ is required for the reason noted following (i) above. Moreover, this same condition, supplemented by the condition that neither f nor any function g that can be called directly or indirectly from f contains an assertion that references any variable in Λ, can be used in (n) to replace the condition that f allows entry.

(p) (Proposition-at-a-function Rule) Suppose that the calls to a function f of R are $\beta_-^{(j)}$, $x_{i_j} = f(expn_1^{(j)},\ldots expn_k^{(j)}), \beta_+^{(j)}$, $j = 1,\ldots,p$. Let the parameters of f be $y_1,\ldots,y_k$, and let P be a predicate having exactly k+1 free variables. Suppose that for each j, $1 \leq j \leq p$, an assertion $\vdash Q_j^{\beta_-^{(j)}}$ is available at $\beta_-^{(j)}$, and also that a maximal assertion having the form

$$(16)\quad (x_{i_j}) \vdash\!\vdash ((\forall u)(Q_j(x_1,\ldots,x_{i_j-1},u,x_{i_j+1},\ldots,x_k) \;\&\; P(x_{i_j},expn_1^{(j)}(x_1,\ldots x_{i_j-1},u,x_{i_j+1},\ldots x_k),\ldots \ldots expn_k^{(j)}(x_1,\ldots,x_{i_j-1},u,x_{i_j+1},\ldots x_k))))^{\beta_+^{(j)}}$$

is available at $\beta_+^{(j)}$. Suppose also that the assertion $\vdash P^f$ is available at f, and that this assertion implies all other assertions available at f. Then the maximal assertion

$$(17)\qquad (z) \vdash\!\vdash (P(z,y_1,\ldots,y_k))^f$$

at f can be added to R, with preservation of correctness.

Proof: Arguing as in (h) above, we can show that R remains correct if any call $x = f(expn_1,\ldots,expn_k)$ is replaced by $x = v$ where $P(v,expn_1,\ldots,expn_k)$. Since $\vdash P^f$ is available at f, the present rule follows immediately from the definition

of maximality for a proposition-at-a-function.

(q) (Return statement rule) Let f be a function of R, let its parameters be $y_1,\ldots,y_k$, and let the remaining variables of f be $y_{k+1},\ldots,y_m$. Let $(z) \vdash\!\!\vdash (P(z,y_1,\ldots,y_k))^f$ be a maximal assertion at f. Let the place π in f immediately precede the statement <u>return</u> *expn*. If the proposition $P(expn,y_1,\ldots,y_k)$ implies every assertion at π which is not independent of the variables of f, then the maximal assertion $(y_{k+1},\ldots,y_m) \vdash\!\!\vdash (P(expn(y_1,\ldots,y_m), y_1,\ldots,y_k))^{\pi}$ can be added to R, with preservation of correctness.

We will normally use the above-stated prama proof and transformation rules in the following manner. Any proof of praa correctness will ordinarily begin with the statement of a group of temporary assumptions; these assumptions will generally be attached to function entries and to the heads of loops, in a manner which (necessarily) breaks every closed path through the praa R being proved. Often no other statements than the temporary assumption itself ever needs to be generated at the place π at which such an assumption is made. If this is the case, then a maximal assertion built from the assumption and involving some or all of the variables appearing in it can be put at π without spoiling the validity proof of R (since the presence of a maximal assertion at π restricts deduction of other assertions at π, but does not restrict deduction at any other place). If this can be done, then the proof of R's correctness will *ipso facto* prove the correctness of a prama R', largely identical with R, but containing maximal assertions not present in R. To know that R' is correct is of course to know more than the correctness of R; in particular, the transformation rules stated in this section allow maximal assertions to be moved about in R'. Once a maximal assertion $(x_1,\ldots,x_n) \vdash\!\!\vdash P^{\pi}$ has been moved to the place π, the following technique can be used to justify substantial modifications of the code immediately preceding π, if such modification is desired:

(i) Introduce the where statement

$$\langle x_1,\ldots,x_n\rangle = \langle u_1,\ldots,u_n\rangle \text{ where } P(u_1,\ldots,u_n,x_{n+1},\ldots x_m)$$

immediately before the place π, retaining the maximal assertion at π; this introduces a new place π'.

(ii) The presence of the where statement at π' makes all the variables $x_1,\ldots,x_n$ dead at π'. Therefore the dead variable rule (3.b) of Section 2 allows deletion and insertion of other statements or groups of statements preceding π', provided that these other statements modify only the variables $x_1,\ldots,x_n$ and perhaps also other variables used only to calculate values to be assigned to $x_1,\ldots,x_n$. Make deletions and insertions, as appropriate.

(iii) Now use the block substitution rule (1.e) of Section(2.c) to replace the inserted where statement by any appropriate prama.

b. Proofs of Termination

A praa or prama R is said to be *terminate* for a given initial condition of its input variables if the set of all computations which begin with these initial input values and then proceed from the start of R to any place in R is finite. (To reduce the standard notion of program termination to this notion, take a program Q, add the tacit input assumptions of Q to Q explicitly, and also add the assumption $\not\equiv (\text{false})^{\pi}$ to the exit place π of Q, thus obtaining a praa R. Then Q terminates in the conventional sense if and only if R terminates in the sense just explained. Note that execution of a praa is taken to terminate immediately if control reaches a point at which an assumption is violated.) The question of

termination has been deliberately systematically neglected in the preceding pages. We shall now discuss this question briefly, approaching it in a spirit suggested by the transformational formalisms that have been introduced.

Which of the transformations of Section 2.c preserve termination? This obvious but significant question is easily answered. Praa termination is preserved when the following transformation rules are applied: (1a-d), (1f-g), (2a-o), (3a-e), (4a-d). Concerning the more complex substituion rules (1.e), (4.e), and (4.g) we make the following comments:

Ad (1.e): Let R, R_1, L_j, $j = 1,\ldots,k$ be as in transformation rule (1.e), and let $\beta_+^{(j)}$ be the place in R_1 immediately following the label L_j. Let the praa R_1' result if we insert the assumption $\models$ $(false)^{\beta_+^{(j)}}$ at each of the places $\beta_+^{(j)}$, $j = 1,\ldots,k$. Suppose that R_1' terminates (so that any sufficiently long computation in R_1 must reach one of the places $\beta_+^{(j)}$). Then the praa which results if we fuse R_1 into R in the manner described by Rule (1.e) terminates.

This assertion can be proved by an easy adaption of the argument used to prove Rule 1.e; details are left to the reader.

Ad (4.e): Let R, E, f and R' be as in Rule (4.e), Section (2.c). We shall say that f *allows terminating entry* if f allows entry in the sense of Rule (4.e), and if

in addition the praa R' terminates. Let π' be any place within R, and let x' be any variable not otherwise occurring in R. Assume that f allows terminating entry. Then if we insert the function call

(18) $$x' = f(expn_1,\ldots,expn_m)$$

at π' , termination is preserved.

Ad (4.g). Rule (4.g) is derived from (4.e). If in applying Rule (4.e) we assume that the function f appearing in the inserted call (18) allows terminating entry, then termination is preserved.

Ab initio proofs of termination generally combine references to a program's statement order and block structure with a few supplementary mathematical facts. The following straightforward definitions and 'termination proof rules' formalize the techniques most commonly used.

Definition. Let R be a praa. A *subsection* s of R is a set of places in the main block of R, together with the statements (or labels, which we treat here as 'no-ops') immediately following these places. A *decomposition* of a subsection s of R is a collection of disjoint subsections s_i of R whose union is s. One place π in R is a *predecessor place* of another place π' if the statement at π' can be the next statement executed after execution of the statement at π. The *entry places* of a subsection s are all places in s having predecessor places not in s; the *exit places* of a subsection s are all places in s which are predecessors of places not in s. A subsection s of R is said to *terminate* if for each of its entry places π, the praa R_π obtained by modifying R in the following manner terminates:

(i) Drop all statements in the main block of R which do not belong to s; add a label L_0 at the place π if necessary and also add one statement, which will be the entry statement of R_π , having the form go to L_0. Also,

(ii) Add a stop statement with label L_1 to the end of R_π , change every statement if C then go to L in s which references

a label L outside s to *if* C *then go to* L_1 , and insert the statement *go to* L_1 immediately after every statement in s whose immediate successor statement is not in s.

We now give some termination proof rules:

(a) (Simple forward branch rule) Any praa subsection containing only forward branches and no selections $\ni$ s from infinite sets terminates.

(b) (Generalized forward branch rule) Let s be a subsection of the praa R, and let $s_1,\ldots,s_n$ be a decomposition of s. Suppose that s_j terminates for each j = 1,...,n, and that whenever π and π' are both places of s and π is a predecessor place of π', then if π belongs to s_j and π' belongs to s_k with $j \neq k$ we have $j < k$. Then s terminates.

(c) (Backward branch rule) Let s be a terminating subsection of the praa R, let $\pi_1,\ldots\pi_k$ be all the entry places of s, and let $\pi'_1,\ldots,\pi'_n$ be places which have predecessors in s but which are not themselves in s. Suppose that for j = 1,...,n, the statement following the place π'_j has the form *if* C_j *then go to* L_j , where the label L_j is at some place of s that is physically prior to the place π'_j (so that the *if*-statement represents a possible backward branch). Suppose that there exists a partially ordered set Δ in which indefinitely long ascending sequences of elements are impossible, and an expression expn(vars) in the variables $x_1,\ldots x_m$ of R, such that we have

$$[\pi_i]E \cup \left\{ (x_1=x'_1 \& \ldots \& x_m=x'_m)^{\pi_i} \right\} \vdash\ (\mathrm{expn}(x_1,\ldots x_m) > \mathrm{expn}(x'_1,\ldots x'_m))^{\pi'_j}$$

for all entry places π_i of s and all exit places π'_j; where E is the set of assumptions of R, '>' is the order relationship in the partially ordered set Δ, and the x'_j are variables not otherwise appearing. Let s' be the union of s and the statements *if* C_j *then go to* L_j at the places π'_j. Then s' is a terminating subsection of the praa R.

Rule (c) reflects the familiar observation, which goes back as far as [Floyd, 67], that termination proofs for a root praa R will often proceed by mapping the data objects of R into some appropriately devised partially ordered set. Note however that for praa subsections s only involving loops of the 'do' type and their

set-theoretic equivalents, rules (a),(b),(c) normally yield easy termination proofs, since in these cases Δ can simply be either a finite range of integers in its natural order or the lattice of subsets of a finite set, ordered by inclusion. The reader may be interested in applying this remark to prove termination of the counting-sort praa considered earlier.

Programs in our set-theoretic language proceed nondeterministically, since the selection opreator x = ∋expn makes a nondeterministic choice. Thus 'termination' has two possible senses, namely finiteness of *all possible* computations (which is the sense adopted in the last few pages), and the existence of *at least one* computation which reaches a given program point.

While a deterministic program can only terminate by either reaching its exit place or aborting, a program deliberately written to exploit nondeterminism can terminate in two equally legitimate senses, either by reaching some exit point and thereby succeeding, or by failing along all paths and thereby certifying that the combinatorial problem which the program explores is unsolvable. Hence reachability becomes a significant issue in the nondetermistic case, i.e., to justify use of a program written in a manner deliberately exploiting nondeterminism, termination in both senses must be proved. To formalize this, suppose that we say that a place π in a valid praa R is *reachable* if there exists at least one R-computation from the entry place of R to π. Equivalently we can say that π is reachable if R is valid but becomes invalid if the assertion $\vdash (false)^{\pi}$ is added to the assertion set of R.

To prove reachability, one can often proceed in the following manner:

(i) First prove that R must terminate if π is not reached (i.e., that the praa R' obtained by adding $\models (false)^{\pi}$ to R terminates).

(ii) Next consider a computation c of maximal length in R', and suppose the final state of c is at the place π'. We shall call π' a *stopping place* of R; clearly $\pi' \neq \pi$.

(iii) Our aim now is to show that R can have no stopping place π'. To do this, we apply the following observations.

(iv) π' can be a stopping place if some unsatisfiable assumption $\models P^{\pi'}$ is present at π'. To avoid this possibility, we shall suppose throughout the next few paragraphs that no

assumptions are present at any place other than the entry place of R. (In effect, this means that all other assumptions must have been reduced to assertions.)

(v) If π', x = expn, then π' can only be a stopping place if an operation error can arise in the evaluation of *expn* (e.g. *expn* might contain the subexpression x/y, where the value of y might be zero). Direct inspection of *expn* will always reveal the types of operation errors possible in its execution. If assertions ruling out these operation errors are available at π', then π' cannot be a stopping place.

(vi) Similarly, if π', x = ∋ expn, and if assertions both ruling out operation errors in the evaluation of *expn* and ensuring that *expn* has a non-null value are available at π', then π' is not a stopping place. If π', <u>if</u> C <u>then go to</u> L , and if assertions ruling out operation errors in the evaluation of C are available at π', then π' is not a stopping place. If π', $x = f(expn_1,\ldots,expn_k)$, where f is a function of R, and if assertions ruling out operation errors in the evaluation of each of the argument expressions $expn_j$ are available at π', then π' is not a stopping place.

(vii) Suppose that π', <u>return</u> *expn*, and that assertions ruling out operation errors in the evaluation of *expn* are available at π'. Then π' can only be a stopping place if there exists an assumption $\models P^f$ at the function f containing the place π' which comes to be violated at the moment that the <u>return</u> statement is executed. This possibility is ruled out if we assume that R contains no assumptions-at-functions (which is to say that all such assumptions have been converted to assertions).

Programs which deliberately exploit nondeterminism will generally contain <u>fail</u> statements; the semantic effect of such a statement is simply to terminate any computation which reaches it. (In our restricted formalism, such a statement could be represented as a deliberate error, e.g. x = 1/0.) To prove reachability using the termination-based technique

just outlined, it is necessary to prove that such fail statements (or the erroneous statements used to represent them) are avoided (or more precisely, can be avoided). Arguments useful for this purpose can be based on the following notions. Suppose that R_1 is a correct praa. Then R is a *generalization* of R_1 if R is obtained from R_1 by replacing certain of the assignments x = ∋ expn (resp. x = expn) in R_1 by arguments x=∋expn' (resp. x=expn) and by dropping some of the assertions of R_1, provided that an assertion $\vdash$ (expn $\subseteq$ expn') $^{\pi_1}$ (resp. $\vdash$ (expn $\in$ expn') $^{\pi_1}$) is available in R_1 at each point π_1 where such a replacement is made. It is clear that if R is a generalization of R_1 and π is reachable in R_1, then π is reachable in R. (Note however that the correctness of R does not follow from that of R_1, but must be proved separately.)

As an illustration of this we consider the following nondeterministic praa which solves the well-known 'eight queens' problem (placing 8 queens on a chessboard so that no queen attacks any other). We develop the solution as a vector *posns* whose j-th component shows the position of the j-th queen. The praa is:

```
(19) rows = {n|1≤n≤8};  posns = nℓ;  n = 1;
     while n≤8; ⊨ vector(posns) & range(posns) ⊆ rows
            & (1≤∀i,j<n|i≠j)(posns(i)-posns(j)) ∉{0,i-j,j-i})
       posns(n) = ∋(rows-{posns(k)+tilt*(n-k), 1≤k<n,
                                    tilt ∈ {0,+1,-1}});
       n = n+1;
     end while;
     ⊢ vector(posns) & range(posns) ⊆ rows
                  & domain(posns)   = {n, 1≤ n ≤ 8}
          & (1≤ ∀i,j ≤ 8| i≠j)((posns(i)-posns(j)) ∉{0,i-j,j-i})
```

That this praa is correct is easily verified using the temporary assumption shown, and the termination-proof techniques described earlier make it obvious that the praa terminates.

What we wish to show is that, if we assume that the eight-queens problem has a solution, then the end of the praa (19) is reachable. To do this, we make use of the following modified praa, of which (19) is (essentially) a generalization.

```
(20)  soln = u where range(u) ⊆ {n, 1 ≤ n ≤ 8}
                    & domain(u) = {n, 1 ≤ n ≤ 8}
              & (1 ≤ ∀i,j ≤ 8|i≠j)((u(i)-u(j)) ∉ {0,i-j,j-i})
      rows = {n | 1 ≤ n ≤ 8};  posns = nℓ;  n = 1;
      while n ≤ 8;  ⊨ posns = soln(1:n-1)
           ⊢ soln(n)∈(rows-{posns(k) + tilt * (n-k),
                                   1≤k<n, tilt ∈ {0,1,-1}})
           posns(n) = soln(n);
           n = n+1;
      end while;
```

Now since we assume that the eight-queens problem has a solution, the first statement of (20) is not a stopping place, and thus (20) possesses no stopping place. It is equally clear that (20) terminates; thus the end of (20) is reachable. But by virtue of the assertions following the while statement in (20), (20) clearly has the following generalization:

```
(21)  soln = u where range(u) ⊆ {n, 1≤n≤8} & domain(u)={n,1≤n≤8}
            & (1≤∀i,j≤8|i≠j)((u(i)-u(j)) ∉ {0,i-j,j-i});
      rows = {n|1≤n≤8};  posns = nℓ;  n = 1;
      while n ≤ 8;
           posns(n) = ∋(rows-{posns(k)+tilt*(n-k),1≤k<n,
                                        tilt ∈ {0,1,-1}});
           n = n+1;
      end while;
```

Hence the end of (21) is reachable. But in (21) the variable *soln* is dead. Since deletion of an assignment to a dead variable clearly does not spoil reachability, it follows immediately that the end of (19) is reachable.

We conclude the present section with a discussion of the question of the reachability properties of the praa transformations listed in Section (2.c). We have already noted that π is reachable in R if R is valid but becomes invalid if the assertion $\vdash (false)^{\pi}$ is added to R. This makes it plain that all of the *reversible* transformation rules listed in Section (2.c) preserve reachability. This includes Rules (1.a), (1.d), (1.f), (1.g), (2.a-f), (2.k-i) provided that the place π to be reached is not one of those directly involved in the transformation applied, and with the same restriction also (2.m), (2.n), (2.o), (3.a), (3.c-e), (4.a-c). The remaining rules require additional comment:

Ad (1.b) and (1.c): If $e_1 \subseteq e_2$ is known at a place π_- in the praa R, then π_- , $x = \ni e_2$, π_+ can be replaced by π_- , $x = \ni e_1$, π_+ preserving reachability of π, provided that addition of the assumption $\models (x \in e_2 - e_1)^{\pi_+}$ makes it possible to deduce the assertion $\vdash (false)^{\pi}$. Similarly, if $e_1 \in e_2$ is known at π_- , then to replace $x = \ni e_2$ by $x = e_1$ we must be able to show that $\models (x \in e_2 - \{e_1\})^{\pi_+}$ leads to $\vdash (false)^{\pi}$.

Ad (1.e): To give adequate form to this rule, we need to introduce an extended notion of reachability. Let R_1 be a valid praa with variables $x_1,\ldots,x_\ell$, let $n < m < \ell$, let P be a predicate with free variables $x_1,\ldots,x_m$, and let C be a predicate with free variables $u_1,\ldots,u_n,x_1,\ldots,x_m$. Let π be a place in R_1. Then we say that *the triple* (π,C,n) *is P-reachable* if, for all $u_1,\ldots,u_n$, $x'_1,\ldots,x'_m$ satisfying $P(x'_1,\ldots,x'_m)$ & $C(u_1,\ldots,u_n,x'_1,\ldots,x'_m)$, there exists an R_1-computation c terminating at π in whose initial state the variables $x_1,\ldots,x_m$ have the values $x'_1,\ldots,x'_m$ and in whose final state the variables $x_1,\ldots,x_n$ have the values $u_1,\ldots,u_n$.

Now let R, R_1, $\beta_-^{(j)}$, L_j, $j = 1,\ldots,k$ be as in transformation rule (1.e), let $\beta_+^{(j)}$ be the place in R_1 immediately following the label L_j, and let C_j be as in formula (8) of Rule (1.e). Suppose that the assumptions made in connection with Rule (1.e) are all satisfied, and suppose also that at each of the places $\beta_-^{(j)}$, $j = 1,\ldots,k$ in R a proposition P_j with free variables $x_1,\ldots,x_m$ is available, such that the triple $(\beta_+^{(j)}, C_j, n)$ is P_j-reachable. Then rule (1.e) can be applied, with preservation of reachability.

To prove that a triple (π, C, n) is P-reachable, we can generally proceed as follows. Let R be the praa containing π. Introduce variables $x_1',\ldots,x_m'$, $u_1',\ldots,u_n'$ not otherwise occurring, and add the assumption

$$\models (P(x_1',\ldots,x_m') \;\&\; C(u_1',\ldots,u_n', x_1',\ldots,x_m') \;\&\; (x_1' = x_1 \;\&\; \ldots \;\&\; x_m' = x_m))^{\pi_0}$$

to the entry place π_0 of R, thus obtaining a new praa R'. Find a correct praa R" of which R' is a generalization, and which is such that the assertion $\vdash (x_1 = u_1' \;\&\ldots\&\; x_n = u_n')^{\pi}$ can be proved in R".

A less elaborate discussion suffices to dispose of the remaining transformation rules.

Ad (2.g): Let C_1 be as in Rule (2.g). Then Rule (2.g) can be applied, with preservation of reachability, provided that at the place π immediately preceding the go-to statement to be split there are available assertions implying the impossibility of operation errors in the evaluation of the boolean expression C_1.

Ad (2.j): As in Rule (2.j), let s_1 and s_2 be isomorphic sections of a praa R. Then Rule (2.j) can be applied, with preservation of reachability, provided that the place π to be reached does not belong to the code section s_1.

Ad (2.k): As in rule (2.k), let s be an isolated section of a praa R. Then rule (2.k) can be applied, with preservation of reachability, provided that if any statement if C then go to L of s is modified, the place π to be reached does not belong to the code section s.

Ad (2.ℓ): Let the label L be as in rule (2.ℓ). Then rule (2.ℓ) can be applied, with preservation of reachability, provided that the place π to be reached is not the place immediately following the label L.

Ad (3.b): Reachability is preserved by insertion of assignment statement x = *expn* whose target variable is dead, provided that at the point of insertion assertions implying the impossibility of operation errors in the evaluation of *expn* are available. Similarly, reachability preserved by insertion of a selection statement x = $\exists$*expn* whose target variable is dead, provided that at the point of insertion assertions implying the impossibility of operation errors in the evaluation of *expn*, and also implying that *expn* cannot be null, are available.

Ad (4.d): As in rule (4.d), let f and f' be two isomorphic functions. Then provided that the place π to be reached is not a call to f, rule (4.d) can be applied, with preservation of reachability.

Ad (4.e): As in rule (4.e), let f be an n-parameter function of the praa R, let π' be a place in R, and suppose that f allows entry. Let R_1 be obtained from R as follows:

The functions of R_1 are the functions of R, but the main block of R_1 is the body of the function f, with all label and variable names changed to new names, and with each return statement replaced by a statement go to L, where L is a label not otherwise occurring. Let $\bar{\pi}$ be the place following L.

Suppose that P is a predicate whose free variables are the variables of R_1 which correspond to the parameters of f, and suppose also that the triple $(\bar{\pi},P,n)$ is P-reachable in the sense of the discussion concerning (1.e) found a few paragraphs above. Let $expn_j$ be the argument expressions of a function

call which is to be inserted at π'. Suppose that at the place π' there are available assertions implying the impossibility of operation errors in the evaluation of any of these expressions, and also assertions implying the proposition $P(expn_1,...,expn_n)$. Then reachability is preserved even if we insert a function call $x = f(expn_1,...,expn_n)$ at π', provided that the target variable x of the call has no other occurrences in R.

Appendix. Proof rules for other semantic features.

A variety of useful syntactic and semantic features, including label and function variables, procedures able to modify their parameters, procedure variables, and nondeterminism, are available in languages like the one which we have been considering, and it is therefore appropriate to state proof rules which cover these features.

a. Label Variables. If we enumerate the labels $L_1,\ldots,L_n$ occurring in a praa R, we can treat label variables simply as integers. If the labels recurring in a particular function f of R (or in the main block M of R) are $L_i,\ldots,L_j$, then each statement go to x occurring in f (or M) can be treated as if it read

```
if x = i then go to L_i else if x=i+1 then go to L_{i+1} else if
    ... else if x=j then go to L_j else error;
```

where the *error* statement terminates execution.

b. Function Variables. We can treat function variables in a very similar way. If we enumerate the functions $f_1,\ldots,f_n$ occurring in a praa R, and let the number of parameters of f_j be p(j), then function variables can be treated simply as integers. A call $x = y(expn_1,\ldots,expn_m)$ to a function variable y can simply be regarded as an abbreviation for the conditional code sequence

```
        if p(y) = m then go to L2;
L1:     go to L1                          /* a 'stop' statement */
L2:     if y = 1 then
                x = f_1(expn_1,...,expn_m);
        else if n = 2 then
                x = f_2(expn_1,...,expn_m)
        else if ...
```

```
else if y = n then
        x = f_n(expn_1,...,expn_m)
end if;
```

Of course, all functions f, for which $p(j) \neq m$ can and should be omitted from the compound if-statement following label L2.

c. Modifiable Arguments, Procedures, and Procedure Variables.

Provided that parameters are passed by value with return of values at the moment of procedure return, the procedure call

$$p(x_1,\ldots,x_n);$$

can be represented by the equivalent code sequence

$$t = f(x_1,\ldots,x_n)$$
$$x_1 = t(1);\ \ldots,\ x_n = t(n);$$

where the function f is obtained from the procedure p as follows:

i. For $j = 1,\ldots,n$, replace every occurrence of the (modifiable) parameter y_j of p by an occurrence of a new variable y_j' not otherwise occurring;

ii. Insert the group of assignments $y_1' = y_1,\ldots,y_n' = y_n$ at the entry place of p;

iii. Replace every simple return statement in p by the statement return $\langle y_1',\ldots,y_n' \rangle$.

If this transformation is used, then procedure variables can be handled by the function-variable technique sketched in paragraph b.

Other common styles of parameter-passing, e.g., parameters passed by reference, are less readily dealt with. In the worst case, treatment of a feature of this kind may force much of the mechanism of its intended implementation to be made explicit by introduction of a comprehensive mapping of pointers into values.

If this is necessary, application of the proof formalism developed in the preceding pages will become much clumsier. However, it will still be possible to use auxiliary proof rules which establish the semantic equivalence of value and pointer semantics in particular cases to prove the correctness of many algorithms written in a language which makes use either of call-by-reference or other, still more general, pointer mechanisms.

d. Nondeterminism. As already noted, the set-theoretic selection operator $\ni s$ acts nondeterministically. This operator can be used to represent other semantic mechanisms used to introduce nondeterminism into programming languages. For example, a binary primitive function ok which is nondeterministically either *true* or *false* can be written simply as $\ni\{true,false\}$. Hence, if we ignore the issue of termination, the proof and transformation rules set forth above cover deterministic and nondeterministic programs indifferently. However, as indicated in Section 4, the question of termination must be approached in a rather different way for nondeterministic than for deterministic programs.

e. Existential Operator. A powerful and broadly useful programming construct is the existential operator

$$(1) \qquad t = \exists x_1,\ldots,x_n \mid C(x_1,\ldots,x_n) \ ,$$

which both checks for the existence of values $x_1',\ldots,x_n'$ satisfying the condition C and which assigns $x_1',\ldots,x_n'$ to the variables $x_1,\ldots,x_n$ if $x_1',\ldots,x_n'$ can be found. This operator has two useful properties:

(a) It can be expanded, in an obvious way, into a 'search' loop; and

(b) Immediately after (1), the assertion

(2) $\vdash (t \Rightarrow C(x_1,\ldots,x_n)) \;\&\; (\neg t \Rightarrow (\forall x_1,\ldots,x_n)\ \neg C(x_1,\ldots,x_n))$

is available.

The corresponding numerical iterator

(3) $t = m \leq \exists k \leq n \mid C(k)$

can be expanded into a loop which searches in increasing order; moreover, immediately after (3), the assertion

(4) $\vdash (t \Rightarrow C(k) \;\&\; (m \leq \forall j < k)\ \neg C(j)) \;\&\; (\neg t \Rightarrow (m \leq \forall j \leq n)\neg C(j))$

is available.

A third useful property of the existential (3) is that it can be replaced by $t = m_1 \leq \exists k \leq n_1 \mid C(k)$, provided that

(5) $\vdash m \leq m_1 \leq n \;\&\; m \leq n_1 \leq n \;\&\; ((m \leq \forall k \leq m_1)\ \neg C(k)) \;\&\; ((n_1 \leq \forall k \leq n)\ \neg C(k))$

is available at the place preceding (3). (Of course, (1) can also be transformed in similar fashion.)

The simple 'bubble sort' illustrates the use of this broadly useful rule. If we write the bubble sort praa as

(6)
$\models\!\!\models$ vector(v) & range(v) $\subseteq$ reals & v = v'
while $1 \leq \exists k < \#v \mid v(k) > v(k+1)$;
$\vdash 1 \leq \forall j < k \mid \neg(v(j) > v(j+1))$
swap(v(k),v(k+1)); $\vdash$ rangecount(v) = rangecout(v')
$\vdash 1 \leq \forall j < k-1 \mid \neg(v(j) > v(j+1))$
end while;
$\vdash 1 \leq \forall k < \#v \mid v(j) \leq v(j+1)$ & rangecount(v) = rangecount(v').

its correctness is obvious. But then the transformation rule for existentials that has been stated allows us to rewrite the body of (6) as

```
(7)         kk = 1;
            while kk ≤ ∃k < #v | v(k) > v(k+1);
               swap(v(k), v(k+1));
               kk = max(k-1,1);   ⊢ 1 ≤ kk < #v
            end while;
```

Then, if the existential operator is rewritten as a search loop we get

```
(8)         kk = 1;
     Loop:  k = kk;
    Sloop:  if v(k) > v(k+1) then go to Lswap;
            k = k+1;
            if k ≥ #v then go to Lout;
            go to Sloop;
    Lswap:  swap(v(k), v(k+1));
            kk = if k = 1 then 1 else k-1;
            go to Loop;
     Lout:
```

Since the variables k and kk are never alive at the same time, they can be identified, which makes it easy to transform (8) into

```
(9)         k = 1;
    Loop:   if v(k) > v(k+1) then
               swap(v(k), v(k+1));
               k = if k = 1 then 1 else k-1;   ⊢ 1 ≤ k < #v

            else
               k = k+1;
               if k ≥ #v then go to Lout;
            end if;
            go to Loop;
    Lout:   ...
```

which can in turn be converted into the standard form of the bubble sort.

It should be noted that other set-theoretic constructs which expand into stereotyped code sequences and which make available assertions of predictable form can be exploited in similar fashion. This remark applies to 'generator' routines which generate all the elements of some large set one after another, and also to codes which develop solutions to equations of various useful standard forms.

Bibliography

Burstall, R. and Darlington, J. [1975] *Some Transformations for Developing Recursive programs.* Proc. of 1975 International Conference on Reliable Software, pp. 465-472.

Cohen, P. [1966] *Set Theory and the Continuum Hypothesis.* Mathematics Lecture Note Series, W. A. Benjamin Publ. Co., New York.

Floyd, R. [1967] *Assigning Meanings to Programs.* Proc. Symp. Appl. Math., American Math. Society, v. 19.

Gerhart, S. [1975a] *Correctness-Preserving Program Transformations.* Symposium on Principles of Programming Languages, pp. 54-66. Also available as Tech. Rep. CS-1975-4, Computer Sci. Dept., Duke Univ., Durham, N.C.

[1975b] *Knowledge abouy Programs: A Model and Case Study.* Tech. Rep. CS-1975-1, Comp. Sci. Dept., Duke Univ., Durham, N.C.

[1976] *Proof Theory of Partial Correctness Verification Systems.* Tech. Rep. CS-1976-5, Comp. Sci. Dept., Duke Univ., Durham, N.C.

Hoare, A. [1969] *An Axiomatic Basis for Computer Programming.* Comm. ACM, v. 12, pp. 576-580.

[1971] *Procedures and Parameters: An Axiomatic Approach.* Symposium on the Semantics of Algorithmic Languages (ed. Engeler, E.). Springer Lecture Notes on Mathematics No. 188, pp. 102-116, Springer-Verlag, Germany.

[1972] *Proof of Correctness of Data Representations.* Acta Informatica, v. 1, pp. 271-280.

London, R. [1 70] *Bibliography on Proving the Correctness of Computer Programs.* Machine Intelligence 5 (eds. Meltzer, B. & Michie, D.) pp. 569-580. American Elsevier Publ. Co., N.Y.

[1972] *The Current State of Proving Programs Correct.* Proc. ACM Annual Conf. 1972, pp. 39-46.

Misra, J. [1975] *Relations Uniformly Conserved by a Loop.* IRIA Conference on Proving and Improving Programs, pp. 71-80. Arc et Senans, France.

Morris, J. H., and Wegbreit, B. [1976] *Subgoal Induction.* Tech. Rept., Xerox Palo Alto Research Cr., Palo Alto, Cal.

Sites, R. [1974] *Proving that Computer Programs Terminate Cleanly.* Ph.D. Thesis, Stanford Univ.; also Comp. Sci. Rept. STAN-CS-74-418.

Walker, S. and Strong, H. R. Jr. [1972] *Characterization of Flowchartable Recursions.* IBM Res. Rep. RC-3844. T. J. Watson Res. Lab., Yorktown Heights, New York.

Wegbreit, B. [1974] *The Synthesis of Loop Predicates.* Comm. ACM v. 17, pp. 102-112.

[1976] *Verification of Program Performance.* Tech. Rept., Xerox Palo Alto Research Center, Palo Alto, Calif.

Appendix. Derivation of a related group of searching praas.
(by Edith Deak)

The following example illustrates the derivation of several functionally related algorithms from a common root praa using techniques discussed in Sections 1 and 2. All of the praas presented search for an element R in a sorted array A, and realize the output assertion:

$$\vdash \text{found} \equiv (1 \leq \exists k \leq h)\ i=k\ \&\ A(k) = R$$

We first present a general, high level root praa and then show how three more specific algorithms can be derived from it by applying correctness-preserving transformations. The success of this method depends on the nonprocedural nature of the where diction and the block substitution rule.

A. Root praa for searching a sorted array.

$$\models A \in \text{vector(reals)}\ \&\ \#A = n\ \&\ \text{Real}(R)\ \&\ \text{sorted}(A)\ \&\ n>0$$

π_1 found = false;
π_2 ℓ = 1;
π_3 u = n;
π_4 L1: $\vdash (1 \leq \forall j < \ell)\ A(j)<R\ \&\ (u < \forall j \leq n)\ A(j) > R$
π_5 if u < ℓ go to L3;
π_6 i = j where $\ell \leq j \leq u$;
π_7 if A(i) = R go to L2;
π_8 if A(i) > R then
π_9 u = i - 1;
π_{10} else
π_{11} ℓ = i + 1;
π_{12} end if;
π_{13} go to L1;
π_{14} L2: $\vdash (1 \leq \exists k \leq n)\ i = k\ \&\ A(k) = R$
π_{15} found = *true*;
π_{16} L3: $\vdash \text{found} \Leftrightarrow (1 \leq \exists k \leq n)\ i = k\ \&\ A(k) = R$

Several search algorithms can be obtained by specifying how i at π_6 is to be selected.

B. Sequential search praa.

To derive a sequential search, we make an obvious choice, setting i = ℓ, and perform the following transformations of the root praa. (Justification is given for each transformation, all of which are either rules mentioned in Section 2 or standard optimization techniques.)

(i) Replace π_6 by i = ℓ; (block substitution rule)

(ii) Move π_5 to the end of all predecessor blocks (all predecessors are single exit)

(iii) Replace uses of u and i by n and ℓ respectively (equality substitution)

The resulting praa is:

$\models$ A $\in$ vector(reals) & #A = n & Real(R) & sorted(A) & n > 0

```
π1          found = false;
π2          ℓ = 1;
π3          u = n;
π4          if n < ℓ go to L3;
π5     L1;
π6          i = ℓ;
π7          if A(ℓ) = R go to L2;
π8          if A(ℓ) > R then
π9               u = ℓ-1;
π10              if ℓ-1 < ℓ go to L3;
π11         else
π12              ℓ = ℓ+1;
π13              if u < ℓ go to L3;
π14         end if;
π15         go to L1;
π16    L2:
π17         found = true;
π18    L3: |- found ≡ (1 ≤ ∃k ≤ n) i = k & A(k) = R
```

π_{10} then becomes a go to statement, which makes u at π_9 dead. Then u at π_{13} is equal to n, and so π_3 becomes dead. We have now:

$\models$ A $\in$ vector(reals) & #A = n & Real(R) & sorted(A) & n > 0

π_1 found = false;
π_2 ℓ = 1;
π_3 if n < ℓ go to ℓ3;
π_4 L1:
π_5 i = ℓ;
π_6 if A(ℓ) = R go to L3;
π_7 if A(ℓ) > R go to L3;
π_8 ℓ = ℓ+1;
π_9 if n < ℓ go to L3;
π_{10} go to L1;
π_{11} L2:
π_{12} found = true;
π_{13} L3: $\vdash$ found $\equiv$ (1 $\leq$ $\exists$k $\leq$ n) i = k $\wedge$ A(k) = R

Then, to eliminate i, note that

$$\vdash [(1 \leq \exists k \leq n)\ i = k\ \&\ A(k) = R \Rightarrow i = \ell]^{\pi_{13}^{+}} \Rightarrow$$

$$\vdash [\text{found} \equiv (1 \leq \exists k \leq n)\ \ell = k\ \&\ A(k) = R]^{\pi_{13}^{+}}$$

C. Binary search praa

A binary search procedure is obtained immediately by using the block substitution rule to replace π_6 by the statement:

$$i = \lfloor(\ell + u)/2\rfloor;$$

D. Fibonaccian search praa

We now derive the Fibonaccian search procedure described in Knuth, Vol.3, p. 415, from our original root praa. We refer the reader to the Fibonacci tree shown by Knuth on p. 415 as an aid to understanding what follows. The algorithm computes the next value of i without any multiplication or subsequent divisor.

Values of i are nodes in the Fibonacci tree, obtained by adding or subtracting successively smaller Fibonacci numbers. The algorithm uses two auxiliary variables q and p, which are always consecutive Fibonacci numbers satisfying q < p. As in Knuth's first version of the algorithm, we make the additional assumption that n+1 is a Fibonacci number. If we assume i as the root of the current subtree in the Fibonacci tree being searched, ℓ-1 indexes the left most leaf of that subtree, and u indexes the rightmost leaf of the subtree. It is interesting to note that while the final algorithm uses p and q and not ℓ and u, our root praa is a natural path to derivation of the Fibonaccian search praa. The proof of the Fibonaccian algorithm is facilitated by putting it in this framework.

We are going to assume that our verification system has been supplied with information concerning some basic mathematical properties of Fibonacci numbers. Let the function fib(k) specify the k-th Fibonacci number, so that fib(0) = 0, fib(1) = 1, fib(2) = 1, etc. Variables i, p, and q are initialized as follows:

```
i = fib(k)  where  n+1 = fib(k+1);
p = fib(k-1) where   i = fib(k);
q = fib(k-2) where   i = fib(k);
```

If A(i) < R, the algorithm moves i down the left branch of the tree, which is done by executing the code:

```
i = i - q;
q = p - q;
p = p - q;
```

If A(i) > q, i moves down the right branch, by executing the code

```
i = i + q;
p = p - q;
q = q - p;
```

We embed these two code fragments in two fragmentary Fibonacci praas, both of which can be seen to preserve the following proposition P under appropriate conditions on p and q:

$$(\exists k)\ p = fib(k)\ \&\ q = fib(k-1)\ \&\ i+p = u+1\ \&\ \ell+q+p = i+1$$

Fib praa 1:

$\models$ P & q > 0
u = i-1;
i = i-q;
q = p-q;
p = p-q;
$\vdash$ P

Fib praa 2:

$\models$ P & p > 1
ℓ = i+1;
i = i+q;
p = p-q;
q = q-p;
$\vdash$ P

The invariance of P can easily be verified by calculating strongest postconditions and then performing appropriate algebraic manipulations.

These two Fibonacci praas embody most of the information concerning the *fib* function that we require.

Returning now to the root praa, the first step of our derivation is to move statements π_5 and π_6 backwards to all predecessor blocks, as was done in section A. This gives

$\models$ A $\in$ vector(reals) & #A = n & Real(R) & Sorted(A) & n > 0

```
π1          found = false;
π2          ℓ = 1;
π3          u = n;
π4          if u < ℓ go to L3;
π5          i = j where ℓ ≤ j ≤ u;
π6    L1:
π7          if A(i) = R go to L2;
π8          if A(i) < R then
π9                u = i-1;
π10               if u < ℓ go to L3;
π11               i = j where ℓ ≤ j ≤ u;
π12         else
π13               ℓ = i+1;
π14               if u < ℓ go to L3;
π15               i = j where ℓ ≤ j ≤ u;
π16         end if;
π17         go to L1;
π18   L2:
π19         found = true;
π20   L3:   ⊢ found ≡ (1 ≤ ∃k ≤ n) i = k & A(k) = R
```

We then observe that proposition P $\Rightarrow$ $\ell \leq i \leq u$. The Fibonacci search praas given above can therefore be substituted for the where statements in our search praa, using the block substitution rule.

The next version of the search praa is obtained by the following sequence of transformations:

(i)	strengthen initial assumption, adding (∃k > 2) n+1 = fib(k)	
(ii)	replace initialization of i at π_5	(block substitution)
(iii)	initialize p and q	(shadow variable)
(iv)	replace use of u at π_{10} by i-1	(equality substitution)
(v)	replace π_{11} by fib praa 1	(code block substitution)
(vi)	delete π_9	(dead code elimination)
(vii)	replace use of ℓ at π_{14} by i+1	(equality substitution)
(viii)	replace π_{15} by fib praa 2	(code block substitution)
(ix)	delete π_{13}	(dead code elimination)

The result is:

$\models$ A $\in$ vector(reals) & #A=n & real(R) & sorted(A) & ($\exists$k>2) k+1 = fib(k)

```
π1          found = false;
π2          ℓ = 1;
π3          u = n;
π4          if u < ℓ go to L3;
π5          i = fib(k) where k+1 = fib(k+1);
π6          p = fib(k-1) where i = fib(k);
π7          q = fib(k-2) where i = fib(k);
π8    L1:
π9          if A(i) = R go to L2;
π10         if A(i) < R then
π11               if i-1 < ℓ go to L3;
            ⊨ P & q > 0
π12               u = i-1;
π13               i = i-q;
π14               q = p-q;
π15               p = p-q;
π16         else
π17               if u < i+1 go to ℓ3;
            ⊨ P & p > 1
π18               ℓ = i+1;
π19               i = i+q;
π20               p = p-q;
π21               q = q-p;
π22         end if;
π23         go to L1;
π24   L2:
π25         found = true;
π26   L3: ⊢ found ≡ (1 ≤ ∃k ≤ n) i = k & A(k) = R
```

P is loop invariant, and therefore the assumptions $\models P$ at π_{12}^{+} and π_{17}^{+} can be degraded to assertions.

Next, we want to eliminate ℓ and u from the program. First observe that

$$i-1 < \ell \equiv i \leq \ell \equiv q+p-1 \leq 0 \quad \text{since} \quad \ell+q+p = i+1.$$

Since q and p are consecutive Fibonacci numbers, this can only occur when $q = 0$. Therefore $i-1 < \ell \equiv q \leq 0$. Similarly, $u < i+1 \equiv u \leq i \equiv p \leq 1$ since $i+p = u+1$. This justifies the following sequence of transformations:

(i) replace π_{16} by if $q \leq 0$ go to L3; (equivalence substitution)

(ii) degrade $\models (q > 0)^{\pi_{11}^{+}}$ to an assertion

(iii) replace π_{17} by if $p \leq 1$ go to $\ell 3$; (equivalence substitution)

(iv) degrade $\models (p > j)^{\pi_{17}^{+}}$ to an assertion

(v) replace π_4 by if $n < 1$ go to $\ell 3$; (equality substitution)

(vi) delete π_4 since $n < 1$ is false

(vii) delete π_2, π_3, π_{12}, π_{18} (shadow variable rule)

The resulting program, shown just below, is a correct Fibonacci search praa:

$\models$ A $\in$ vector(reals) & #A=n & real(R) & sorted(A) & ($\exists$k>2) n+1=fib(k)

```
     found = false;
     i = fib(k)  where  n+1 = fib(k+1);
     p = fib(k-1) where   i = fib(k);
     q = fib(k-2) where   i = fib(k);

L1:  if A(i) = R go to L2;
     if A(i) < R then
          if q ≤ 0 go to L3;
          i = i-q;
          q = p-q;
          p = p-q;
     else
          if p ≤ 1 go to L3;
          i = i+q;
          p = p-q;
          q = q-p;
     end if;
     go to L1;
L2:
     found = true;
L3:
```

$\vdash$ found $\equiv$ (1 $\leq \exists k \leq$ n) i = k & A(k) = R

METAMATHEMATICAL EXTENSIBILITY FOR THEOREM VERIFIERS AND PROOF-CHECKERS

Martin Davis and Jack Schwartz

A full-blown program verification technology must rest on more than the informal or semiformal type of reasoning customary in ordinary published mathematics, since reasoning of this type does not prevent numerous small errors from intruding into proofs. For this reason, any fully satisfactory program verification technology will have to make use of proofs which are expressed in computer readable form and which are then certified formally by a programmed proof-checker or theorem prover. The proof-checker system used will then play the role of a fundamental verification yardstick, and must therefore meet very stringent (albeit only manual) standards of verification. On the other hand, a central aim of verification technology is to reduce the cost of program verification drastically, and thus use of a single inextensible verification formalism will be self-defeating. It is therefore interesting to note that in ordinary mathematical practice, expressions of proofs are greatly facilitated by the availability of metamathematical extension mechanisms. A familiar example of this kind of metamathematical extension justifies the ordinary habit of using a predicate-calculus statement of the associative and distributive laws to set up an algebraic formalism and then of accepting algebraic calculations in lieu of detailed predicate calculus proofs.

These considerations show that in establishing a verification mechanism suitable for long-term reliable use, we will need to define a system having the following three properties:

(A) SOUNDNESS. The system must be capable of verifying the correctness of mathematical proofs, and of maintaining a library of theorems for which correct proofs have already been supplied. We must be entirely convinced that any proof of a theorem which the system certifies as correct should indeed be so.

(B) EXTENSIBILITY. It should be possible to augment the system by adding new symbols, schemes of notation, and extended rules of inference of various kinds (e.g. rules allowing proof

by algebraic or other formal computation to be incorporated into what is originally a system containing predicate calculus statements of the commutative, associative, and distributive laws.)

(C) STABILITY. The changes to the system envisioned in (B) must not alter the soundness demanded in (A).

It is clear that stability is crucial to the long-term success of verification systems. Uncontrolled insertion of unverified, even if plausible, new proof methods can be entirely fatal to the usability of such a system. A verification system guards against the possibility of an incorrect statement entering into its library of verified statements by refusing to admit a statement into this library unless a proof of it has been accepted by the system. In order to use a similar technique to guard against the introduction of unsound proof methods, it is necessary to fully formalize our metamathematics. Then, for each proposed new method of proof, we can form a 'justifying sentence' which asserts that anything which can be proved using the new method was already provable before its introduction. The system will then accept a new proof method only if it succeeds in checking a proof (supplied to the system) of this justifying sentence.

In this paper, we will present a logical prototype of such a system; will then describe the way in which it could accept general notational extensions of its initial proof formalism; will touch upon some of the logical issues which arise in connection with the 'computerization' of such systems; and will analyze some of the metamathematical questions raised by the method used to achieve extensibility.

In our initial analysis, we shall deliberately impose very drastic restrictions on the programming environment which supports the extensible proof checker systems we consider, so as to postpone certain technical considerations which would otherwise have to be faced immediately. However, in a final section, we will extend our initial analysis by considering the issues which need to be faced in order to extend our initial rudimentary programming environment to one in which more adequate computing mechanisms, programs, and programming languages can be used.

2. A Formal System (FS)

We work with a formal system (or theory) FS which is suitable for the formalization of substantial portions of ordinary mathematics. (To simplify our exposition, a powerful but rather minimal formal system will be used.) Without attempting to specify FS completely, we assume:

(a) FS contains the usual predicate logic (i.e., the first order predicate calculus). The expressions (terms and formulas*) of FS are character strings on a finite alphabet consisting of alpha-numeric characters ordinarily available in computer systems, and have a convenient syntax of the kind ordinarily used in programming languages. (Of course this implies that the variables of FS are represented by character-string names rather than separate symbols.)

(b) There is a term of FS, which we write $\emptyset$, whose intended interpretation is the empty set; for each pair of terms α,β of FS, there are formulae $\alpha \in \beta$, $\alpha = \beta$, and terms $\{\alpha,\beta\}$, $\alpha-\beta$, $\alpha \cup \beta$, and $P(\alpha)$ (this last designating the 'power set' or set of all subsets of α). Within FS there exist axioms implying that these formulas and terms have all their ordinary set-theoretic properties. We write $\{\alpha\} = \{\alpha,\alpha\}$. Following von Neumann, we recursively identify each nonnegative integer with the set of all nonnegative integers preceding it, i.e. $0 = \emptyset$, $n+1 = n \cup \{n\}$. Thus each nonnegative integer is identified with a term of FS. (This identification plays no essential role in which follows, but does simplify our exposition.) Also there is a term ω of FS whose intended interpretation is the set of nonnegative integers, such that for each nonnegative integer the formula $n \in \omega$ is provable in FS.

(c) We write $\langle\alpha,\beta\rangle = \{\{\alpha\},\{\alpha,\beta\}\}$, thus using the Wiener-Kuratowski definition of ordered pair. Proceeding recursively, we set $\langle\alpha\rangle = \alpha$, $\langle\alpha_1,\ldots,\alpha_n\rangle = \langle\alpha_1,\langle\alpha_2,\ldots,\alpha_n\rangle\rangle$. We also write $[\alpha_1,\ldots,\alpha_n] = \langle n,\alpha_1,\ldots,\alpha_n\rangle = \langle n,\langle\alpha_1,\ldots,\alpha_n\rangle\rangle$. This latter "n-tuple" has the virtue that its length is unambiguously determined. We assume that for each term α of FS, there is a term $\mathrm{Len}(\alpha)$ such that the equation $\mathrm{Len}(\alpha) = n$ is provable in FS

* Formulas are sometimes called w.f.f.'s.

for some nonnegative integer n, and such that whenever the equation $\alpha = [\alpha_1,\ldots,\alpha_n]$ is provable in FS, so is the equation $\mathrm{Len}(\alpha) = n$.

(d) For each pair α,β of terms there is a term $\alpha||\beta$. When the equations $\alpha = [\alpha_1,\ldots,\alpha_n]$, $\beta = [\beta_1,\ldots,\beta_m]$ are provable in FS, so is the equation $\alpha||\beta = [\alpha_1,\ldots,\alpha_n,\beta_1,\ldots,\beta_m]$.

(e) For each pair of terms α,β there is a term $\alpha(\beta)$. Whe ever the equations $\alpha = [\alpha_1,\ldots,\alpha_n]$ and $\beta = m$ with $1 \leq m \leq n$ are provable in FS, so is the equation $\alpha(\beta) = \alpha_m$.

(f) For each pair of terms α,β, there is a term $\alpha|\beta$ such that whenever the equation $\beta = m > 1$ is provable in FS, so is the equation

$$\alpha|\beta = [\alpha(1),\alpha(2),\ldots,\alpha(m)] .$$

By (e), this implies that if the equation $\alpha = [\alpha_1,\ldots,\alpha_n]$ is provable in FS with $m \leq n$, then so is the equation $\alpha|\beta = [\alpha_1,\alpha_2,\ldots,\alpha_m]$.

(g) For each pair of terms α,β there is a term $\alpha + \beta$. When the equations $\alpha = m$, $\beta = n$ are provable in FS, so is the equation $\alpha + \beta = k$ where k is the ordinary integer sum of m and n.

(h) For each term α of FS, there are terms $R(\alpha)$, $\mathrm{Lev}(\alpha)$. The sentences $R(0) = \emptyset$, $(\forall n)(n \in \omega \rightarrow R(n+1) = P(R(n)))$ and $(\forall x)(\forall y)(\mathrm{Lev}(x) \leq y \leftrightarrow x \in R(y))$ will all be provable in FS. Intuitively the elements of $R(n)$, for n a nonnegative integer, are those which can be built up in at most n stages beginning with $\emptyset$, where at each stage one is permitted to form any (necessarily finite) set using the elements produced at an earlier stage.

(i) If the formulas $\phi(1),\phi(2),\ldots,\phi(n-1)$ are all provable in FS, then so is the formula

$$(\forall j)_{1 \leq j < n}\ \phi(j) .$$

Now we specify a subsystem LFS of FS within which all sentences (i.e. formulae without free variables) will be routinely decidable by a finite procedure. The terms of LFS are the smallest class T of terms of FS containing the term $\emptyset$ and the variables of FS which is such that whenever α,β are in T, so are $\{\alpha,\beta\}$, $(\alpha \cup \beta)$, $P(\alpha)$, $\mathrm{Len}(\alpha)$, $(\alpha||\beta)$, $\alpha(\beta)$, $(\alpha|\beta)$, $(\alpha+\beta)$, $R(\alpha)$, and $\mathrm{Lev}(\alpha)$. The formulae of LFS are the smallest class F of

of formulae of FS containing all formulae $(\alpha = \beta)$ and $(\alpha \in \beta)$ where α and β are terms of LFS, and which is such that:

(i) whenever ϕ and δ are in F so are $\sim\phi$, $(\phi \ \& \ \delta)$, $(\phi \vee \delta)$, $(\phi \rightarrow \delta)$, and $(\phi \leftrightarrow \delta)$.

(ii) whenever ϕ is in F, X is a variable of FS and α is a term of LFS not containing X, then $(\exists X)(X \in \alpha \rightarrow \phi)$ and $(\exists X)(X \in \alpha \ \& \ \phi)$ are in F.

We speak of terms containing no variables as constant terms. It is clear that the constant terms of LFS (often simply called constants) denote finite data objects. Using this fact we can describe a systematic algorithm which applied to any sentence Φ of LFS returns one of the truth values true or false. This algorithm can be described recursively in terms of the total number of occurrences of the symbols $\sim$ & $\vee$ $\rightarrow$ $\leftrightarrow$ $\forall$ in Φ. If this number is 0, Φ must have the form $(\alpha \in \beta)$ or $(\alpha = \beta)$ where α, β are constants; and hence Φ can be tested in a finite number of steps. If this number is greater than 0 and Φ has one of the forms $\sim\phi$, ϕ & δ, $\phi \vee \delta$, $\phi \rightarrow \delta$ or $\phi \leftrightarrow \delta$, the truth value of Φ can be computed as a Boolean combination of the truth values of ϕ and δ. Finally if Φ has one of the forms

$$(\forall X)(X \in \alpha \rightarrow \phi(X)) \qquad \text{or} \qquad (\exists X)(X \in \alpha \ \& \ \phi(X))$$

the truth value of Φ can be computed from a finite number of truth values of sentences $\phi(\alpha_1), .., \phi(\alpha_n)$.

We assume that FS is sufficiently powerful to permit formalization of the finite computations needed to obtain the truth values of sentences of LFS, i.e. that for every such sentence Φ for which the computed value is true, Φ is provable in FS. Or, as we may say, we assume that FS is complete with respect to sentences of LFS. It is a routine exercise to verify that any of the usual ways of specifying FS satisfy this completeness condition.

It follows from this completeness condition that for every constant α, the sentence $\alpha \in R(n)$ is provable in FS for some nonnegative integer n (indeed for all sufficiently large n).

It is a familiar fact (first used by Gödel in his famous work on undecidability) that the expressions of FS can be coded by nonnegative integers. It will be more convenient for our purposes to use the class of constants (which of course includes terms denoting the nonnegative integers) as codes. Thus, first letting each character of the alphabet FS be coded by a unique integer, any expression consisting of a finite sequence of such symbols (and hence any term or formula of FS) can simply be coded by an n-tuple of integers in the obvious way. We assume that such a code has been set up in some definite way, and write $\bar{\lambda}$ for the constant which codes the term or formula λ of FS. Once such an encoding has been chosen, it is a straightforward task to describe the syntax of FS within LFS. Thus, e.g., we can find rather short formulae TERM(X),FORM(X),SENT(X) of LFS containing one free variable, such that if α is any constant then TERM(α), FORM(α), and SENT(α) respectively have the truth value <u>true</u> when $\alpha = \bar{\lambda}$, for λ a term, formula, or sentence (i.e. formula with no free variables) of FS, respectively. Note that in developing the first two of these formulae, it will be necessary to convert the recursions which would appear in their most direct expression into references to underlying sequences of objects. The needed set-theoretic bounds on these sequences can be expressed using constants of the form R(β).

It is useful for later purposes to extend the preceding considerations by noting that there exist a pair of formulas TRU(X) and FAL(X) of FS (but not of LFS) which express the conditions that X is the encoding of a true (resp. false) sentence of LFS. To construct these formulas, we first construct a formula IS_VALUE(X,Y) of LFS such that IS_VALUE(γ,β) is true for given constants γ,β just in case γ has the form $\bar{\alpha}$ and the equation $\alpha = \beta$ is true. Our technique for doing this is simply to reexpress the obvious recursive definition of the value designated by a term α of LFS, replacing the recursion that this definition involves by the statement that an appropriate sequence of recursive steps exists. Here we recall that this standard

technique always allows recursions to be replaced by references to sequences. However, in order to be sure that there is a formula of LFS obtained by applying the technique to a given recursion, we must be sure that an *a priori* bound for the length of the recursion and of $Lev(\alpha)$ for each component α of the corresponding sequence can be given by applying R, lev, +, etc., to the free variables of the recursively defined predicate we are trying to express. We leave it to the reader to check that in the case of the predicate IS_VALUE(X,Y) such a bound can be given.

Next, using IS_VALUE, we create two additional formulae of LFS, called TERM_IS_MEMB(X,Y) and TERM_IS_EQ(X,Y) respectively. The first (resp. the second) of these is to be true for given constants γ_1, γ_2 if and only if these have the form $\bar{\alpha}, \bar{\beta}$, where α and β are terms of LFS and $\alpha \in \beta$ (resp. $\alpha = \beta$) are true. These formulae can of course be written as

(*) TERM_IS_MEMB(X,Y)
$\leftrightarrow$ (∃U ∈ BOUND(X), V∈BOUND(Y))(IS_VALUE(X,U)
& IS_VALUE(Y,V) & U ∈ V),

etc. where the BOUND needed so that (*) shall be a legitimate formula of LFS can readily be given in terms of R, Lev, etc. Next, using these two formulae, we construct formulae TR_HT(X,Z) and FA_HT(X,Z) of LFS such that for a given nonnegative integer n and constant α, TR_HT(α,n) (resp. FA_HT(α,n)) is true just in case $\alpha = \bar{\phi}$, where ϕ is a true (resp. false) sentence of LFS containing no more than n occurrences of the symbols $\sim$, &, $\vee$, $\rightarrow$, $\leftrightarrow$, $\forall$, and $\exists$. These formulas can be written out by making explicit the recursive algorithm used in computing the truth value of sentences of LFS, as described above (and of course by replacing a recursion by reference to an appropriate sequence). Note that in describing the truth value of a formula of the form $(\forall X)(X \in \alpha \rightarrow \phi(X))$ or $(\exists X)(X \in \alpha \rightarrow \phi(X))$, we must again make use of the fact that an a priori BOUND(X) can be written (in terms of R, Lev, etc.) such that

$$IS\_VALUE(\beta,\gamma) \rightarrow \gamma \in BOUND(\beta)$$

is true for all constants α and β.

Finally we define

$$TRU(X) \leftrightarrow (\exists Z)(Z \in \omega \ \& \ TR\_HT(X,Z))$$

$$FAL(X) \leftrightarrow (\exists Z)(Z \in \omega \ \& \ FA\_HT(X,Z)) \ .$$

TRU, FAL are of course formulas of FS but not of LFS. (Indeed, it is a consequence of Tarski's famous theorem on the impossibility of self-definition of truth that no formula of LFS could be provably equivalent to TRU in FS.)

We have noted above that there exists a systematic procedure for calculating the truth-value of any sentence Φ of LFS. Accordingly, in what follows we will sometimes wish to regard LFS as a rudimentary 'programming language'. The 'programs' of this language are simply formulae $\Psi(X_1,\ldots,X_n)$, and the only operations available in the 'programming environment' PE which it defines are

(a) An 'input' operation, which 'reads in' an appropriate number n of constants $\alpha_1,\ldots,\alpha_n$ and forms the sentence $\Phi = \Psi(\alpha_1,\ldots,\alpha_n)$ by substitution.

(b) 'Execution', which simply calculates the truth value of Φ in some totally reliable way, and announces this value.

Purely for expository reasons, we shall sometimes wish to pretend in what follows that our programming system can issue various 'confirmation messages' and 'diagnostics'. This is merely a matter of coupling the core logical mechanism PE to an auxiliary system, which can use the elementary true-false indications provided by PE to trigger the issuance of such messages.

In the next three sections, every reference to a 'programming environment' should be understood as meaning PE. In a final section, we shall indicate the way in which the formal verification techniques that we discuss can be used to justify the use of more highly developed programming systems.

3. The System VT.

We now sketch a system VT which can check proofs in FS. VT is to be maintained in a programming environment (e.g., that described at the end of the preceding section) which can accept character strings and sequences of character strings as inputs. Thus formulas of FS can be input to VT in a natural format. VT will maintain two libraries VA (verified assertions) and RI (rules of inference) each capable of being updated by VT itself in a manner described below. VA is a set of sentences of FS, hence of character strings. RI is a collection of 'procedures' each of which is in fact a formula $\Phi(X)$ of LFS containing just one free variable. The sentence $\Phi(\alpha)$ for a given constant α is to be true only when $\alpha = [\bar{\lambda}_1,\ldots,\bar{\lambda}_n,\bar{\phi}]$ for suitable formulae $\lambda_1,\ldots,\lambda_n,\phi$ of FS. When $\Phi([\bar{\lambda}_1,\ldots,\bar{\lambda}_n,\bar{\phi}])$ has the value <u>true</u>, we say that ϕ is a consequence of the premises $\lambda_1,\ldots,\lambda_n$ according to the 'rule of inference' $\Phi(X)$. It will be convenient to assume that all rules of inference in RI satisfy the stipulation that a consequence of a given sequence of premises remains a consequence if the order of premises is altered or if additional premises are supplied. Thus in particular if $\Phi([\bar{\lambda}_1,\ldots,\bar{\lambda}_n,\bar{\phi}])$ is true so will $\Phi([\bar{\lambda}_1,\ldots,\bar{\lambda}_n,\bar{\lambda}_{n+1},\ldots,\bar{\lambda}_m,\bar{\phi}])$ be true.

Initially, RI will contain various procedures which correspond to the axioms and rules of inference of an appropriate version of FS. For example, we might expect to find the following items in RI:

(a) A procedure

AXIOM(X)

where the sentence AXIOM(β) is true precisely when $\beta = [\bar{\lambda}_1,\ldots,\bar{\lambda}_n,\bar{\phi}]$ where ϕ is an axiom of FS.

(b) A procedure

MODUSPONENS(X),

where the sentence MODUSPONENS(β) is true precisely when $\beta = [\bar{\lambda}_1,\ldots,\bar{\lambda}_n,\bar{\phi}]$ where there are i,j, $1 \leq i,j \leq n$ and a formula ψ of FS such that $\lambda_i = \psi$ and $\lambda_j = (\psi \rightarrow \phi)$.

For each fixed value of VA and RI, we define a corresponding notion of proof. (Note that we will shortly describe ways in which VA and RI can be modified; such modifications will of course modify our notion of proof in corresponding fashion.) Namely, π is a <u>proof</u> if

$$\pi = [\bar{\phi}_1,\ldots,\bar{\phi}_n]$$

where for each i, $0 \leq i < n$, either $\phi_{i+1} \in$ VA or ϕ_{i+1} is a consequence of premises $\phi_1,\ldots,\phi_i$ according to some rule of inference in RI. In this case π is said to be a <u>proof of</u> ϕ_n.

We can easily construct a formula PROVE(X,Y) of LFS, containing two free variables, such that PROVE(α,π) is true for given constants α,π just in case $\alpha = \bar{\phi}$ where π is a proof of ϕ. To exhibit the formula PROVE(X,Y), one first compiles the two formulae

$$\text{SOME\_VA}(X) = (X = \bar{\phi}_1) \vee (X = \bar{\phi}_2) \vee \ldots \vee (X = \bar{\phi}_n)$$

and

$$\text{SOME\_RI}(X) = \Phi_1(X) \vee \Phi_2(X) \vee \ldots \vee \Phi_n(X) ,$$

where $\phi_1,\ldots,\phi_n$ are a complete list of the formulas which belong to VA and $\Phi_1,\ldots,\Phi_m$ is a complete list of the procedures in RI. The formula PROVE(X,Y) can now be written as follows:

$$(*) \quad Y(\text{Len}(Y)) = X \;\&\; \text{Len}(Y) \neq 0 \;\&\; (\forall i)_{1 \leq i \leq \text{Len}(Y)}[\text{SOME-VA}(Y(i)) \vee \text{SOME-RI}(Y|i)]$$

It is clear that if π is a proof of ϕ and $\alpha = \bar{\phi}$, then PROVE(α,π) is true. Conversely, if PROVE(α,π) is true, it follows that the constant $[\pi(1),\pi(2),\ldots,\pi(\text{Len}(\pi))]$ is a proof of ϕ where $\bar{\alpha} = \pi(\text{Len}(\pi)) = \phi$.

Note that PROVE(X,Y) is a formula of LFS (the quantifiers and integers of (*) are permitted in LFS because under von Neumann's definition of the natural numbers, $j < i$ and $j \in i$ are equivalent conditions). Thus PROVE(X,Y) may be regarded as defining a strictly finite procedure for testing an alleged proof for really being one.

Writing PR-LEV(X,Z) for the formula $(\exists Y)(Y \in R(Z)\ \&\ \mathrm{PROVE}(X,Y))$, we see that this is also a formula of LFS. Finally, we write THM(X) for the formula

$$(\exists Z)(Z \in \omega\ \&\ \text{PR-LEV}(X,Z))\ .$$

We observe that THM(X) is not a formula of LFS (because of the occurrence of the term ω). Note that if $\mathrm{PROVE}(\alpha,\pi)$ is true for given constants α,π, we will be able to prove $\pi \in H(n)$ in FS for some fixed integer n and hence, using predicate logic, we will be able to prove PR-LEV(α,n) in FS. Using predicate logic again, we see that THM(α) will be provable in FS.

For most versions of FS it will be possible to prove the sentence

$$(\forall X)(\mathrm{THM}(X) \leftrightarrow (\exists Y)\mathrm{PROVE}(X,Y))$$

in FS, but we will not make use of this fact.

4. Modes of Use of VT.

We shall now describe six modes in which VT can be used. Of these, the first two correspond to the normal use of VT as a theorem library and proof checker, while others correspond to the various ways in which we allow VT to modify itself.

Mode I (Lookup): This is a simple library lookup function. In this mode, a sentence ϕ can be presented to VT for verification. If $\phi \in$ VA, VT will return a message such as

(1) ϕ IS IN LIBRARY

Otherwise, VT will return a negative message.

Mode II (Verification): When this mode is engaged, a sentence ϕ and its alleged proof π can be presented to VT as inputs. VT will then translate ϕ into its encoding $\bar{\phi}$ (which simply amounts to coding a character string as a constant) and will then execute the procedure PROV($\bar{\phi}$,π). If the value true is obtained, UT returns an appropriate message, e.g.

α IS VERIFIED .

If the value false is obtained VT should return a suitable diagnostic, e.g.

STEP 5 OF PURPORTED PROOF IS ILL-FORMED

or in other cases something like

STEP 11 OF PURPORTED PROOF DOES NOT FOLLOW FROM PREVIOUS STEPS.

Mode III (Assertion Insertion): This mode is similar to Mode II; however, it is intended that a sentence is both to be verified and to be stored in VA. Hence, after issuing the success message (1), VT must update itself. It does so by inserting α in the library VA. Of course, this makes revision of the procedures SOME_VA(X) and PROVE(X,Y) necessary.* The

* Of course, the overall structure of the PROVE procedure will remain invariant. It is just that SOME_VA(X) will now represent a different formula of LFS.

stability of the system is assured by verification of the proof which was input with α.

Mode IV (Rule Insertion): When the rule is engaged, a new proof-rule Φ, a 'justifying' proposition α, and a proof P can be presented to Φ as inputs. Then VT will first check that α has the form

(2) $(\forall X)([\Phi(X) \rightarrow ([(\forall j)_{1\leq j<Len(X)}\ THM(X(j))] \rightarrow THM(X(Len(X)))])$.

If this check fails, a suitable diagnostic, i.e.

RULE IS NOT CORRECTLY JUSTIFIED

will be emitted. If the check succeeds, VT will proceed to translate α into its encoding $\bar{\alpha}$ and will execute the procedure PROVE($\bar{\alpha}$,P). If the value false is obtained, VT will issue an appropriate diagnostic, as in Mode II usage. Otherwise, if the value true is obtained, VT will issue the message

RULE ACCEPTED ,

and will then adjoin Φ to its collection RI of proof rules. Of course, this makes reconstruction of the procedures SOME_RI(X) and PROVE(X,Y) necessary.

A discussion of the stability of the system VT under Mode IV use will be found in Section 5 below.

We shall now explain how rule IV can be used to extend VT so as to permit algebraic notations, calculations, and modes of reasoning to be used directly. Let us first describe the structure of an extension capable of verifying certain propositions by automatic testing of certain (necessarily limited) classes of algebraic calculations. To develop such an extension, we can write out a formula $\Phi(X)$ of LFS such that $\Phi(\alpha)$ is true for a given constant α if and only if $\alpha = [\bar{\lambda}_1,\ldots,\bar{\lambda}_n,\bar{\phi}]$, where one of the $\bar{\lambda}_j$ encodes a sentence of the form $\tau = \tau$, where τ is in turn the encoding of some algebraic identity σ routinely verifiable by an algorithmic technique (e.g. some special case of the binomial theorem) and where ϕ is a translation of σ into FS. The proof of the justifying sentence for such a $\Phi(X)$ would simply

be a formalization of some standard proof of the correctness of whatever algorithm was used to test such alleged identities. Once VT accepted this new rule, any identity belonging to the class handled by Φ could be introduced as a single step of any proof that required it.

Of course, in practice one would wish to improve the efficiency with which formulae of this class were handled, specifically by replacing the slow general procedure needed to evaluate the truth value of $\Phi(\alpha)$ by an invocation of some equivalent but more efficient algebra 'package' directly executable on a computing mechanism M of known structure. Once we have proved that Mode IV use of VT raises no stability problems, it will be straightforward to justify the use of such compiled 'packages'. But since to do so involves consideration of the somewhat technical issue of program correctness, we put off additional discussion of this issue until Section 6 below.

Now let us consider the way in which VT can be extended to allow algebraic reasoning in an algebraic notation, thus making algebraic techniques of a more 'creative' character available. For this, we need first of all to write out a formula $\Psi(X)$ of FS which expresses the statement "X encodes a true formula of algebra". Technically, this can be put as "X encodes a formula which follows from the axioms of algebra by the rules of algebraic deduction". In addition to this, we need various deduction rules corresponding to the rules of algebraic deduction. These would be expressed by formulae $\Delta_1, \Delta_2, \ldots, \Delta_n$ of LFS, which might respectively correspond to the rules of algebraic substitution, multiplication of equals by equals, solution of algebraic equations, etc. Thus, for example, the first of these might be such that $\Delta_1(\alpha,\beta,\gamma)$ for a given constant if and only if α and β respectively encode algebraic identities of the form a = a', and b = b', whereas γ encodes a proposition of the form c = c' which follows by algebraic substitution of b for one of the variables of a and of b' for the corresponding variable of a'.

To use these formulae, we will have to prove statements of the form $(\forall A,B,C)[(\Psi(A) \ \& \ \Psi(B) \ \& \ \Delta_j(A,B,C)) \rightarrow \Psi(C)]$, etc.

We also require a formula $\Phi(X)$ of LFS, such that $\Phi(\alpha)$ is true for a given constant α if and only if $\alpha = [\bar{\lambda}_1,\ldots,\bar{\lambda}_n,\bar{\phi}]$, where either one of the $\bar{\lambda}_j$ encodes a sentence of the form $\Psi(\tau)$, and where $\bar{\phi}$ is the translation of τ into FS, or vice versa. Then we can make Φ available as a rule of inference by extending VT. The proof of its justifying sentence would simply be a formalization of a standard proof of the fact that the translation into FS of a valid algebraic formula is a theorem of FS, and of the appropriate converse of this statement. Φ can then be used to translate in either direction between sentences of FS and algebraic identities.

Mode V (Define New Function). In this mode it is possible to extend the notation of FS by adding a symbol ϕ for a new function using as "definition" a sentence of FS:

$$(\forall X_1,\ldots,X_n,Y)[Y = g(X_1,\ldots,X_n) \leftrightarrow \Gamma(X_1,\ldots,X_n,y)]$$

where $\Gamma(X_1,\ldots,X_n,Y)$ is a formula of FS.
VT will check that g is indeed a new function symbol and then demand proofs of the justifying sentences:

$$(\forall X_1,\ldots,X_n)(\exists Y)\Gamma(X_1,\ldots,X_n,Y)$$

$$(\forall X_1,\ldots,X_n,Y,Z)[(\Gamma(X_1,\ldots,X_n,Y) \ \& \ \Gamma(X_1,\ldots,X_n,Z) \rightarrow (Y = Z)] \ .$$

If such proofs are provided and are verified by VT, then the defining sentence of g is adjoined to VA and SOME_VA(X) recompiled as in Mode III usage.

Mode VI (Define New Relation): When this mode is engaged, a new symbol S, which does not appear in any of the formulae of VA or RI, is presented to VT, and with it the sentence

$$(3) \qquad (\forall X_1,\ldots,X_n)(S(X_1,\ldots,X_n) \leftrightarrow \Delta(X_1,\ldots,X_n)) \ ,$$

where $\Delta(X_1,\ldots,X_n)$ is a formula of FS containing only symbols which appear in VA or RI, and having no free variables other than $X_1,\ldots,X_n$. VT proceeds immediately to add (3) to VA, and S becomes available for use as an n-parameter predicate symsol of FS as extended. No justifying sentence is needed in this mode.

We end this section with some informal remarks intended to indicate the significance we attach to the extensibility mechanisms that have just been defined. Given any proof verification system V, one can define the *difficulty* of a sentence S to be the length of the shortest input which will cause VT to accept S, or to be ∞ if no such input exists. Even though it will follow by arguments to be offered in the next section that the extension principles we have described can never reduce the difficulty of S from ∞ to a finite quantity (i.e., can never enlarge the set of verifiable sentences), we conjecture that the availability of these extension principles will reduce the difficulty of large classes of sentences by very large amounts relative to what the difficulty of these sentences would have been in any fixed extension of V not containing an extensibility principle or something equivalent to it.

It is known that certain extensions of formal systems (e.g. by large cardinal axioms or so-called reflection principles) which increase the class of decidable arithmetic sentences, also decrease the difficulty of some such sentence by arbitrarily large (recursive) amounts. However, the sentences obtained in this way seem always to have an artificial post-hoc flavor (even though they can always be taken to simply assert the nonsolvability in integers of some Diophantine equation). And in fact, there seem at present to be no general principles, not already available in Zermelo-Fraenkel set theory as formalized within predicate calculus, which promise to enlarge the class of propositions interesting to the working mathematician that can be proved. (Recent work in descriptive set theory, where new axioms have led to real advances, may give something of a counterexample to this assertion. But this work is still quite remote from "everyday" mathematics.)

We can thus state the remarkable and fundamental fact that the very simple formal mechanisms embodied in predicate calculus and the Zermelo-Fraenkel axioms can track mathematical discourse in a quite comprehensive manner. But this tracking has been very much a matter of principle rather than of practice. It is obviously

quite unfeasible, in practical terms, to represent ordinary mathematical discourse in raw formalized Zermelo-Fraenkel set theory. Our extensibility mechanisms are intended to enable one to incorporate the various dictions and methods of ordinary mathematical discourse more comfortably into a fully formal system. On the basis of ordinary mathematical experience we have every reason to expect that the difficulty (in the precise sense we have defined) of various important theorems will be greatly decreased in this way. Although we have been unable to formulate and prove any metatheorems that would serve as a formal demonstration of this conjecture, we can point to some suggestive evidence. It is well known in proof theoretic research that the addition of new rules of inference to so-called cut-free systems can drastically decrease the lengths of proofs. In fact an exponential improvement is obtained (for a suitable class of theorems) for each use of cut rules that is permitted. Analogously, in connection with the system VT sketched above, we have seen that the introduction of an appropriate "algebra" rule of inference shortens to 1 the difficulty of a sentence which asserts an algebraic identity.

We note that attempts over the past few years to develop practical program verification systems have increased the pragmatic urgency of supplying verifiers which keep the difficulty of significant classes of sentences low. Unfortunately, this has tended to lead to the proliferation of numerous incommensurable systems, since different authors have proposed systems designed to lower the difficulty of one or another somewhat special class of sentence. This has in part tended to blunt the thrust of the verification literature. It is rather clear that the extensibility mechanisms described above allow one to start with any convenient one of these formalisms and (after considerable but only finite difficulty) extend it to include any or all of the others. Thus we provide a single mechanism which, if we leave a necessary initial investment out of account, can be made as comprehensive as any of the previously proposed verifiers, each on its own chosen ground.

5. Metamathematical Considerations Concerning the Stability of VT.

After our hypothetical system VT has been in operation for a while, assuming that it has been used in Modes III, IV, V and/or VI, the "proofs" being supplied the system may be syntactically quite different (and hopefully less tedious) from those acceptable to the original system. We shall write $\vdash \alpha$ to indicate that the sentence α of FS was verifiable by VT in its *original version*.

We have been using the notation PROVE(X,Y), PR-LEV(X,Z) and THM(X) ambiguously, since these formulas change as VT modifies itself. To remove this ambiguity we have only to introduce a counter t, writing $\text{PROVE}_t(X,Y)$, $\text{PR-LEV}_t(X,Z)$ and $\text{THM}_t(X)$. We let t be initialized at 0 and assume that whenever VT is used in a Mode which results in modification of PROVE_t, t is incremented by 1. It is obvious that $\text{PROVE}_t(\alpha,\pi)$ implies $\text{PROVE}_{t+1}(\alpha,\pi)$. That is, if VT accepts a sentence (upon presentation of a proof π), it will continue to accept ϕ as it modifies itself (by virtue of the same proof π). The stability result we wish to prove is simply that if for any value of t, $\text{PROVE}_t(\bar{\phi},\pi)$ is true, then $\vdash \phi$. Note that, using the formal techniques introduced earlier, this assertion (for any particular t, representing some particular sequence of extensions of VT), can itself be represented by the sentence of FS:

$$(*) \qquad (\forall X)[\text{THM}_t(X) \rightarrow \text{THM}_0(X)] \ .$$

Hence it might be expected that the stability proof we give could itself be formalized in FS. However, this is not quite the case, and to formalize the proof of (*) in a system like FS we need to use a system which, though it may be considerably weaker than FS in many regards, goes beyond FS in a certain direction. Specificatly, we need to assume the following condition which we call *weak ω-consistency* (because it is implied by Gödel's notion of ω-consistency):

If $\Delta(X)$ *is a formula of LFS, and if* $\vdash (\exists X)(X \in \omega \ \& \ \Delta(X))$, *then there is a natural number* n *such that* $\Delta(n)$ *is true.*

The assertion that FS is weakly ω-consistent can be expressed in FS by the infinite collection of sentences:

(4) $\mathrm{THM}_0(\overline{(\exists X)(X \in \omega \ \& \ \Delta(X))}) \rightarrow (\exists X)(X \in \omega \ \& \Delta(X))$

(or more satisfactorily by a single sentence, which implies all of the sentences of (4)). However there are obviously particular formulas $\Delta(X)$ for which if (4) were a theorem of FS, we should have:

(5) $\qquad \vdash \sim \mathrm{THM}_0(\overline{(\exists X)(X \in \omega \ \& \ \Delta(X))})$

(E.g., take for $\Delta(X)$ the formula $X = 0 \ \& \ X = 1$). But it follows from well known results of Gödel that (5) cannot be proved in FS if FS is a consistent system. (This is because (5) amounts to asserting that the consistency of FS is provable in FS.) This shows clearly that we cannot expect our stability proof to be formalizable in FS. However, the proof we are about to give can be formalized in any system which contains:

(a) a certain relatively weak subsystem of FS, and

(b) additional axioms sufficient to prove all of the sentences of (4).

Having made these preliminary explanations, we shall now proceed to present our formalizable arguments quite informally.

We have:

Lemma 1. If $\mathrm{PROVE}_t(\alpha,\pi)$ is true, then $\vdash \mathrm{THM}_t(\alpha)$.

Proof. We have $\vdash \pi \in R(n)$ for some nonnegative integer n. Therefore by predicate calculus

$$\vdash \pi \in R(n) \ \& \ \mathrm{PROVE}_t(\alpha,\pi)$$

and hence

$$\vdash (\exists Y)(Y \in R(n) \ \& \ \mathrm{PROVE}_t(\alpha,Y)) \ ,$$

i.e.

$$\vdash \mathrm{PR\text{-}LEV}_t(\alpha,n) \ .$$

Since $\vdash n \in \omega$, we can use predicate calculus again to obtain:

$$\vdash (\exists Z)(Z \in \omega \ \& \ \text{PR-LEV}_t(\alpha, Z)) \ ,$$

i.e.,

$$\vdash \text{THM}_t(\alpha) \ .$$

Next we shall need a converse to Lemma 1, and in order to obtain it we will have to use our assumption that FS is weakly ω-consistent.

Lemma 2. If $\vdash\!\!\!-\text{THM}_t(\alpha)$ then there is a constant π such that $\text{PROVE}_t(\alpha,\pi)$.

Proof. We are given

$$\vdash (\exists Z)(Z \in \omega \ \& \ \text{PR-LEV}_t(\alpha, Z)) \ .$$

By weak ω-consistency, for some nonnegative integer n, $\text{PR-LEV}_t(\alpha,n)$ is true. I.e., the sentence of LFS

$$(\exists Y)(Y \in R(n) \ \& \ \text{PROVE}_t(\alpha, Y))$$

is true. Hence for some constant π , $\text{PROVE}_t(\alpha,\pi)$ is true. Q.E.D.

We are now in a position to prove our main result, which gives the stability of VT:

Theorem. If ϕ is a sentence of FS and $\text{PROVE}_t(\bar{\phi},\pi)$ is true then $\vdash \phi$.

Proof. Our proof is by induction on the counter t. The result is obvious for t = 0. We therefore need only verify that the fact asserted is preserved under use of VT in Modes III, IV, V and VI. It is a well known property of predicate calculus that use of the extensions obtained under Modes III, V, and VI do not increase the class of provable sentences. Hence we need only show that the property is preserved under use of VT in Mode IV.

To obtain this result, let us suppose that the transition from t to t+1 was the result of a use of VT in Mode IV. and let ϕ be a sentence of FS such that $\text{PROVE}_{t+1}(\bar{\phi},\pi)$ is true for a given constant π. We shall show that $\vdash \phi$. We write

$\text{SOME-VA}_t(X)$, $\text{SOME-RI}_t(X)$ for the formulas of LFS introduced earlier, showing the counter t explicitly.

We are given the sequence $\pi = [\bar{\phi}_1,\ldots,\bar{\phi}_n]$, where $\phi_n = \phi$ and $\text{PROV}_{t+1}(\bar{\phi},\pi)$ has the value true. We shall show that $\vdash \phi_i$ for $i = 1,2,\ldots,n$. The proof is itself by induction, so in proving the result for i we may assume that it is known for all $j < i$. (Thus the whole of the present proof has the form of a double induction -- an inner induction on j and an outer induction on t.)

Case 1. $\text{SOME-VA}_{t+1}(\bar{\phi}_i)$ is true. Then, since we are assuming that the change to VT involved in the counter increasing to t+1 was not of MODE III, $\text{SOME-VA}_t(\bar{\phi}_i)$ is likewise true. Hence $\text{PROVE}_t(\bar{\phi}_i,[\bar{\phi}_i])$ is true. By induction hypothesis, $\vdash \phi_i$.

Case 2. $\text{SOME-RI}_{t+1}([\bar{\phi}_1,\bar{\phi}_2,\ldots,\bar{\phi}_i])$ *is true.* By induction hypothesis $\vdash \phi_j$ for all $j < i$. So $\text{PROVE}_0(\bar{\phi}_j,\pi_j)$ is true for suitable constants π_j and therefore $\text{PROVE}_t(\bar{\phi}_j,\pi_j)$ is likewise true for all $j < i$.

We consider two subcases:

Case 2a. $\text{SOME-RI}_t([\bar{\phi}_1,\bar{\phi}_2,\ldots,\bar{\phi}_i)]$ *is true.* Then, if we set

$$\pi = \pi_1||\pi_2||\ldots||\pi_{i-1}||[\bar{\phi}_i]$$

we have that $\text{PROVE}_t(\bar{\phi}_i,\pi)$ is true. By induction hypothesis (on t), $\vdash \phi_i$.

Case 2b. $\text{SOME-RI}_t([\bar{\phi}_1,\bar{\phi}_2,\ldots,\bar{\phi}_i])$ *is false.* Then, the transition from t to t+1 involved adjoining a new rule of inference $\Phi(X)$ to RI, and moreover $\Phi([\bar{\phi}_1,\bar{\phi}_2,\ldots,\bar{\phi}_i])$ is true. Since $\Phi(X)$ was accepted by VT, it follows that VT (with index t) must have verified the correctness of the justifying sentence (2). That is, for some constant π,

$$\text{PROVE}_t(\overline{(\forall X)[\Phi(X)\rightarrow([(\forall j)_{1\leq j<\text{Len}(X)}\,\text{THM}_t(X(y))]\rightarrow\text{THM}_t(X(\text{Len}(X)))]},\ \pi)$$

has the value true. By induction hypothesis (on t),

$$\vdash (\forall X)[\Phi(X) \rightarrow ([(\forall j)_{1 \leq j < Len(X)} THM_t(X(j))] \rightarrow THM_t(X(Len(X)))].$$

By the completeness of FS with respect to sentences of LFS

$$\vdash \Phi([\bar{\phi}_1, \bar{\phi}_2, \ldots, \bar{\phi}_i]) .$$

Since $PROVE_t(\bar{\phi}_j, \pi_j)$ is true for all $j < i$, using Lemma 1 we have $\vdash THM_t(\bar{\phi}_j)$, for $j < i$. Therefore, we have also

$$\vdash (\forall j)_{1 \leq j < i} \; THM_t(\bar{\phi}_j)$$

Using predicate calculus we conclude that

$$\vdash THM_t(\bar{\phi}_i)$$

By Lemma 2, there is a constant π such that $PROVE_t(\bar{\phi}_i, \pi)$ is true. Finally, using the induction hypothesis on t, $\vdash \phi_i$. Q.E.D.

6. Program Verification and Extension of the Programming Environment.

In the present section, we shall supplement the preceding analysis by considering additional issues which need to be faced when we wish to extend the rudimentary programming environment used till now, in order to allow use of other computing mechanisms, programs, and programming languages.

It is convenient for us to view a computing mechanism M simply as a system which can accept texts of a particular kind, and which, once having accepted such a text, will use it to establish some kind of initial internal state, following which this internal state will be transformed, cycle by cycle. Such transformation may eventually lead the mechanism to termination. If termination occurs, M will exhibit some part of its internal state, and this exhibited part constitutes its 'output'. Suppose that, as is customary, we always regard the input to such a mechanism as being divided into two distinguishable parts, called 'program' and 'input-data' respectively (though in particular cases either of these may be null). Then we can write YIELDS(program, input_data, output) for the sentence "if the finite objects (and to make contact with the preceding sections, let us agree that these objects will be encoded as constants, i.e. constant terms of LFS) 'program', 'input_data' are presented to M, and if termination occurs, then 'output' will be exhibited".

For us to be willing to use M as part of a formal verification system, we must of course feel that we understand the way in which it operates, and for this to be the case we must be in possession of some formula 'GOESBY' of LFS which describes the internal operation of M, i.e. a formula for which we believe the equivalence

(1) $$\text{YIELDS}(\text{program},\text{input\_data},\text{output}) \leftrightarrow (\exists c)(\exists n)(n \in \omega \;\&\; c \in R(n)) \;\&\; \text{GOESBY}(\text{program},\text{input\_data},\text{output},c))$$

to be correct. We can regard (1) either as a mathematical definition of an (idealized) mechanism M, or equivalently

(at least for our purposes) as an axiom about the behavior of a certain physical object (namely an actual computer of some particular type).

To use the computing mechanism M as part of a verification system like our VT, we will generally need to make use of a program P_0 for it which can cause M to calculate the truth value of some class of sentences of LFS. In order that P_0 can be used reliably, we must know that P_0 is 'correct', i.e. that if P_0 processes a sentence S of LFS and terminates with the output 'true' (resp. 'false') then the truth value of S is indeed 'true' (resp. 'false'). (Note however that P_0 need not be capable of processing every sentence S of LFS, i.e. for some input S, P_0 may yield the result 'can't handle this input', or may simply fail to terminate). We can express the condition that P_0 be correct as a pair of statements of FS. In writing these statements, we shall suppose that we have already constructed a formula SENT(X) of LFS such that SENT(α) is true for a constant α just in case $\alpha = \bar{\phi}$, where ϕ is a sentence of LFS. We will also make use of the sentences TRU(X) and FAL(X) constructed in section 2 above. (Note again that these are sentences of FS but not of LFS.) Then the correctness assertion needed to justify use of the program P_0 is expressed by

(2a) $(\forall X)(\text{SENT}(X) \;\&\; \text{YIELDS}(\beta,X,\text{true}) \rightarrow \text{TRU}(X))$

(2b) $(\forall X)(\text{SENT}(X) \;\&\; \text{YIELDS}(\beta,X,\text{false}) \rightarrow \text{FAL}(X))$,

where β is a constant which doees the program P_0.

In practice one will want to lighten the considerable labor of proving the correctness of programs like P_0 by making use of auxiliary program verification systems. This implies a less direct approach to proof of the correctness of P_0 than that which we have just outlined. The following considerations show how such systems can be used, ultimately to establish assertions like (2a) and (2b). We can regard any such program verification system (PVF) as being described by a formula

IS_VERIF(X,Y,Z) of LFS containing exactly three free variables. The sentence IS_VERIF(α,β,γ) is to be true for given constants α,β,γ just in case α is the code for an object $\mathcal{O}$ which PVF will 'accept' or 'verify', β codes the 'program part' of $\mathcal{O}$, and γ encodes the 'input-output assertion part' of $\mathcal{O}$. (The objects α will often belong to some formal system of extended or annotated programs which PVF is able to handle, i.e., they can be program texts 'decorated' with Floyd assertions. E.g., in the verification formalism of [Schw 1], such objects would be 'praas'. Generally, the 'program part' of such an object will simply be the program text which it contains, stripped of the decorating annotations attached to this text within $\mathcal{O}$. Note that this program part is assumed to encode a program P, possibly in a high level language of which some subset is directly acceptable to the computing mechanism M.) The part γ of $\mathcal{O}$ is assumed to be the encoding $\overline{\phi(X,Y)}$ of a formula of LFS which expresses some relationship between P's output Y and its input X which PVT verifies as part of the verification of $\mathcal{O}$. Since we allow the program part of $\mathcal{O}$ to be the encoding of a high level language, only a subset of which gives valid M-programs, we will also need a formula IS_MPROG(X), such that IS_MPROG(β) is true for a given constant β if and only if β codes a program of the restricted form acceptable to M. Finally we will need a formula SUBST(X,U,V,W) expressing the relationship the sentence W arises by substitution of the particular constants U,V into the (two-parameter) formula X". That is, SUBST(α,γ,δ,β) is to be true for given constants α,γ,δ,β just in case $\alpha = \overline{\phi(X,Y)}$ for some formula $\phi(X,Y)$ of FS and $\beta = \overline{\phi(\gamma,\delta)}$. Then, in order to justify use of PVF, we will need to prove the following theorem (for clarity, we shall give multicharacter mnemonic names to the variables occuring in it):

(3) $(\forall X, PROG, PROP, INP, OUP, ASRT)[(IS\_VERIF(X,PROG,PROP)$ &
$IS\_MPROG(PROG)$ & $YIELDS(PROG, INP, OUP)$
& $SUBST(PROP, INP, OUP, ASRT) \rightarrow TRU(ASRT)]$

Once (3) has been proved, any P_0 coded by a constant β satisfying $IS\_MPROG(\beta)$ for which PVF has been used to establish $IS\_VERIF(\alpha,\beta,\gamma)$, where $\gamma = \overline{\phi(X,Y)}$, and where we also have

(4a) $(\forall X)[(SENT(X) \ \& \ \phi(X,true) \rightarrow TRU(X)]$

and

(4b) $(\forall X)[(SENT(X) \ \& \ \phi(X,false)) \rightarrow FAL(X)]$

can be executed on M with $\bar{\Phi}$ as input to evaluate the truth value of Φ. We note again that a given P_0 used in this way need not be capable of handling all Φ; for some inputs Φ if the program P_0 may fail to terminate or may exhibit outputs other than 'true' or 'false'. In practice, it may be advantageous to begin by developing rather simple programs P_0 which are capable of computing the truth values of certain particular crucial classes of formulas Φ, and then to use these P_0 to verify (2a-2b), (3), and (4a-4b) for more powerful P_0 and for additional auxiliary verification systems. All of this is merely the kind of 'bootstrapping' process typically involved in getting any complex system under way. Of course, in actually building a verification system, one might proceed manually and with a relative minimum of formal checking to implement some level of system function that ought more properly be reached by bootstrapping. If this is done, it is desirable, and may be possible, to use the initial system to verify itself.

Finally, we note that it may be possible, in a particular auxiliary formalism PVF, to find an object γ satisfying $IS\_MPROG(\gamma)$, for which we are also able to establish the universal validity of the formula

(5) $IS\_VERIF(\gamma,PROG,PROP)$ & $YIELDS(\gamma,[X,PROG,PROP],[Y,PROG',PROP']$
$\rightarrow IS\_MPROG(PROG')$ & $IS\_VERIF(Y,PROG',PROP')$.

In this case, the program C can be used as a compiler, i.e., it can be executed with inputs which are verified programs in the extended sense of PVF, to construct verified programs (having related, and perhaps identical, effect) which are directly executable. Note that more and more powerful compilers of this kind will be constructible by a suitable bootstrapping process.

Reference

J. T. Schwartz: *On Correct-Program Technology*, Preceding Paper, this collection.

DATE DUE

1 20	G24		
MAY 12			
GAYLORD			PRINTED IN U.S.A.

N.Y.U. Courant Institute of
Mathematical Sciences
251 Mercer St.
New York, N. Y. 10012