FORMAL SYSTEMS FOR PROGRAM VERIFICATION

by

N. Redding

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

CS-76-24

May 1976

FORMAL SYSTEMS FOR PROGRAM VERIFICATION

by

N. Redding

Computer Science Department
University of Waterloo
Waterloo, Ontario

0. Introduction

The development of formal systems for program verification starts with the representation of the computer by a 1st-order theory, in which the function and predicate symbols denote the machine operations and tests, and the axioms describe the characteristics of the operations and tests in the intended interpretation. The results of Godel on the non-categoricity of theories that contain arithmetic show that a 1st-order theory can never characterize a machine in its intended interpretation: the axioms for the machine will always admit a large number of non-standard models, and hence a proof of a program property from the axioms is not only a proof of the property for the intended machine, but is also a proof for non-standard machines (with enlarged domains) that also satisfy the axioms. The implication of this is that no axiom system can be strong enough to verify all properties of programs that are true; the power of a system for program verification (measured by what properties are verifiable within the system) is dependent on the strength of the axioms. We might ask if the power of verification systems is wholly dependent on the strength of the machine axiomatization, or if some aspects of the "control structures" of the programming language are also uncharacterizable. By control structure, for example the IF-THEN-ELSE and WHILE constructs found

in many programming languages, we mean a feature which specifies the composition of machine operations that defines the function computed by the program. Control structures can be construed to have logical properties that are invariant with the machine interpretation, for example, the statement

IF p(x) THEN f(x) ELSE g(x)

has the property that "if p(x) is true then its value is f(x), and if p(x) is false, then its value is g(x)". This property is clearly independent of the interpretations of p, f, g and x; hence an axiom for processing such a statement is not tied to the axioms for p, f, g and x.

Probably the first important contributions to formal systems for program verification were made by Manna [5] and Hoare [4]. The approach of Manna fails to distinguish in any way between control structure and machine operation; he considers programs in flowchart form, and shows how to construct a formula whose validity is equivalent to the correctness of the given program. The verification takes place entirely within the axiom system for the machine, the control features of the program having been replaced by formula constructs. This loss of distinction precludes an answer to our question. The approach of Hoare does distinguish control structures in that special rules of inference are given for the reduction of program segments containing, for example, IF-THEN-ELSE and WHILE constructs. However, in the absence of a completeness result for Hoare's system, we cannot know if the rules for the control structures are in any sense "adequate", i.e. if the system would become more powerful by the addition of new rules for the structures. We build on the work of

Ashcroft and Wadge [1,2], and are able to answer the question in the negative, i.e. the controls structures found in most languages are characterizable, and the power of verification systems is wholly dependent on the machine axiomatization.

In [1,2] the programming language LUCID is described. LUCID was designed with simple semantics and ease of program verification in mind, and the language maintains a sharp distinction between machine operation and control structure. The language itself has the remarkable property that any LUCID program is not only a recipe (to use the authors' own word!) for computing a function, but also a mathematical assertion in a system of three-valued logic. The program, as an assertion, can be used (without modification) to define itself for the purpose of verification. Our development starts with a study of three-valued logic; the system that we use is somewhat weaker than that in [1,2], but it is still strong enough to encompass the syntax and semantics of LUCID. The advantage of reducing the power (and we are speaking here of the power of languages for logic, not axiom systems within a logic) is that we have semantic completeness, i.e. valid formulae are provable. This is important since verification methods require that formulae be shown to be true in certain classes of models (i.e. "valid" in the models; this is also termed "semantic implication" since the concept is independent of the symbolic aspects of the formulae). A completeness result assures us that symbolic proofs of such implications are always possible. In section 2, we describe a verification method for structured LUCID programs and prove that the method is complete, in that if a program has a stated

property in every model admitted by the axioms, then the proof method will verify the property. The proof method, however, is described in terms of semantic implication, and in section 3 we show that we can apply the results of section 1 to deduce that (i) syntactic proofs of semantic implications within the proof procedure are always possible, (ii) contrary to a conjecture in Ashcroft and Wadge [2], proof by contradiction can be used without sacrificing completeness, and (iii) the power of the proof system is dependent only upon the strength of the machine axiomatization. The result (iii) is obtained in the course of proving (i), and (ii) is an immediate corollary of (i).

The reader is assumed to be familiar with elementary concepts of mathematical logic, as described in [8], and the fixpoint theory of computation, as in [6].

1. Many-Valued Logic

Our goal in this section is to define the system of three-valued logic within which LUCID is defined, and establish the semantic completeness of the system. We begin with a review of many-sorted alphabets and their associated algebras, which we use to define the syntax and semantics of many-valued logic. A many-sorted alphabet is a collection.

$$\Sigma = \{\Sigma_{\langle w,i\rangle} : i \in I,\ w \in I^* \times I\}$$

where I is a set of sorts and each set $\Sigma_{\langle w,i\rangle}$ is a collection of function symbols of type $\langle w,i\rangle$. If $w = s_1 \ldots s_k$ and $f \in \Sigma_{\langle w,i\rangle}$, then f is a function symbol that takes arguments of sorts $s_1,\ldots,s_k$ and produces a result of sort i. Note that if $w = \lambda$, then f is a constant of sort i.

We attach meaning to the symbols in Σ by specifying a Σ-algebra, which is a collection $A = \{A_i : i \in I\}$ of domains and an assignment of maps to the symbols in Σ so that for $f \in \Sigma_{\langle w,i\rangle}$, f_A is a map from $A_{s_1} x \ldots x A_{s_k}$ to A_i for $i \in I$, we define the set of Σ-terms of sort i inductively: each member of $\Sigma_{\langle \lambda,i\rangle}$ is a term of sort i, and if $t_1,\ldots,t_k$ are terms of sorts $s_1,\ldots,s_k$ respectively, and $f \in \Sigma_{\langle w,i\rangle}$ then $f(t_1,\ldots,t_k)$ is a term of sort i. For $i \in I$ we denote the set of Σ-terms of sort i by $(W_\Sigma)_i$. Now let W_Σ be the Σ-algebra where the domain of sort i is $(W_\Sigma)_i$ and the value $f_{W_\Sigma}(t_1,\ldots,t_k)$ is the term $f(t_1,\ldots,t_k)$. W_Σ has the property that for any Σ-algebra A, there exists a unique homomorphism h_A from W_Σ to A. A homomorphism from a Σ-algebra B to a Σ-algebra C is a family of maps $\{h_i : i \in I\}$ where $h_i : B_i \to C_i$, and for $f \in \Sigma_{\langle w,i\rangle}$ and $b_j \in B_{s_j}$ we have

$$f_C(h_{s_1}(b_1),\ldots,h_{s_k}(b_k)) = h_i(f_B(b_1,\ldots,b_k)).$$

The unique homomorphism h_A from W_Σ to A is given by $(h_A)_i(c) = c_A$ for $c \in \Sigma_{\langle \lambda,i\rangle}$, and

$$(h_A)_i(f(t_1,\ldots,t_k)) = f_A((h_A)_{s_1}(t_1),\ldots,(h_A)_{s_k}(t_k))$$

for $f \in \Sigma_{\langle w,i\rangle}$. The verification that h_A is a homomorphism (and unique) is an easy exercise. Now let X_i be a denumerable set of variables of sort $i \in I$. $\Sigma(X)$ denotes the I-sorted alphabet obtained from Σ by adding the set X_i to $\Sigma_{\langle \lambda,i\rangle}$ for each $i \in I$. Given a Σ-algebra A, and a family of maps

$$\{u_i : X_i \to A_i : i \in I\}$$

let A[u] denote the $\Sigma(X)$-algebra obtained by extending A so that for

$x \in X_i$, $x_{A[u]} = u_i(x)$. Now the homomorphism $h_{A[u]}$ assigns values to $\Sigma(X)$-terms under the interpretation u of the variables.

Example Let I = {0,1}. The sort 0 is for "truthvalue" and the sort 1 is for "integer". Let $\Sigma_{\langle\lambda,0\rangle} = \{T,F\}$, $\Sigma_{\langle\lambda,1\rangle} = \{0,1,2,\ldots\}$, $\Sigma_{\langle 11,1\rangle} = \{+,-,*,/\}$ and $\Sigma_{\langle 11,0\rangle} = \{=\}$. Let $\Sigma_{\langle w,i\rangle}$ be empty in all other cases. Now Σ is an alphabet for simple arithmetic; $(W_\Sigma)_0$ is the set of equations of the form "=(s,t)" and $(W_\Sigma)_1$ is the set of arithmetic terms from which s and t would be chosen. Let A be the Σ-algebra where A_0 = {true,false} and A_1 is the set of natural numbers. Let the symbols +,-,*,/ and = be interpreted as usual. Now for $\alpha \in (W_\Sigma)_0$, we have $(h_A)_0(\alpha)$ = true if and only if α is a true arithmetic identity.

We are now ready to define our system of logic. A 1st-order alphabet is an n-sorted alphabet Σ (n = {0,...,n-1}) with distinguished symbols $T,F \in \Sigma_{\langle\lambda,0\rangle}$, $\vee \in \Sigma_{\langle 00,0\rangle}$, $\neg \in \Sigma_{\langle 0,0\rangle}$ and $=_i \in \Sigma_{\langle ii,0\rangle}$ for each $i < n$. We will omit the type-specifying subscript on the = symbol since it is clear from the context in which it is used. Sort 0 is distinguished as the sort for "truthvalue", and the remaining sorts may be chosen at will. Note that the symbols $\neg$ and $\vee$ are the familiar negation and disjunction connectives, and = is the equality predicate. We will always use the latter two symbols in any infix way, e.g. we write s = t rather than =(s,t). We choose many-sorted alphabets to form the base of our machine representation because it seems natural: many programming languages distinguish data sorts, and incorporate the sorts into the syntax. In addition, many-sorted alphabets allow us to treat the sort 0 just as any other sort, and hence operations on truthvalues can be included in a

uniform way. Now, a 1st-order alphabet Σ defines the basic machine operations and tests, and a Σ-algebra defines an interpretation of the machine. In order to make statements in the langage, we need variables and quantifiers. To this end, let X_i be a denumerable set of variables for each $i < n$. Let Σ^+ come from $\Sigma(X)$ by adding the quantifiers "$\forall x$" to $\Sigma(X)_{\langle 0,0\rangle}$ for each $x \in X_i$ and $i < n$. The set of 1st-order terms of sort i is $(W_{\Sigma^+})_i$, and the set of 1st-order formulae is $(W_{\Sigma^+})_0$. An occurrence of a variable x that is below a quantifier $\forall x$ is said to be bound. An occurrence of x is free if it is not bound. For example, in the formula

$$\forall x(P(x,y) \rightarrow \forall y(P(y,z))$$

the single occurrence of x is bound, the single occurrence of z is free, and y has one free occurrence and one bound occurrence. We use lower case roman letters for terms (of any sort), and lower case greek letters for formulae. Constructs of the form $t[x]$ (or $\alpha[x]$) denote an occurrence of the variable x in t (or α), and $t[s]$ (respectively $\alpha[s]$) denotes the result of replacing all free occurrences of x by the term s. Note that, syntactically, the symbol "$\forall x$" is a function. We cannot interpret "$\forall x$" as a function, however, since its purpose is to bind the variable x in the terms that occur below it. To circumvent this problem, we will define a Σ-interpretation for Σ^+ as an extension of a Σ-algebra; the former will not, in general, be a Σ^+-algebra. We will consider Σ-algebras A that satisfy the following minimal conditions:

(i) The elements T_A and F_A of A_0 are distinct.

(ii) For $b \in A_0$, $\neg_A(b) = T_A$ iff $b = F_A$, $\neg_A(b) = F_A$ iff $b = T_A$, and $\neg_A(\neg_A(b)) = b$.

(iii) For $b,b' \in A_0$, $b \vee_A b' = T_A$ iff $T_A \in \{b,b'\}$ and $b \vee_A b' = F_A$ iff $b = b' = F_A$.

(iv) For $a,a' \in A_i$, $(a =_A a') = T_A$ iff $a = a'$ and $(a =_A a') = F_A$ iff $a \neq a'$.

Note that we do not require $A_0 = \{T_A,F_A\}$. Below, we consider only the case where A_0 has three elements, but this constraint is not necessary just yet. We now define a Σ-interpretation to be a pair (A,Q) where A is a Σ-algebra satisfying (i)-(iv), and Q is a map from the set of subsets of A_0 to A_0 such that for any subset B of A_0, $Q(B) = T_A$ iff $B = \{T_A\}$ and $Q(B) = F_A$ iff $F_A \in B$. Given an assignment $u:X \to A$ of the variables, we define the <u>evaluation map</u>

$$A[Q,u]:W_{\Sigma^+} \to A$$

inductively: if t is a Σ^+-term that contains no quantifiers, then $A[Q,u](t) = h_{A[u]}(t)$. If the term t is the formula $\forall x(s)$ then we define

$$A[Q,u](\forall x(s)) = Q(\{A[Q,u(x/a)](s):a \in A_i\})$$

where $x \in X_i$. Hence the value of x is allowed to range over A_i, and the resultant set of truthvalues is processed by Q. Note that $u(x/a)$ is the assignment that agrees with u at every point except that $u(x) = a$. Given a Σ-interpretation A (we discard the notation (A,Q) and consider Q to be included in A), an assignment u to the variables, and a formula α, we write $A \models_u \alpha$ if $A[Q,u](\alpha) = T$. We write $A \models \alpha$ if $A \models_u \alpha$ for every assignment u to the variables. If Δ is a set of formulae, then we write $\Delta \models \alpha$ (read "Δ semantically implies α") if whenever $A \models \Delta$ (i.e. $A \models \delta$ for all $\delta \in \Delta$) we have $A \models \alpha$.

Example Retaining the notation of the last example, we have

$$A \models \forall x(x \neq 0 \rightarrow x*x \neq 0).$$

The construct "$\alpha \rightarrow \beta$" abbreviates "$\alpha \neq T \vee \beta$" and we have abbreviated "$\neg(s=t)$" by "$s \neq t$".

We now consider syntactic operations on systems of logic. A derivation system consists of a set of axioms, which are formulae to be used as initial assumptions in proofs, and rules of inference which allow certain formulae to be derived from other formulae. A formal proof is a finite sequence $\alpha_1,\ldots,\alpha_n$ of formulae such that each α_i is either an axiom, or follows from previous formulae in the sequence by a rule of inference. We are free to specify axioms for the machine, which are used when proving properties of programs, subject to the constraint that the axioms be true in the intended interpretation of the machine. We include in all axiom systems the following formulae, which are valid in every Σ-interpretation.

L1. $\alpha \rightarrow (\beta \rightarrow \alpha)$

L2. $(\alpha \rightarrow (\beta \rightarrow \gamma)) \rightarrow ((\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma))$

L3. $(\alpha \neq T \rightarrow \beta \neq T) \rightarrow ((\alpha \neq T \rightarrow \beta) \rightarrow \alpha)$

E1. $t = t$

E2. $s = t \rightarrow (\alpha[s] \rightarrow \alpha[t])$ where no free variable of s or t is quantified in α.

E3. $s = t \vee s \neq t$

T1. $T \neq F$

T2. $\alpha \rightarrow (\alpha = T)$

T3. $(\alpha = T) \rightarrow \alpha$

N1. $\neg\alpha \rightarrow (\alpha = F)$

N2. $((\neg\alpha) = F) \rightarrow \alpha$

N3. $\neg(\neg\alpha) = \alpha$

D1. $\alpha \veebar \beta \rightarrow (\alpha = T \vee \beta = T)$

D2. $(\alpha \vee \beta) = (\beta \vee \alpha)$

D3. $T \vee \alpha$

D4. $\neg(\alpha \vee \beta) \rightarrow \neg\alpha$

D5. $\neg\alpha \rightarrow (\neg\beta \rightarrow \neg(\alpha \vee \beta))$

G1. $\forall x(\alpha[x]) \rightarrow \alpha[t]$ where no free variable of t is quantified in α.

G2. $\forall x(\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \forall x(\beta))$ where α contains no free occurrences of x.

G3. $\forall x(\alpha) \leftrightarrow \forall x(\alpha = T)$

G4. $\neg(\forall x(\alpha)) \leftrightarrow \neg(\forall x(\alpha \neq F))$

The axioms L1-L4 define the properties of implication ($\rightarrow$), E1-E3 the properties of equality (=), T1-T3 the properties of truth (this is needed because our system is not two-valued), N1-N3 the properties of negation ($\neg$), D1-D5 the properties of disjunction ($\vee$) and G1-G4 the properties of quantification. We admit two fundamental rules of inference, namely, Modus Ponens (from $\alpha \rightarrow \beta$ and α, infer β) and Generalization (from α infer $\forall x(\alpha)$). We can add other rules of inference, provided that if a rule infers α from a set of formulae Δ, then $\Delta \models \alpha$. It is easily seen that Moduls Ponens and Generalization have this property.

Having defined the syntax and semantics of 1st-order languages, we define a Many-Valued Logic (abbreviated "MVL") to be a triple $(\Sigma,\{A\},D)$ where Σ is a 1st-order alphabet, $\{A\}$ is a collection of Σ-interpretations such that for all $A,A' \in \{A\}$, $A_0 = A'_0$ and $Q = Q'$, and D is

a derivation system containing the basic axioms given above, the rules of Modus Ponens and Generalization, and any other axioms and rules of inference that are admissible. Admissible rules are defined above, and an axiom α is admissible if $\{A\} \models \alpha$, i.e. $A \models \alpha$ for all $A \in \{A\}$. We write $D \vdash \alpha$ if α is derivable in D, and it is clear that if $D \vdash \alpha$ then $\{A\} \models \alpha$. The reader should observe that the power of the language Σ is determined by the breadth of $\{A\}$. For example, if Σ is the alphabet for arithmetic, and $\{A\}$ consists solely of the standard structure for arithmetic, then the language is as powerful as it can possibly be. Godel's Theorem shows that such an MVL cannot be complete no matter how extensive the axiom system. At the other extreme, $\{A\}$ could consist of all Σ-interpretations that satisfy all of the axioms in D. In this case, it would be complete. However, $\{A\}$ does not have to be quite as wide as this for completeness; we will restrict ourselves MVL's that are <u>full</u>. An MVL $(\Sigma,\{A\},D)$ is full if $\{A\}$ is the set (up to isomorphism) of all countable models of the axioms in D. Assuming that D is consistent, $\{A\}$ is non-empty.

We now consider MVL that are <u>three-valued</u>, and prove that every full, three-valued MVL is complete (i.e. $\{A\} \models \alpha$ implies $D \vdash \alpha$). An MVL $(\Sigma,\{A\},D)$ is three-valued if there is a constant $\perp \in \Sigma_{\langle\lambda,0\rangle}$ such that

$$D \vdash \forall x(x=T \vee x=F \vee x=\perp)$$

where x is a variable of sort 0. If this is the case, then we must have $A_0 = \{T_A, F_A, \perp_A\}$ for all $A \in \{A\}$. Note also that the quantifier function Q is completely determined for three-valued MVL; the only definition to Q that satisfies the conditions is

$$Q(\{T\}) = T$$

$$Q(\{F\}) = Q(\{T,F\}) = Q(\{F,\perp\}) = Q(\{T,F,\perp\}) = F$$

$$Q(\{\perp\}) = Q(\{T,\perp\}) = \perp$$

Three-valued MVL have an important normal form property; the observant reader will have noticed that we allow quantifiers to occur below non-logical symbols in formulae, whereas this is not permitted in standard predicate logic. (A non-logical symbol is a symbol other than $\neg$, $\vee$, $\forall x$ and $=_0$.) This flexibility does not increase the power of the logic, as we will show. A formula α is said to be normal if no quantifier occurs below a non-logical function symbol. The following result is required when we prove the Model Existence Theorem for MVL; it assists us in using structural induction on formulae.

Proposition Let $(\Sigma,\{A\},D)$ be a three-valued MVL. For every formula $\alpha[x_1,\ldots,x_k]$ with free variables in $x_1,\ldots,x_k$ there exists a formula $\alpha'[x_1,\ldots,x_k]$ with the same free variables as α such that α' is normal, and $D \models \alpha = \alpha'$.

Remark The derivability of $\alpha = \alpha'$ implies that α and α' take the same value under every Σ-interpretation and assignment to variables.

Proof (Sketch) Let us call an occurrence of a quantifier in α "bad" if it occurs below a non-logical symbol. We use induction on the number of bad quantifiers in α. If there are none, then α is already normal. Now assume that the Proposition is true for formulae with no more than m-1 bad quantifiers, and suppose that α has m bad quantifiers. Now α must have a topmost bad quantifier, i.e. there must be a bad quantifier that does not occur below any other bad quantifiers. Hence α can be written as

$$\beta[\gamma[\forall x(\delta)]]$$

where the new variable z, in $\beta[z]$, occurs below logical symbols only, the topmost symbol of γ is non-logical of type $\langle w,0\rangle$ for some w, and the new variable y does not occur below quantifiers in $\gamma[y]$. This can be verified formally by structural induction on α. It should be obvious that none of $\beta[z]$, $\gamma[y]$ and δ has more than m-1 bad quantifiers. Hence we can apply the induction hypothesis to find normal forms for $\beta[z]$, $\gamma[y]$ and δ, say $\beta'[z]$, $\gamma'[y]$ and δ' respectively. Now consider the formula α', defined as

$$\beta'[\sigma_T \ \& \ \sigma_F \ \& \ \sigma_\perp]$$

where for $k \in \{T,F,\perp\}$, σ_k is the formula

$$\forall x(\delta') = k \rightarrow \gamma'[k]$$

α' is clearly in normal form, and in fact $D \vdash \alpha = \alpha'$. To prove this, it suffices to show that

$$D \vdash \gamma'[\forall x(\delta')] = (\sigma_T \ \& \ \sigma_F \ \& \ \sigma_\perp)$$

This can be proved by substitution on the (propositional) theorem

$$D \vdash \xi[x] = ((x=T \rightarrow \xi[T]) \ \& \ (x=F \rightarrow \xi[F]) \ \& \ (x=\perp \rightarrow \xi[\perp]))$$

where $\xi[x]$ is an arbitrary formula. This, of course, depends on the fact that D is three-valued, and the above formula can be proved without the use of quantifiers. We omit the details.

We now turn to the Model Existence Theorem and the Completeness Theorem for three-valued MVL. Our method is that of Boolean Representation, introduced by Rasiowa and Sikorski [9] and used by Bell and Slomson [3] for the two-valued calculus. We want to prove

that in a consistent MVL, every consistent formula has a model.

The idea of the Boolean Representation Method is to construct a Boolean Algebra of formulae from a MVL, show that certain ultrafilters in the Boolean Algebra induce models of themselves (i.e. of the formulae within), and show that any consistent formula is contained in such an ultrafilter. So let $(\Sigma,\{A\},D)$ be a three-valued, full MVL. Define a binary relation E on $(W_{\Sigma^+})_0$ by

$$E(\alpha,\beta) \text{ iff } D \vdash \alpha \leftrightarrow \beta.$$

It is easily verified that E is an equivalence relation; let the equivalence class of a formula α be $|\alpha|$, and let

$$L_\Sigma = \{|\alpha| : \alpha \in (W_{\Sigma^+})_0\}$$

We define $|\alpha| \leq |\beta|$ if $D \vdash \alpha \rightarrow \beta$, and it is easy to show that $(L_\Sigma,\leq)$ is a Boolean Algebra where

$$\inf(L_\Sigma) = |F|$$

$$\sup(L_\Sigma) = |T|$$

$$\text{meet}(|\alpha|,|\beta|) = |\alpha \,\&\, \beta|$$

$$\text{join}(|\alpha|,|\beta|) = |\alpha \vee \beta|$$

$$\text{compl}(|\alpha|) = |\alpha \neq T|$$

The ultrafilters in L_Σ that we consider are those that are Q-preserving. An ultrafilter U in L_Σ is Q-preserving if for every formula $\alpha[x]$, if $|\alpha[t]| \in U$ for all terms t, then $|\forall x(\alpha[x])| \in U$. It can be shown that for any formula $\alpha[x]$,

$$|\forall x(\alpha[x])| = \inf(\{|\alpha[t]| : \text{any term } t\})$$

and so the following result is a direct consequence of a theorem of Tarski [10].

Ultrafilter Theorem for L_Σ Every non-zero element (i.e. element distinct from $|F|$) of L_Σ is contained in a Q-preserving ultrafilter.

Before proceeding with the model existence theorem, we return to many-sorted algebras. Given a Σ-algebra A, we say that an equivalence relation C on the domains of A is a Σ-congruence if for each function symbol f, and members (of the appropriate sorts) $a_1,\ldots,a_n$ and $b_1,\ldots,b_n$ of the domains, if $C(a_i,b_i)$ holds for $i = 1,\ldots,n$ then

$$C(f_A(a_1,\ldots,a_n),\ f_A(b_1,\ldots,b_n))$$

holds. We define the quotient algebra A/C to have the equivalence classes of the domains of A as its domains, and to interpret the function symbols by representative, i.e. if the equivalence class of an element b is denoted b/c then

$$f_{A/C}(b_1/C,\ldots,b_n/C) = f_A(b_1,\ldots,b_n)/C$$

It is easily verified that A/C is a Σ-algebra; this type of quotient construction is very common in algebra, e.g. quotient groups, rings and fields.

Model Existence Theorem: Let $(\Sigma,\{A\},D)$ be a full, three-valued MVL. For each Q-preserving ultrafilter U in L_Σ, there exists $A \in \{A\}$ such that for every closed formula α, $|\alpha| \in U$ if and only if $A \models \alpha$.

Remark A formula is closed if it does not contain free variables. The universal closure of a formula α is

$$\forall x_1(\ldots(\forall x_n(\alpha))\ldots)$$

where $x_1,\ldots,x_n$ are all the free variables of α. From the axioms G1-G4, it is clear that α is consistent iff the universal closure of α is consistent. Hence, in conjunction with the Ultrafilter Theorem, this result shows that every consistent formula has a model.

Proof (Sketch) Let U be given, and define a binary relation C on W_Σ by C(s,t) iff $|s=t| \in U$. Recall that W_Σ is the set of Σ-terms without variables or quantifiers, and that W_Σ is a Σ-algebra. Now by virtue of the axioms E1-E3 for equality, C is a Σ-congruence. Let A be the Σ-algebra W_Σ/C. We claim that A is a Σ-interpretation, and has the required property that $|\alpha| \in U$ iff $A \models \alpha$ for all closed formulae α. To show this, we show that for every formula $\alpha[x_1,\ldots,x_n]$ in normal form, with free variables $x_1,\ldots,x_n$, and for any selection of terms $t_1,\ldots,t_n$ from W_Σ,

(*) $A \models \alpha[t_1,\ldots,t_n]$ if and only if $|\alpha[t_1,\ldots,t_n]| \in U$

If, in addition to this, we show that A satisfies the conditions for a Σ-interpretation, then we can conclude

(i) $A \models \alpha$ iff $|\alpha| \in U$ for all closed, normal α.

(ii) $A \models \alpha$ iff $|\alpha| \in U$ for all closed α, since there exists a closed, normal α' such that $D \vdash \alpha = \alpha'$

(iii) $A \in \{A\}$ since for each axiom α in D, $D \vdash \alpha$ and hence $|\alpha| = |T|$, i.e. $|\alpha| \in U$. Note that we are using the hypothesis that {A} is full (i.e. contains every countable model of D) and we are assuming that Σ is countable (and hence W_Σ/C is countable.)

First, the fact that A qualifies as a Σ-interpretation follows from the hypothesis that D is consistent; the latter tells us that the equivalence classes of T and F in W_Σ/C are distinct (lest $D \vdash T=F$ which contradicts axiom T1). The remaining conditions for a Σ-interpretation then follow immediately from the definition of C and the axioms. To prove (*), we note that $A \models \alpha$ iff $A \models \alpha=T$ and $|\alpha| \in U$ iff $|\alpha=T| \in U$. Hence, it suffices to prove (*) where we assume α to have the form $\beta[x_1,\ldots,x_n] = k$, where k is T,F or $\perp$. The proof of this is a straightforward structural induction on β. Note that since β is in normal form, the induction is on the number of quantifiers and logical connectives. We start with formulae with no quantifiers or connectives, and then proceed with the cases $\gamma \vee \delta$, $\neg(\gamma)$, $\gamma =_0 \delta$ and $\forall x(\gamma)$ where the result is assumed true for γ and δ. We do not give the full proof here, but illustrate the reduction for two of the cases:

If β is $\gamma \vee \delta$ and k is T, then we assume, inductively, that (*) holds for γ and δ. Now by axiom D1 we have

$$
\begin{aligned}
& A \models \beta[t_1,\ldots,t_n] = T \\
\text{iff} \quad & A \models \gamma[t_1,\ldots,t_n] = T \vee \delta[t_1,\ldots,t_n] = T \\
\text{iff} \quad & A \models \gamma[t_1,\ldots,t_n] \text{ or } A \models \delta[t_1,\ldots,t_n] \\
\text{iff} \quad & |\gamma[t_1,\ldots,t_n]| \in U \text{ or } |\delta[t_1,\ldots,t_n]| \in U \\
\text{iff} \quad & |\beta[t_1,\ldots,t_n]| \in U.
\end{aligned}
$$

If β is $\neg(\gamma)$ then

$$
\begin{aligned}
& A \models \beta[t_1,\ldots,t_n] = T \\
\text{iff} \quad & A \models \gamma[t_1,\ldots,t_n] = F \\
\text{iff} \quad & |\gamma[t_1,\ldots,t_n] = F| \in U \\
\text{iff} \quad & |\neg(\gamma[t_1,\ldots,t_n])| \in U
\end{aligned}
$$

Completeness Theorem Let $(\Sigma,\{A\},D)$ be a full, three-valued MVL. Then for every formula α, if $\{A\} \models \alpha$ then $D \vdash \alpha$.

Proof Assume that $\{A\} \vdash \alpha$, but α is not derivable in D. Then we do not have $D \vdash \alpha \neq T \to F$ (otherwise $D \vdash T \to \alpha$, i.e. $D \vdash \alpha$); so $|\alpha \neq T|$ is non-zero in L_Σ, and by the Model Existence Theorem, $\alpha \neq T$ has a model in $\{A\}$. This contradicts $\{A\} \models \alpha$, and so we must have $D \vdash \alpha$ as required.

We end this section with a summary of results on partially ordered sets. A partial order $\leq$ on a set X is a complete partial order (abbreviated "c.p.o.") if there exists in X a unique minimal element under $\leq$, and if every ascending chain

$$x_0 \leq x_1 \leq \ldots$$

in X has a least upper bound. We usually identify a set with its order, and hence "X is a c.p.o." means that the order on X is a c.p.o. The least element of a c.p.o. is denoted $\perp$, which coincides with the constant introduced with three-valued MVL and is explained below. A c.p.o. X is "flat" if for any two elements $x,y \in X$, $x \leq y$ iff $x = y$ or $x = \perp$. In our application, $\perp$ denotes an "undefined" element and the order $\leq$ expresses approximation. A flat c.p.o. is a set in which $\perp$ approximates everything, and defined (i.e. non-minimal) elements approximate themselves, but nothing else. Now, given c.p.o.'s X and Y, a map $f:X \to Y$ is continuous if it is monotonic (i.e. $x \leq x'$ implies $f(x) \leq f(x')$) and for each ascending sequence

$$x_0 \leq x_1 \leq \ldots$$

in X, we have

$$f(\sqcup\{x_i: i = 0,1,...\}) = \sqcup\{f(x_i): i = 0,1,...\}$$

The symbol $\sqcup$ denotes the least upper bound operation; hence f is continuous if it preserves least upper bounds. All computable functions will be continuous, although the converse of this is not true. The following result is central to fixpoint semantics.

Fixpoint Theorem Let X be a c.p.o. and let $f:X \to X$ be a continuous map. Then f has a least fixpoint, i.e. there exists $e \in X$ such that $f(e) = e$ and whenever $f(e') = e'$ for some other $e' \in X$, $e \leq e'$. In fact,

$$e = \sqcup\{e_i: i = 0,1,...\}$$

where $e_0 = \perp$ and $e_{i+1} = f(e_i)$.

Given a c.p.o. X and an arbitrary set S, $[S \to X]$ denotes the set of all maps from S to X. $[S \to X]$ can be ordered by

$$f \leq g \text{ iff } f(s) \leq g(s) \text{ for all } s \in S$$

and in fact $[S \to X]$ is a c.p.o. under this order. The order $\leq$ on $[S \to X]$ is the point-wise extension of the order $\leq$ on X. Given a continuous map $f:X \to Y$ where X and Y are c.p.o.'s, the point-wise extension f_S of f to $[S \to X]$ and $[S \to Y]$ is defined by

$$f_S(g,h)(s) = f(g(s),h(s))$$

It is easily verified that f_S is continuous with respect to the order $\leq$ on $[S \to X]$ and $[S \to Y]$. In the next section, we define the semantics of LUCID as fixpoints of continuous maps on c.p.o.'s of the form $[\omega \to A]$ where A is the basic data domain.

A three-valued MVL $(\Sigma,\{A\},D)$ is continuous if for each $A \in \{A\}$, each domain in A is a flat c.p.o. and for each function symbol f of Σ,

except the equality symbols, f_A is continuous. In the case of the domains of sort 0, we assume that the constant $\perp$ already introduced is the minimal element. The basic logical functions $\neg$ and $\vee$ are then continuous. The equality relation can never be continuous, but it does not matter since this form of equality is not computable. Later on, we introduce a computable equality predicate which takes the value $\perp$ if either of its arguments is $\perp$ (recall that = always returns T or F). Finally, we note that axioms of continuity of function symbols in Σ can be included in D to ensure that $(\Sigma,\{A\},D)$ is continuous. It is easy to show that in a flat c.p.o., a function is continuous iff it is monotonic. Hence, D should contain the axiom

$$\forall x_1 \ldots \forall x_n \; \forall y_1 \ldots \forall y_n (x_1 \leq y_1 \; \& \; \ldots \; x_n \leq y_n \rightarrow f(x_1,\ldots,x_n) \leq f(y_1,\ldots,y_n))$$

for each function symbol f, and the definition

$$(x \leq y) = (x=\perp \vee \; x=y).$$

2. The Semantics of Lucid

In this section, we define the programming language LUCID and describe a method for representing and proving assertions about LUCID programs. The reader is referred to Ashcroft and Wadge [1,2] for a complete description of LUCID with detailed examples and discussion; we give only the definitions and some illustrative examples.

LUCID is based on the idea that the value of a variable within a program is not just a single member of some data domain, but a history of the values that the variable has taken during the execution of the program. Time is considered to pass in discrete units, and so the domain of a variable is a set of the form $[\omega \rightarrow A]$ - the set of all maps from the natural numbers to some data domain A. If the value of the variable x is h, then at time $t \in \omega$, the value of x is h(t). For any particular input to a program, each variable in the program takes exactly one value in the set $[\omega \rightarrow A]$, and hence the semantics of LUCID tells us how the value of each variable is assigned. The result of a program's "execution" is the value of the output variable at some time. The time at which the output variable represents the result of the program is, in general, determined by the values of other variables at other times. Let us take, as an example, a LUCID program that finds the largest square number not exceeding its input:

```
first X = 0
 next X = X+1
square  = X*X
 output = square as soon as (next square) > input
```

In this program, the symbols "input" and "output" are respectively the input and output variables, and the symbols "X" and "square" are auxiliary variables. The first two statements assert that the initial value (at time t=0) of X is 0, and that at time t+1 the value of X is X(t)+1. Hence X is the sequence <0,1,2,3,...>. The third statement asserts that the value of square at any time t is X(t)*X(t); hence square is the sequence <0,1,4,9,...>. The final statement asserts that the value of the output variable is the value of square at such time t as square (t+1) > input(t+1). We consider the value of the input variable to be constant through time, i.e. input is a sequence <n,n,...>. Furthermore, the result of an as soon as expression is constant through time; hence the value of output is constantly square(t) where t is the least number such that square(t+1) > input, i.e. output is the largest square number not exceeding input. Observe that each statement of the above program is not only an executable command (in that it can be used to generate a computation sequence) but also an assertion; the value of a variable (as a sequence) is fixed for any given input, and the statements of the program are equations that must hold for an assignment of values to the variables to be correct. At this point, it becomes apparent that some means for handling non-terminating computations must be included. For example, if the definition of X is as above, then the statement

output = X+10 as soon as X < 0

is non-terminating because there does not exist a time t at which X(t) < 0. In this case, the value of output would be "undefined", and

the most elegant and usable way to formalize this is with fixpoint semantics. With fixpoint semantics, a particular element of the data domain is singled out as the element to represent "undefined"; as a result, all functions may be considered to be total, thus simplifying the equational approach outlined above. Hence, in the case where the conditional part of an as soon as expression is never true, the expression takes the value undefined.

We now give the full syntax of LUCID; the main constituent of a LUCID language is a continuous, three-valued MVL $(\Sigma,\{A\},D)$. The machine operations and tests are given by the alphabet Σ; the interpretations in $\{A\}$ determine the semantics of LUCID programs over Σ, but note that the breadth of $\{A\}$ does not affect the syntax or semantics since interpretation is with respect to a single member of $\{A\}$. The derivation system D is the basis of the proof system; we will discuss this in section 3. In order that expressions can be formed which take ω-indexed sequences as values, we duplicate the alphabet Σ so that each new symbol is the pointwise extension to ω of the symbol that it duplicates. More precisely, (assuming that Σ is n-sorted) let Σ' be the 2n-sorted alphabet defined by

$$\Sigma'_{\langle s_1 \ldots s_k, i\rangle} = \Sigma_{\langle s_1 \ldots s_k, i\rangle}$$

$$\Sigma'_{\langle (s_1+n)\ldots(s_k+n), i+n\rangle} = \{f' : f \in \Sigma_{\langle s_1 \ldots s_k, i\rangle}\}$$

where $s_1,\ldots,s_k$ and i range from 0 to n-1 and $k \in \omega$. The sorts i+n are the sequences of sort i; thus if $c \in \Sigma_{\langle\lambda,i\rangle}$ then c' is a constant of sort i+n, and c'_A will be interpreted as $\langle c_A, c_A, \ldots\rangle$ where c_A is the

interpretation of c. If $f \in \Sigma_{\langle s_1 \ldots s_k, i\rangle}$ then f' is a function that takes sequences of sorts $s_1,\ldots,s_k$ as arguments and produces a sequence as a result; more precisely, if $Z_1,\ldots,Z_k$ are sequences of elements of sorts $s_1,\ldots,s_k$ respectively, and if f_A is an interpretation of f, then the value of $f'_A(Z_1,\ldots,Z_k)$ is a sequence where

$$f'_A(Z_1,\ldots,Z_k)(t) = f_A(Z_1(t),\ldots,Z_k(t))$$

i.e. f' is the point-wise extension of f. We now obtain Σ^L from Σ' by adding the symbols

$\underline{\text{first}}_i$ and $\underline{\text{next}}_i$, each of type $\langle i,i\rangle$

$\underline{\text{as soon as}}_i$, of type $\langle in,n\rangle$

$\underline{\text{followed by}}_i$ of type $\langle ii,i\rangle$

E_i of type $\langle i,i\text{-}n\rangle$

$=_i$ of type $\langle ii,0\rangle$

where i ranges from n to 2n-1. We will omit the type-specifying subscripts, since the types will be clear from the context. The operator first produces a constant sequence, each member of which is the initial member of the argument. Hence first $\langle a_0,a_1,\ldots\rangle = \langle a_0,a_0,\ldots\rangle$. The operator next truncates the initial member of its argument; thus next $\langle a_0,a_1,\ldots\rangle = \langle a_1,a_2,\ldots\rangle$. The binary operators as soon as and followed by will be written as infix; the operator followed by has the property that for any sequences X and Y,

$$\underline{\text{first}}\ (X\ \underline{\text{followed by}}\ Y) = \underline{\text{first}}\ X$$

and $$\underline{\text{next}}\ (X\ \underline{\text{followed by}}\ Y) = Y.$$

Hence the followed-by operator concatenates the initial value of X to the sequence Y; this is used to eliminate definitions of the form "first X = s; next X = t", replacing them by "X = s followed by t". The operator E is used to extract the initial value of a sequence, not as a constant sequence, but as an individual element. Thus $E(\langle a_0,a_1,\ldots\rangle) = a_0$. This operator is needed for technical reasons; we will dispense with it in the next section. Finally, we have added new equality predicates for each sort from n to 2n-1. The reason for this is that when Σ' is formed, the equality predicates in Σ (and, note, the logical connectives $\vee$ and $\neg$) are duplicated, but their interpretation is point-wise. For example, if =' is point-wise equality, then the value of

$$\langle 0,1,2,3,4,5,\ldots\rangle =' \langle 0,0,2,0,4,0,\ldots\rangle$$

is the sequence $\langle T,F,T,F,T,F,\ldots\rangle$ whereas strict equality returns "F" since the sequences are distinct. We need a strict equality predicate for each sort if the alphabet is to qualify as a first-order alphabet. We also need the strict equality for technical reasons. To distinguish the point-wise and strict equality, the former will be written ='.

We can form terms from Σ^L just as from any other many-sorted alphabet. A LUCID-term is a member t of $(W_{\Sigma^L(X)})_i$ where i > n-1 and t contains no occurrences of = or =' . Note that t cannot contain quantifiers, since we are considering $\Sigma^L(X)$ rather than $(\Sigma^L)^+$. Note also that function symbols from Σ, and the operator E are precluded. Only the duplicate symbols denoting point-wise extensions, and the LUCID functions

are permitted. We do not allow = or =' because they are not continuous; comparisons for equality are made via a computable form of equality, which is included in the basic alphabet Σ. Now, a Lucid-module is an unordered set P of assertions, each of the form

first X = s
next X = t
or X = t

where s and t are LUCID-terms and X is a variable name. The following constraints are placed on the assertions:

(i) Each variable is defined exactly once by one of the two assertion types. This is with the exception of a distinguished set of input variables which are not defined, but may be used in terms just as constants.

(ii) If a definition of the form first X = s occurs in P, then the term s must be quiescent. A quiescent term is one which always evaluates to a constant sequence; we define the set of quiescent terms (which depends, in part, on P) inductively: a constant of Σ' of sort > n-1 is a quiescent term, as is an input variable; if $t_1,\ldots,t_k$ are quiescent terms and f is a function symbol of Σ, then $f'(t_1,\ldots,t_k)$ is a quiescent term; any term of the form (first t) or (s as soon as t) is a quiescent term; finally, if the definition Y = t is in P and t is a quiescent term, then Y counts as a quiescent term.

(iii) There is a distinguished output variable O which is defined by an assertion O = s where s is quiescent.

Let P be a Lucid module; we will write $P[I_1,\ldots,I_k]$ to distinguish the input variables $I_1,\ldots,I_k$. Given an interpretation $A \in \{A\}$ and input values $a_1,\ldots,a_k$ (note that $I_1,\ldots,I_k$ have sorts, and the sort of an input value for I_j must be s_j-n where I_j has sort s_j) we define a system of equations over c.p.o.'s such that the least fixpoint of the system (viewed as a continuous function) provides the values of the variables in P. The c.p.o.'s that we use will be the point-wise extensions of the c.p.o.'s in A. We have already described informally the interpretation of the new symbols in Σ^L; we now give a formal definition. For $A \in \{A\}$ let A' be the Σ^L-algebra defined as follows:

1. For $i < n$, $A'_i = A_i$
2. For $i > n-1$, $A'_i = [\omega \to A_{i-n}]$, the set of all maps from ω to A_{i-n}
3. For $f \in \Sigma$, $f'_A = f_A$
4. For $f \in \Sigma$, $f'_{A'}$ is the point-wise extension of f_A to $[\omega \to A_{i-n}]$, and $=_{A'}$ is true equality on $[\omega \to A_i]$
5. For sequences $\langle a_0,a_1,\ldots\rangle$ and $\langle b_0,b_1,\ldots\rangle$,
   $\underline{\text{first}}_{A'}\langle a_0,a_1,a_2,\ldots\rangle = \langle a_0,a_0,a_0,\ldots\rangle$
   $\underline{\text{next}}_{A'}\langle a_0,a_1,a_2,\ldots\rangle = \langle a_1,a_2,\ldots\rangle$
   $\langle a_0,a_1,\ldots\rangle \ \underline{\text{followed by}}_{A'} \langle b_0,b_1,\ldots\rangle = \langle a_0,b_0,b_1,\ldots\rangle$
6. For sequences $\langle a_0,a_1,\ldots\rangle$ and $\langle b_0,b_1,\ldots\rangle$, the latter being in A'_n, if there exists j such that
   $b_0 = b_1 = \ldots = b_{j-1} = F_A$ and $b_j = T_A$

Calling the resultant module $P'[I_1,\ldots,I_k]$, we see that P' is a set of equations

$$X_1 = t_1[X_1,\ldots,X_m,I_1,\ldots,I_k]$$
$$\vdots$$
$$X_m = t_m[X_1,\ldots,X_m,I_1,\ldots,I_k]$$

where $t_1,\ldots,t_m$ are LUCID terms with variables in $X_1,\ldots,X_m$ and $I_1,\ldots,I_k$. Given $A \in \{A\}$ and assignments $Z_1,\ldots,Z_k$ of <u>constant</u> sequences to the input variables $I_1,\ldots,I_k$, the Lemma shows that P' represents a continuous map from

$$[\omega \to A_{s_1'-n}]x\ldots x[\omega \to A_{s_k'-n}]$$

to itself, given by

$$P'[Z_1,\ldots,Z_k](W_1,\ldots,W_m)$$
$$= \langle t_{1_A}[W_1,\ldots,W_m,Z_1,\ldots,Z_k],\ldots,t_{m_A}[W_1,\ldots,W_m,Z_1,\ldots,Z_k]\rangle$$

Note that the sort of the variable X_j is s_j', and that the output variable of P is one of $X_1,\ldots,X_m$. Now the Fixpoint Theorem states that $P'[Z_1,\ldots,Z_k]$ has a least fixpoint. This fixpoint is a tuple $\langle W_1',\ldots,W_m'\rangle$, and assuming that the output variable is X_1, the value of P on input $Z_1,\ldots,Z_k$ is W_1' (recall that the output variable is quiescent, and so W_1' is a constant sequence).

Having defined the value of a LUCID module on a given input, we can now consider each module to be a function; the value $P_A(z_1,\ldots,z_k)$ is $E_{A'}(W_1')$ where $\langle W_1',\ldots,W_m'\rangle$ is the least fixpoint of $P'[Z_1,\ldots,Z_k]$, $Z_1,\ldots,Z_k$ being the constant sequences

$$\langle z_1,z_1,\ldots\rangle,\ldots,\langle z_k,z_k,\ldots\rangle$$

respectively. It is easy to show that P_A is continuous with respect to the flat c.p.o.'s in A, and this suggests the possibility of LUCID modules referencing other LUCID modules in the same way as they reference basic functions in Σ, namely, as point-wise extensions. Let us define a LUCID-program to be an ordered set $\{P_1,\ldots,P_m\}$ of LUCID modules. We attach the name p_j to each module P_j and we will allow the names $p_1,\ldots,p_{j-1}$ to appear in the module P_j in the same contexts as basic function symbols. Note that the sorts of the input variables and of the output variable of P_j determine the type of the symbol p_j, i.e. if $I_1,\ldots,I_k$ have sorts $s_1,\ldots,s_k$ respectively, and if the output variable O of P_j has sort i, then the type of the symbol p_j is $\langle s_1,\ldots,s_k,i\rangle$. Given $A \in \{A\}$, we define the semantics of LUCID programs by induction on the number of modules (observe that if $\{P_1,\ldots,P_m\}$ is a LUCID program then so is $\{P_1,\ldots,P_j\}$ for each $j < m$). For $m = 1$ the semantics is as above, since P_1 cannot reference any other procedures.† The value ascribed to P_1 is $(P_1)_A$. Now assume that the semantics of $\{P_1,\ldots,P_{j-1}\}$ has been defined; let $Q_1,\ldots,Q_{j-1}$ be the (continuous) functions ascribed to $p_1,\ldots,p_{j-1}$ respectively. Let $A'[Q_1,\ldots,Q_{j-1}]$ be the extension of A; to the symbols $p_1,\ldots,p_{j-1}$ so that $(p_\ell)_{A[Q_1,\ldots,Q_{j-1}]}$ is Q_ℓ. Then the value for P_j is $(P_j)_{A[Q_1,\ldots,Q_{j-1}]}$. Hence, the tree-like structure of $\{P_1,\ldots,P_m\}$ allows us to define the function computed by each module inductively, starting at the tips of the tree and proceeding towards the root.

Example We define two modules - one that calculates the square root of an integer, and another that decides whether or not its input is the

† Note that the definition precludes recursion of any form; the results of this paper will be extended to include recursion in a future paper.

the sum of two squares. The basic alphabet consists of the arithmetic operators +, - and *, the relational operator > ("greater than") and the weak equality predicate EQ. Note that all of these operators return the value $\perp$ if either argument is $\perp$. The constants in the alphabet are the natural numbers {0,1,2,...}. First, here is the square root module:

```
root(N):
first X = 0
 next X = X+1
first Y = 1
 next Y = Y+2*X + 3
      0 = X as soon as Y > N
```

The input variable is N, and the output variable is 0. This program has the property that for $N \geq 0$,

$$0*0 \leq N < (0+1)*(0+1).$$

We will formally prove this in section 3. The decision procedure for integers that are sums of squares is

```
sumsquare(I):
first X = 0
next  X = X+1
     S1 = root(X)*root(X)
     S2 = root(I-X)*root(I-X)
    sum = S1 + S2
   test = (sum EQ I) v (X EQ I)
      0 = (sum EQ I) as soon as test
```

The strategy of sumsquare is to check if

$$root(X)^2 + root(I-X)^2 = I$$

for some X between 0 and I. This probably isn't the most efficient way of solving the problem, but the programs do illustrate the use of LUCID modules. The principal advantage is that "block structure" can be implemented; the variables of root are local to the procedure body and it is correct to think of the auxiliary variables of root as being distinct and completely independent of the variables of sumsquare. The reader has probably noticed that we have yet to use a construct of the form IF-THEN-ELSE. The reason why IF-THEN-ELSE is not included as a LUCID function is that it can be a member of Σ. For each sort $i < n$ we use a ternary function symbol ITE of type $\langle 0ii,i\rangle$ and add to D the axiom

$$\forall b(\forall x,y((b \to ITE(b,x,y) = x) \;\&\; (\neg b \to ITE(b,x,y) = y) \;\&\; (b = \perp \to ITE(b,x,y) = \perp)))$$

This axiom completely characterizes IF-THEN-ELSE and it is easily checked that ITE is continuous as defined. As an example of the use of ITE, here is a module that find the largest sum of squares not exceeding its input (in a rather inefficient way)

```
largest-sumsquare(I):
first X = 0
 next X = X+1
first N = 0
 next N = IF sumsquare(next X) THEN
             next X ELSE X
      0 = N as soon as X EQ I
```

We prefer the infixed IF-THEN-ELSE to ITE.

We now turn to the verification of LUCID programs. Let $\{P_1,\ldots,P_m\}$ be a LUCID program, and let us assume that we are to verify a number of input-output conditions for the procedures, i.e. for each procedure P_j we are given Σ-formulae $\tau_j[x]$ and $\pi_j[x,y]$ for which it is alleged that whenever the input condition $\tau_j[x]$ holds, the output condition $\pi_j[x,y]$ holds where y is the result of the computation of P_j on input x (for the sake of brevity, we will consider all modules to have but one input variable - the extension to several input variables is quite trivial). Since some of the modules in $\{P_1,\ldots,P_m\}$ reference other modules, we would like to make some assumptions about the results of modules references within modules. The tree-like structure of $\{P_1,\ldots,P_m\}$ is useful here; if we show that P_1 satisfies the alleged I-O condition without making any assumptions (for P_1 cannot reference any other modules) and then show that for $j = 2,\ldots,m$, P_j satisfies its I-O condition where we assume that $P_1,\ldots,P_{j-1}$ satisfy their I-O conditions, then we have proved that each module in $\{P_1,\ldots,P_m\}$ satisfies its I-O condition. The main result of this section is that there are formulae $\Phi_1,\ldots,\Phi_m$ and $\Psi_1,\ldots,\Psi_m$ such that the above proof procedure can be formalized as

$$\{A'\} \models (\Phi_1 \;\&\; \ldots \;\&\; \Phi_{j-1}) \rightarrow \Psi_j$$

for $j = 1,\ldots,m$. Note that $\{A'\}$ is the class $\{A' : A \in \{A\}\}$ of Σ^L-interpretations, and that we are verifying the properties of the modules in all of the machine interpretations in $\{A\}$.

We denote the LUCID program $\{P_1,\ldots,P_m\}$ by P, and for $A \in \{A\}$, we denote the extension of A to the module names $p_1,\ldots,p_m$ by A_p, where the symbol p_j is interpreted as the function $(P_j)_A$ - i.e. the function computed by P_j under the interpretation A. We denote the conjunction of the assertions in P_j by $(\&P_j)$, and define the formula Ψ_j to be

$$(\underline{\text{first}}\ I = I\ \&\ \tau_j[E(I)]\ \&\ (\&P_j)) \rightarrow \pi_j[E(I),E(0)].$$

The formula Ψ_j asserts that if the value of the input variable is constant, the input condition is satisfied, and the assertions of the module are true then the output condition is satisfied where the value of the (constant) output variable of P_j is substituted for y. Now let Φ_j be the formula

$$\forall x(\tau_j[x] \rightarrow \pi_j[x,p_j(x)]).$$

The formula Φ_j expresses the assumption that P_j satisfies its I-O condition.

Theorem A Let $p = \{P_1,\ldots,P_m\}$ be a LUCID program, and let τ_i and π_i be I-O conditions for $i = 1,\ldots,m$. If, for $j = 1,\ldots,m$, we have

$$\{A'\} \models (\Phi_1\ \&\ \ldots\ \&\ \Phi_{j-1}) \rightarrow \Psi_j$$

then each module of P satisfies its I-O condition.

Remark We will consider the symbolic demonstration of these semantic implications in section 3. We have here a sufficient condition for the modules to satisfy their I-O conditions, but it is not always necessary. We will consider necessity below.

Proof Note that the class {A'} of interpretations does not specify the interpretation of the module names $p_1,\ldots,p_m$. The meaning is that the interpretation of the latter can be arbitrary, but this is of no consequence since the procedure names only appear in the bodies of the modules, and in the conclusions of the formulae Φ_j.

Given $A \in \{A\}$, we must show that $A_P \models \Phi_i$ for $i = 1,\ldots,m$. We use induction on i; let $A \in \{A\}$ and let z be an input to P_i such that $A_P \models \tau_i[z]$. Let u be the assignment of variables of P_i so that $u(I) = \langle z,z,\ldots\rangle$ and the auxiliary variables of P_i are assigned the least fixpoint of the system $P_i'[u(I)]$. Using the premiss of the theorem and the induction hypothesis, we deduce that $A_P \models \pi_i[E(I),E(0)]$ and hence $A_P \models \Phi_i$ as required.

Theorem A gives a sufficient condition for the I-O conditions on P to be true. We are interested in the circumstances under which those conditions are necessary, the reason being that we want to apply our completeness result for many-valued logic to get a symbolic method of verification that will always succeed in verifying conditions that are true. The choice of the formulae τ_i and π_i has been unrestricted so far, but for the converse of Theorem A to hold, we must impose a number of constraints. First, we allow neither τ_i nor π_i to contain occurrences of the procedure names $p_1,\ldots,p_m$. Secondly, the formula $\pi_i[x,y]$ must be monotonic in the variable y, i.e. for each $A \in \{A\}$ and assignments of variables u and u', if u and u' agree except possibly $u(y) \leq u'(y)$ where $\leq$ is the flat c.p.o. on the domain of y, then

$$h_{A[u]}(\pi_i[x,y]) \leq h_{A[u']}(\pi_i[x,y]).$$

This constraint corresponds to π_i being an "admissible predicate" as used in program verification by computational induction [7]. Thirdly, the formula π_i must be categorical in y, i.e. for all x such that $\tau_i[x]$ holds, if there exists a y such that $\pi_i[x,y]$ holds then there exists exactly one such y. This can be expressed as

$$\{A\} \models \forall x,y,y'(\tau_i[x] \;\&\; \pi_i[x,y] \;\&\; \pi_i[x,y'] \rightarrow y=y')$$

and if this is satisfied, then τ_i and π_i effectively define the function computed by P_i. The final condition that must be satisfied is that if a module P_i makes reference to a module P_j (hence $j < i$) then the parameters given by P_i to P_j must satisfy the input condition of P_j. This amounts to ensuring that the I-O conditions provide adequate information for a proof of correctness. We formalize this condition as follows: let $\tau_j^!$[x] be the formula that comes from $\tau_j[x]$ by replacing each symbol in τ_i by its counterpart in Σ' (recall that τ_i is a Σ-formula); hence a symbol of type $\langle s_1 \ldots s_k, i\rangle$ is replaced by its counterpart of type $\langle (s_1+n) \ldots (s_k+n), i+n\rangle$. Now for each occurrence in P_i of a term of the form $P_j(t)$ we must have

$$\{A'\} \models \Phi_1 \;\&\; \ldots \;\&\; \Phi_{i-1} \;\&\; \tau_i[E(I)] \;\&\; (\&P_i) \rightarrow \tau_j^![t].$$

If all of these four constraints are satisfied, then we say that the I-O conditions are admissible.

Theorem B Let $P = \{P_1,\ldots,P_m\}$ be a LUCID program, and let τ_i and π_i be admissible I-O conditions for $i = 1,\ldots,m$. If each of the I-O conditions is correct, then for $j = 1,\ldots,m$ we have

$$\{A'\} \models (\Phi_1 \;\&\; \ldots \;\&\; \Phi_{j-1}) \rightarrow \Psi_j.$$

Remark This is the converse of Theorem A.

Proof We use induction on j, and simultaneously prove

(*) For $j = 1,\ldots,m$ if A^+ is some extension of $A \in \{A\}$ to the module names $p_1,\ldots,p_j$ such that

$$A^+ \models \Phi_1 \;\&\; \ldots \;\&\; \Phi_j$$

then for $k = 1,\ldots,j$ and all assignments u such that $A \models_u \tau_k[x]$ we have

$$(p_k)_A + (u(x)) = (p_k)_{A_p}(u(x))$$

i.e. the interpretation of p_k by A^+ agrees with the function computed by p at all points where the input condition of p_k is satisfied.

We start with the case $j = 1$. Let $A \in \{A\}$ and assume that the premiss of Ψ_1 is satisfied by an assignment of variables u. We must show that $A' \models_u \pi_1[E(I),E(0)]$, thus demonstrating $A' \models_u \Psi_1$ (note that the latter is vacuously true if the premisses of Ψ_1 are not true). If the auxiliary variables of p_1 are $X_1,\ldots,X_m$ then $\langle u(X_1),\ldots,u(X_m)\rangle$ is a solution of $P_1'[u(I)]$. This solution, however, is not necessarily the minimal solution. Let u' be an assignment of variables that agrees with u except that $\langle u'(X_1),\ldots,u'(X_m)\rangle$ is the least fixpoint of $P_1'[u(I)]$. Since the I-O conditions for P are correct, we know that $A' \models_{u'} \pi_i[E(I),E(0)]$. But $u'(0) \leq u(0)$ and since π_i is monotonic in the variable y, we have $A' \models_u \pi_i[E(I),E(0)]$ as required.

To verify the condition (*) for j = 1, we note that $A \models_u \tau_1[x]$ implies both

$$A_P \models_u \pi_1[x,p_1(x)] \text{ (since P is correct)}$$

and $$A^+ \models_u \pi_i[x,p(x)] \text{ (by the above proof).}$$

Now the categoricity of π_i in the variable y shows that

$$(p_1)_{A_P}(u(x)) = (p_1)_{A^+}(u(x))$$

as required.

Now assume the results to be true for $j' < j$, where $j > 1$. Let $A \in \{A\}$ and let A^+ be an extension of A to the symbols $p_1,\ldots,p_{j-1}$ such that

$$A^+ \models \Phi_1 \ \& \ \ldots \ \& \ \Phi_{j-1}.$$

We must show that $A^+ \models \Psi_j$. Let u be an assignment of variables such that the premisses of Ψ_j are satisfied. We want to argue along the same lines as in the case j = 1, but first we must argue that $\langle u(X_1),\ldots,u(X_m)\rangle$ (where $X_1,\ldots,X_m$ are the auxiliary variables of P_j) is greater than or equal to the least fixpoint of $P_j'[u(I)]$. In the case j = 1, this was immediate since P_j does not reference any other modules. In this case, however, P_j may contain references to the modules $P_1,\ldots,P_{j-1}$. The interpretations of $p_1,\ldots,p_{j-1}$ are furnished by A^+, and we can invoke (*) to deduce that the functions $(p_k)_{A_P}$ agree with $(p_k)_{A^+}$ at points that satisfy the input condition for p_k. By the conditions on the modules in P,

we know that if a term $p_k(t)$ occurs in P_j, then each element of the sequence t (when interpreted in A^+) satisfies the input condition τ_k. Hence the vector $\langle u(X_1),\ldots,u(X_m)\rangle$ is also a solution to $P_j'[u(I)]$ in the interpretation A', and we can apply the same reasoning as in case j = 1 to deduce that $A' \models \pi_j[E(I),E(0)]$.

The verification of (*) is as in the case j = 1, using the truth of (*) for j-1 and the proof above that $\Phi_1 \&\ldots\& \Phi_{j-1} \models \Psi_j$. This completes the proof.

We have shown that the verification of a wide class of I-O conditions is equivalent to the verification of a semantic implication. The advantages of this equivalence are twofold: first, the semantic implications only require manipulation of one module at a time - hence the complexity of proofs is reduced; second, we are able to use the equivalence to apply completeness results, and study the characterization of the LUCID control functions by formal axiom systems. This is the topic of the next section.

3. Formal Derivation Systems for LUCID

In section 2, we studied the pure semantics of LUCID without regard to axiomatizations of the LUCID control functions. In this section, we present an axiomatization that is the basis of a formal (symbolic) proof system for LUCID programs. We will see that the axiomatization completely characterizes the LUCID control functions for the purpose of program verification.

Given a continuous, three-valued MVL we have constructed extensions that encompass the syntax and semantics of LUCID programs. However, our extensions are not quite strong enough to axiomatize the fact that LUCID control functions operate on domains of the form $[\omega \rightarrow A]$ and that the basic operations act point-wise on these domains. What we must do is add an indexing set to the extensions, and express the above two concepts by adding an operator that extracts individual values from indexed sequences. Hence the characterization problem for the LUCID control functions reduces to the characterization problem of the canonical indexing set, namely ω; since ω is not characterizable (as noted in the Introduction) it appears that the LUCID control functions cannot be characterized. This is true in the general case, but for the purpose of program verification we use the LUCID functions in a restricted way, namely within programs and in the absence of quantifiers. For this case, a relatively weak axiomatization of the indexing set suffices to characterize the LUCID functions.

For the remainder of this section, $(\Sigma,\{A\},D)$ will be a continuous, full, three-valued MVL, where Σ has n sorts. We have constructed Σ^L as a 2n-sorted alphabet, and we extend Σ^L to a (2n+1)-sorted alphabet Σ^N as follows:

(i) We alter the type of the symbols E_i (of type $\langle i+n,i\rangle$ for $i<n$) to $\langle(2n)(i+n),i\rangle$. E_i is the operator which extracts individual elements from sequences which are indexed by the domain of sort 2n.

(ii) We include the symbols $0 \in \Sigma_{\langle\lambda,(2n)\rangle}$, $=_{(2n)}, < \in \Sigma_{\langle(2n)(2n),0\rangle}$ and $s \in \Sigma_{\langle(2n),(2n)\rangle}$.

The new symbols allow a weak axiomatization of ω; the purpose of 0 should be clear, and s is the successor function. < and = are the "less than" and identity relations respectively. We now extend the derivation system D to include axioms for the index set and for the LUCID control functions. First, the indexing set:

I1. $\forall x(x=0 \vee 0<x)$

I2. $\forall x(0 < s(x))$

I3. $\forall x(x < s(x))$

I4. $\forall x(x > 0 \rightarrow \exists y(x = s(y)))$

I5. $\forall x,y(x < y \rightarrow (y = s(x) \vee s(x) < y))$

I6. $\forall x,y,z(x < y \;\&\; y < z \rightarrow x < z)$

I7. $\forall x(\neg(x < x))$

I8. $\forall x,y(x < y \vee y < x \vee x = y)$

I9. $\forall x,y(x < y \rightarrow x \neq y)$.

This axiomatization is fairly standard. Note that the domain of sort 2n is <u>not</u> meant to be a c.p.o. in any sense. There is no undefined element, and hence all operations and tests on the domain of sort 2n are total (i.e. tests return either T or F). The axioms for the LUCID functions are as follows:

$$\forall X,z(E(z,\underline{\text{first}}\ X) = E(0,X))$$

$$\forall X,z(E(z,\underline{\text{next}}\ X) = E(s(z),X))$$

$$\forall X,Y(E(0,X\ \underline{\text{followed by}}\ Y) = E(0,X))$$

$$\forall X,Y,z(E(s(z),X\ \underline{\text{followed by}}\ Y) = E(z,Y))$$

$$\forall X,B,z(E(z,B)\&\forall z'(z' \prec z \rightarrow \neg E(z',B)) \rightarrow \forall z'(E(z',X\ \underline{\text{as soon as}}\ B) = E(z,X)))$$

$$\forall X,B(\forall z(\exists w(w < z\ \&\ E(w,B) \neq T) \vee \exists w(w < z\ \&\ E(w,B) = \perp)) \rightarrow z(E(z,X\ \underline{\text{as soon as}}\ B) = \perp)$$

The first two axioms define first and next respectively. The next two axioms define followed by and the last two are for as soon as. Finally, we include axioms that define the duplicate symbols in Σ' to be point-wise extensions of the symbols in Σ, and an axiom which states that if some member of a sequence is equal to x, then there exists a minimal indice such that the corresponding element of the sequence is x. This imposes a constraint on the order $<$ on the index domain, but note that it does not make $<$ a well-order. The axiom is included so that as soon as is well-defined: if some member of a sequence is true, then as soon as must select the indice of the first occurrence of T in the sequence. The axioms are as follows:

(i) For each symbol $f \in \Sigma$,

$$\forall X_1,\ldots,X_k(\forall z(E(z,f'(X_1,\ldots,X_k)) = f(E(z,X_1),\ldots,E(z,X_k))))$$

(ii) $$\forall X,x(\exists z(E(z,X) = x) \rightarrow \exists_{min} z(E(z,X) = x))$$

where $\exists_{min} x(\phi[x])$ abbreviates the formula $\exists z(\phi[z]\ \&\ \forall z'(z' < z \rightarrow \neg(\phi[z'])))$.

Let D^N denote the result of extending D as above. Now {A} is the set of all countable models of the axioms in D; let $\{A^L\}$ denote the set of all countable models of D^N. Each member A^L of $\{A^L\}$ can be considered as a triple (A,A^+,N) where $A \in \{A\}$, N is a model of the axioms I1-I9 and

$$A^+ \subset A^N = \{[N \rightarrow A_i]:i < n\}$$

Note that A^+ is a <u>proper</u> subset of A^N, since the latter is uncountable. Hence A^+ does not contain all N-indexed sequences, but this is of no consequence. In section 2, the Σ^L-interpretation A' was defined, and it should be clear that A' is essentially the same as (A,A^ω,ω). We showed how to define the semantics of LUCID programs under interpretations of the form A', and the definition of semantics under the interpretation A^N is straightforward: all of the functions on the domains of A^N are continuous - a fact which is easily verified. Hence for each LUCID program $P = \{P_1,\ldots,P_m\}$ and interpretation (A,A^N,N) we can define the function computed by P under the interpretation A^N. If N is not ω, then we can consider P to be executing with "non-standard" control functions, since N is a non-standard model of the axioms for ω. It seems possible that non-standard control functions will cause P to compute different functions; such is not the case, as we will show. The axioms I1-I9 are strong enough to ensure that the function computed by P under the interpretation A^N is the same as the function computed under the interpretation A^ω. This result shows that the axioms for the LUCID functions characterize them for the purpose of computation with LUCID, and a consequence of this is the completeness

result for program verification. Before proving the characterization of the LUCID functions, we prove some technical results.

Lemma 1 Each model N of the axioms I1-I9 is an end-extension of ω.

Proof For N to be an end-extension of ω means that N contains a copy of ω, and for each element x of the domain of N, either x is a member of the copy of ω, or x is greater than every element of the copy of ω, under the order $<_N$. The copy of ω within N is clearly $\{0_N, s_N(0_N), \ldots\}$. If x is a domain member that is not in this set, then we can prove $x > y$ for all y in the set by an induction argument: by axiom I1, $x > 0$; if $x > y$ then axiom I5 asserts that either $x = s(y)$ (in which case x is in the set) or $x > s(y)$. By hypothesis, only the latter can be true, and hence $x > y$ for all y in the set. This completes the proof.

Lemma 1 is simply an assertion that the axioms I1-I9 have a certain degree of strength. It shows that ω is an "initial segment" of each model of the axioms. Let N be a model of I1-I9, and let X be an N-indexed sequence. We say that X is finitary if there exists $n \in \omega$ such that $X(n) = X(z)$ for all $z \in N$ such that $z > n$ (note that we now consider all models N of I1-I9 to contain ω). Hence X is finitary if it only contains finitely many distinct elements. The next lemma shows that LUCID terms preserve finitary sequences.

Lemma 2 Let $t[X_1,\ldots,X_k]$ be a LUCID term, and let $A \in \{A\}$ and N be a model of I1-I9. Let u be an assignment of variables to A^N such that each $u(X_i)$ is finitary. Then the sequence $h_{A^N[u]}(t[X_1,\ldots,X_k])$ is finitary.

Proof We use structural induction on t. If t contains no LUCID functions, then t is (essentially) the point-wise extension of a Σ-term. In this case, the result is immediate. If we assume that result to be true for the terms s and s', and consider the cases where t is first s, next s, s followed by s' and s as soon as s' then the result follows immediately. Note that a first or as soon as expression always evaluates to a constant sequence, which is clearly finitary. This completes the proof.

Given that LUCID terms preserve finitary sequences, we now consider the interpretation of terms in the standard control structure ω, and in a non-standard structure N. Given an ω-indexed sequence X and an N-indexed sequence Y, we say that "X is contained in Y" if for each $n \in \omega$, $X(n) = Y(n)$.

Lemma 3 Hypothesis as in Lemma 2. Let u' be an assignment of variables to A^ω such that $u'(X_i)$ is contained in $u(X_i)$ for $i = 1,\ldots,k$. Then $h_{A^\omega[u']}(t)$ is contained in $h_{A^N[u]}(t)$.

Proof We use structural induction on t. The case analysis is as in Lemma 2, and the only non-trivial case is the terms of the form s as soon as s', where the result is assumed true for s and s'. We must show that if $h_{A^\omega[u']}(t)$ is undefined, then so is $h_{A^N[u]}(t)$ (the converse is trivial since $h_{A^\omega[u']}(s')$ is contained in $h_{A^N[u]}(s')$). This, however, is immediate by Lemma 2, since if it were not true then we would have

$$h_{A^\omega[u']}(s') = \langle F_A, F_A, \ldots, F_A, \ldots \rangle$$

and

$$h_{A^N[u]}(s')(z) = T_A \text{ for some } z \text{ outside of } \omega.$$

which is a contradiction. (Note the use of the induction hypothesis here.) If both $h_{A^N[u]}(t)$ and $h_{A^\omega[u']}(t)$ are defined, then the latter must be contained in the former since $h_{A^\omega[u']}(s')(n)$ must be true for some $n \in \omega$. This completes the proof.

Given a LUCID program $P = \{P_1,\ldots,P_m\}$, $A \in \{A\}$ and a model N of I1-I9, let A_P^N denote the extension of A^N to the symbols $p_1,\ldots,p_m$ so that p_j is assigned the function computed by P_j in A^N.

The Characterization Theorem For each model N of I1-I9 and $i = 1,\ldots,k$ the functions $(p_i)_{A^\omega}$ and $(p_i)_{A^N}$ are identical.

Proof We prove the theorem for the case $k = 1$. The extension to all k is quite trivial, given the method of section 2 for assigning semantics to LUCID programs. Let a be an input to the module P_1, and let I and I' be $\langle a,a,\ldots\rangle$ and $\langle a:i \in N\rangle$ respectively. Now $(p_1)_{A_P^\omega}(a)$ is the least fixpoint of the system $P_1'[I]$ of equations under the interpretation A^ω; this fixpoint is, by the Fixpoint Theorem, of the form

$$\sqcup\{\langle t_1^k[\bot,\ldots,\bot,I],\ldots,t_m^k[\bot,\ldots,\bot,I]\rangle : k \in \omega\}$$

where the assertions of $P_1'[I]$ are

$$X_1 = t_1[X_1,\ldots,X_m,I]$$
$$\vdots$$
$$X_m = t_m[X_1,\ldots,X_m,I]$$

The same is true of $(p_1)_{A_P^N}(a)$, except that the interpretation of the terms $t_1,\ldots,t_m$ is in A^N rather than A^ω. Now each element of the above tuple is a sequence, and the evaluation of the least upper bound of sequences is point-wise. It follows that we need only show

$$h_{A^N}(t_i^j[\perp,\ldots,\perp,I]) \qquad (1)$$

and

$$h_{A^\omega}(t_i^j[\perp,\ldots,\perp,I]) \qquad (2)$$

to agree at points in ω (i.e. (2) is contained in (1)) for all $j \in \omega$ and $i = 1,\ldots,m$. We will then have shown that the least upper bounds agree at points in ω,and since the output variable is quiescent, we will have proved that

$$(p_1)_{A^N}(a) = (p_1)_{A^\omega}(a)$$

as required. But the sequences $\perp$ and I are clearly finitary; hence inductive application of Lemmas 2 and 3 shows that (1) and (2) are finitary, and agree at points in ω for all $j \in \omega$ and $i = 1,\ldots,m$. This completes the proof.

We now return to the verification of LUCID programs. In section 2, we defined the formulae Φ_i and Ψ_i; if, in the formulae Ψ_i, we replace the terms E(I) and E(0) by E(0,I) and E(0,0) respectively, then Theorems A and B assert that a LUCID program $P = \{P_1,\ldots,P_m\}$ is correct in all interpretations in {A} iff

$$A^\omega \models (\Phi_1 \;\&\; \ldots \;\&\; \Phi_{j-1}) \rightarrow \Psi_j$$

for $j = 1,\ldots,k$, and all $A \in \{A\}$. If we show this to be equivalent to

$$\{A^L\} \models (\Phi_1 \;\&\; \ldots \;\&\; \Phi_{j-1}) \rightarrow \Psi_j$$

for $j = 1,\ldots,k$, then we can invoke the completeness theorem for three-valued MVL to deduce the equivalent

$$D^N \models (\Phi_1 \;\&\; \ldots \;\&\; \Phi_{j-1}) \rightarrow \Psi_j$$

which gives us a complete symbolic method of verification.

The Verification Completeness Theorem Let $P = \{P_1,\ldots,P_m\}$ be a LUCID program. Then the modules in P are correct with respect to admissible I-O conditions τ_j, π_j if and only if

$$D^N \models (\Phi_1 \;\&\; \ldots \;\&\; \Phi_{j-1}) \rightarrow \Psi_j \qquad (1)$$

for $j = 1,\ldots,k$.

Proof We use the method outlined above, i.e. we show that

$$A^\omega \models (\Phi_1 \;\&\; \ldots \;\&\; \Phi_{j-1}) \rightarrow \Psi_j \qquad (2)$$

for all $A \in \{A\}$ iff

$$\{A^L\} \models (\Phi_1 \;\&\; \ldots \;\&\; \Phi_{j-1}) \rightarrow \Psi_j \qquad (3)$$

First, if (3) holds, then (1) holds by the completeness theorem for three-valued MVL. Hence (2) holds since the derivation system D^N is consistent. The converse is not quite so trivial. If (2) holds for all $A \in \{A\}$, then let $A^L \in \{A^L\}$ and assume that

$$A^L \models \Phi_1 \;\&\; \ldots \;\&\; \Phi_{j-1}$$

and that u is an assignment of variables such that the premisses of Ψ_j are true, i.e.

$$A^L \models_u \underline{\text{first}}\ I = I$$

and $$A^L \models_u \tau_j[E(0,I)]$$

and $$A^L \models_u (\&P_j)$$

Now the values of the variables of P_j under the assignment u represent a solution to the equations in $P_j'[u(I)]$; however, they need not represent the least solution. Let u' be the assignment of variables such that u' agrees with u except that the variables of P_j are assigned the least fixpoint of $P_j'[u(I)]$. By the Characterization Theorem, we know that

$$u'(0)(0) = (p_j)_{A_P^\omega}[u(I)(0)]$$

and so, since P_j is correct with respect to the I-O conditions τ_j and π_j, we have

$$A' \models_{u'} \pi_j[E(I),E(0)]$$

Now $u'(0) \leq u(0)$, and $\pi_j[x,y]$ is monotonic in the variable y. Hence

$$A^L \models_u \pi_j[E(0,I),E(,0)]$$

as required. This completes the proof.

The Verification Completeness Theorem shows that if a program is correct in all models of the machine axioms, then a symbolic proof of the correctness if possible. A corollary of this is that proof by contradiction can be used without sacrificing completeness, i.e. the program P is correct if and only if we can deduce a contradiction from the assumption

$$\Phi_1 \ \& \ \ldots \ \& \ \Phi_{j-1} \ \& \ \neg\Psi_j$$

Conventional techniques, such as resolution, can be applied to find contradictions from such assumptions, and it is certainly of practical advantage to be able to incorporate proof by contradiction into an implementation.

We end this section by formally verifying the program

{root, sumsquare}

of section 2. The I-Q conditions for root are

$$\tau_r[x] : x \geq 0$$

$$\pi_r[x,y] : y*y \leq x \ \ \& \ \ x < (y+1)*(y+1)$$

and the I-O conditions for sumsquare are

$$\tau_s[x] : x \geq 0$$

$$\pi_s[x,y] : y \vee \neg y \ \& \ (y \rightarrow \exists a,b(a*a+b*b = x))$$
$$\& \ (\neg y \rightarrow \forall a,b(a*a+b*b \neq x)).$$

Note that our alphabet is normal arithmetic on the integers. The first conjunct in π_s asserts that y is defined; this is not needed in π_s since $\pi_r[x,\perp]$ can never be true, whereas $\pi_s[x,\perp]$ is always true in the absence of the conjunct $y \vee \neg y$. To verify root, we must show that $D \vdash \Psi_r$. The deduction theorem for (virtually any) system of logic states that to prove $\alpha \rightarrow \beta$ it suffices to take α as a hypothesis, and prove β without generalizing over any free variables of α. Hence we make the hypotheses

(1) first I = I

(2) E(0,I) ≥ 0

(3) &root

First, we demonstrate the termination of the statement

$$0 = X \text{ as soon as } Y > N$$

Using the termination rule

$$\forall Z(Z > \text{next } Z \rightarrow \forall n \exists w(E(w,Z) \geq n)$$

and noting that

$$Y + 2*X + 3 > Y$$

i.e. next $Y > Y$, we can use the rule to introduce a constant c and the deduction

$$E(c,Y) > E(0,N) \ \& \ \forall w(w < c \rightarrow E(w,Y) \leq E(0,N)) \qquad (4)$$

By the axiom for as soon as and (4), we can deduce $E(0,0) = E(c,X)$. We must therefore show that

$$E(c,X*X) \leq E(0,N) < E(c,(X+1)*(X+1))$$

First, we show that $Y = (X+1)*(X+1)$. We use the LUCID induction formula

$$(\text{first } P) = T \ \& \ (P \rightarrow \text{next } P) = T \rightarrow P = T$$

This formula is a consequence of the axiom (ii) on page 42 .
The variable P is of sort n, i.e. is a sequence of truthvalues. Here, we substitute the formula $Y =' (X+1)*(X+1)$ for P; note that =' is point-wise equality. Using the point-wise nature of =', + and * we find that first P is

$$\text{first } Y =' (\text{first } X+1)*(\text{first } X+1)$$

i.e. $(1 =' (0+1)*(0+1))$

where we use the definitions of X and Y to reduce the expression. We clearly have an arithmetic identity, and so we conclude that first P is true. We must now show

$$P \rightarrow \text{next } P$$

This reduces to

$$\forall w(E(w,Y =' (X+1)*(X+1)) \rightarrow E(s(w),Y =' (X+1)*(X+1)))$$

Using the definitions of X and Y, this reduces to

$$y = (x+1)*(x+1) \rightarrow y+2*x+3 = (x+2)*(x+2)$$

which is an algebraic truth. We now have

$$E(c,Y) = E(c,(X+1)*(X+1)) \qquad (5)$$

Combining this with (4), we get

$$E(0,N) < E(c,(X+1)*(X+1)) \qquad (6)$$

which is half of what we set out to prove. To complete the proof, we must show that

$$E(c,X*X) \leq E(0,N)$$

We use proof by contradiction. Note that both E(c,X) and E(0,N) are defined, the former by nature of its definition, and the latter by hypothesis (2). So we can assume

$$E(c,X*X) > E(0,N)$$

for the purpose of reduction. We consider two cases: if c = 0 then since E(0,X) = 0, we have E(0,N) < 0, contradicting (2). Now, assuming that c > 0, there exists c' such that c = s(c'). Hence

$$E(c,X*X) = E(c',(X+1)*(X+1)) = E(c',y)$$

But now E(c',Y) > E(0,N). Since c' < c, this contradicts (4), and completes the proof.

We now verify the module sumsquare; we must show

$$D \vdash \Phi_r \rightarrow \Psi_s$$

and as above, we use the deduction theorem. Our hypotheses are

(1) $x \geq 0 \rightarrow (root(x)*root(x) \leq x \;\&\; x < (root(x)+1)*(root(x)+1)$

(2) first I = I

(3) $I \geq 0$

(4) &sumsquare

First, we show that $E(0,0) \vee \neg E(0,0)$ holds, i.e. that the statement

0 = (sum EQ I) as soon as test

terminates. By the definition of test, we need only show that $E(w,X) = E(0,I)$ for some w. This follows from the termination rule

$x \geq 0$ & first Z = 0 & next Z = (Z+1) $\rightarrow \exists w(E(w,Z) = x)$

Hence we can introduce a new constant c, and the deduction

(5) $E(c,test) \;\&\; \forall w(w < c \rightarrow E(w,test))$

Note that we do not specify which disjunct of test is true at time c; we have demonstrated that the disjunct (X EQ I) will be true at some time, but it is possible that (sum EQ I) is true before (or at) time c. We now prove

$E(0,0) \rightarrow \exists a,b(a*a+b*b = x)$

This is immediate, since by the definition of 0,

(6) $E(0,0) \rightarrow E(c,sum\ EQ\ I)$

and hence

(7) $E(0,0) \rightarrow E(c,S1) + E(c,S2) = E(0,I)$

The definitions of S1 and S2 show that E(c,S1) and E(c,S2) are both squares, and hence the existence of a and b such that $a*a+b*b = E(0,I)$ has been demonstrated. We must now show that

$\neg E(0,0) \rightarrow \forall a,b(a*a+b*b \neq E(0,I))$

We use proof by contradiction. Assume E(0,0) and

(8) p*p + q*q = E(0,I)

for some integers p and q. We may assume that p and q are non-negative, and we claim that root (p*p) = p and root(q*q) = q. This is to be expected, of course, but we did not prove that root finds the exact square root of a square number. Assume that root(p*p) ≠ p; by (1), we know that

root(p*p)*root(p*p) ≤ p*p

By our hypothesis, we have removed the option of equality, and so (taking the actual square root of each side) we have

root(p*p) < p

But now root(p*p)+1 ≤ p, and hence

(root(p*p)+1)*(root(p*p)+1) ≤ p*p

contradicting (1). Hence root(p*p) = p, as required. In the same way, root(q*q) = q, and since p*p+q*q = E(0,I) there must exist d such that

E(d,S1) = p*p and E(d,S2) = q*q

But then E(d,sum EQ I) is true. Noting that p ≤ E(0,I) and q ≤ E(0,I), we see that d ≤ c and this contradicts the hypothesis that E(0,0) is false. This completes the proof.

4. Conclusions

We have seen that the control functions of LUCID can be characterized by a formal axiom system, and that this characterization yields a symbolic verification system, the power of which is wholly dependent on the axiomatization of the machine functions. The constraints placed on the I-O conditions of programs were rather strict from a practical point of view; in particular, the exclusion of formulae which describe I-O properties in terms of the results of other modules is rather unfortunate, although, theoretically, such formulae are not needed for complete generality.

Perhaps the most notable omission from the control structures considered is that of recursion. A paper is in preparation in which the characterization and completeness results of this paper are extended to include recursion; as a bonus, the constraints on the I-O conditions mentioned above are relaxed quite considerably. Another control structure that may be investigated is that of non-determinism, and perhaps, ultimately, that of parallelism. We hope that the analysis of control structures will lead to better understanding of the semantics of programming languages, and the verification of programs.

5. References

[1] Ashcroft, E.A. and Wadge, W.W., "Demystifying Program Proving - an Informal Introduction to LUCID" Research Report CS-75-02, Computer Science Dept., University of Waterloo, Waterloo, Ontario.

[2] _____, "LUCID - A Formal System for Writing and Proving Programs", Research Report CS-75-01, Computer Science Dept., University of Waterloo, Waterloo, Ontario.

[3] Bell, J.L. and Slomson, A.B., "Models and Ultraproducts: An Introduction", North Holland Publishing Company.

[4] Hoare, C.A.R., "An Axiomatic Basis for Computer Programming" Comm. A.C.M. 12 (1969), No.10.

[5] Manna, Z., "The Correctness of Programs", Journal of Computer and System Sciences 3 (1969).

[6] _____, "The Fixpoint Approach to the Theory of Computation", Comm. A.C.M. 15 (1972), No.7.

[7] _____, Ness, S. and Vuillemin, J., "Inductive Methods for Proving Properties of Programs", Proc. A.C.M. Conference on Proving Assertions about Programs, A.C.M., New York, 1972.

[8] Mendelson, E., "Introduction to Mathematical Logic", Van Nostrand, Princeton.

[9] Raslowa, H. and Sikorski, R., "A Proof of the Completeness Theorem of Godel", Fund. Math. 37 (1951).

[10] Tarski, A. cf. [9].