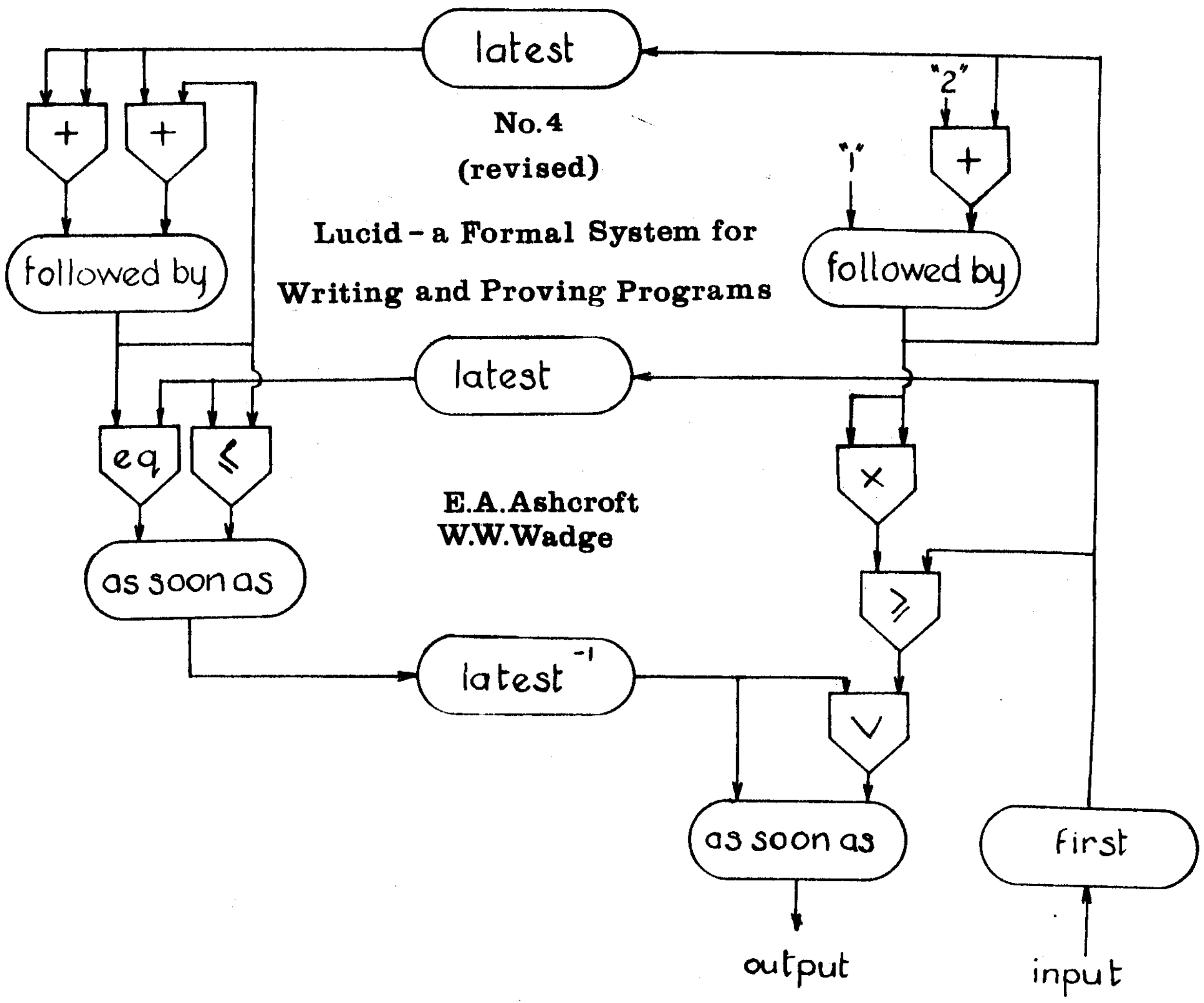
The University of Warwick

THEORY OF COMPUTATION REPORT



No.4

(revised)

Lucid - a Formal System for Writing and Proving Programs

E.A.Ashcroft

W.W.Wadge

Department of Computer Science
University of Warwick
COVENTRY CV4 7AL
ENGLAND.

January 1976

Lucid - A Formal System for Writing and Proving Programs

by

E.A. Ashcroft
Department of Computer Science
University of Waterloo, Waterloo, Ontario, Canada

and

W.W. Wadge
Computer Science Department
University of Warwick
Coventry, England

A condensed version of this paper was presented at the International Symposium on Proving and Improving Programs, 1975.

Abstract

Lucid is both a programming language and a formal system for proving properties of Lucid programs. The programming language is unconventional in many ways, although programs are readily understood as using assignment statements and loops in a 'structured' fashion. Semantically, an assignment statement is really an equation between 'histories', and a whole program is simply an unordered set of such equations.

From these equations, properties of the program can be derived by straightforward mathematical reasoning, using the Lucid formal system. The rules of this system are mainly those of first order logic, together with extra axioms and rules for the special Lucid functions.

This paper formally describes the syntax and semantics of programs, and justifies the axioms and rules of the formal system.

Keywords

Program proving, Formal semantics, Formal systems

0. Introduction

Lucid is both a language in which programs can be written, and a formal system for proving properties of such programs. These properties are also expressed in Lucid. This is possible because a Lucid program is itself simply an unordered set of assertions, or axioms, from which other assertions may be derived by fairly conventional mathematical reasoning. The statements in Lucid programs are special cases of Lucid terms.

In this paper we present the formal basis for Lucid, giving its semantics and justifying various axioms and rules of inference that are used in Lucid proofs. An informal introduction to Lucid can be found in [1], together with a discussion of implementation considerations. This paper will be rather formal, with motivating explanations and examples confined mainly to this introduction.

The language considered here might be called Basic Lucid, since it does not include features like arrays and defined functions. Such extensions are considered in [1].

The main idea in Lucid is that programs should be 'denotational' and 'referentially transparent', even when they contain assignment statements. This means that all expressions in a program must mean something, and that two occurrences of the same expression in a program must denote the same 'something'. Lucid achieves this aim, and yet manages to treat assignment statements as equations. (Thus, if a Lucid program contains the (assignment) statement Y = X + Z, every occurrence of Y in the program can be replaced by X + Z, without changing the meaning of the program.) This is accomplished by considering the program to be talking about the 'histories' of the various variables. Semantically, all expressions in programs without nested loops will denote infinite sequences of data objects. Reassignment to a variable must be done by using the special Lucid function next, and the initialisation of a variable must also be explicit, by using the function first. Thus the two statements first X = 0, next X = X + 1 define the value, or history, of X to be the infinite sequence <0,1,2,3,...>. (The numeral 1 denotes <1,1,1,...>, and + works pointwise. The function next drops off the first item of its argument.) Note that these two statements imply the existence of a 'loop', and explicit control statements are unnecessary. Also, the order of the two statements is irrelevant. If we also have first Y = 0, next Y = Y + X×X, then this loop also generates a history for Y, namely, <0,0,1,5,14,...>. We can get out of the infinite iteration using the Lucid function as soon as: e.g. output = Y as soon as X > 3 gives the variable output the value of Y when X > 3 is first true, i.e. the fifth value, 14. (In fact, output is <14,14,14,...>.) With these three functions it is possible to write programs without nested loops, in a very natural way.

For programs with nested loops we must generalise our notion of 'history'. Consider the following Lucid program, which determines whether the first integer N on the input stream is a prime number or not.

Program Prime

```
N = first input
first I = 2
begin
   first multiple = I×I
   next multiple = multiple + I
   IdivN = multiple eq N as soon as multiple ≥ N
end
next I = I+1
output = ¬IdivN  as soon as  IdivN ∨ I×I ≥ N.
```

The program contains one loop within another. The inner loop is delimited by begin and end. Intuitively, the outer loop generates successive values of potential divisors I of N, starting 2,3,4,..., and, for each value of I, the inner loop generates successive multiples of I, beginning with I^2. The variable IdivN is set true or false depending on whether or not a multiple of I is found which is equal to N. In the outer loop, output is set false or true depending on whether IdivN is ever true or not.

The predicate 'eq' is like '=' except that its value is undefined if either of its arguments is undefined (of course undefined = undefined is true).

The program as it stands is not strictly speaking a set of assertions because of the begin and end. Informally, the effect of begin and end is to "freeze" the values of the global variables I, N and IdivN. (The global variables of a loop are all those variables mentioned outside the loop.) The begin and end can be removed by replacing all enclosed occurrences of I, N and IdivN by latest I, latest N and latest IdivN. (The meaning of the function latest will be given later.) Thus the first line of the inner loop becomes first multiple = latest I × latest I. The resulting transformed program Prime' is an unordered set of assertions which can be used as axioms from which to derive further assertions.

In practice, it is easier to write programs using the begin ... end notation rather than latest and, moreover, we have rules of inference which allow us to carry out proofs of programs in the begin ... end notation, without using latest, as follows. We keep track of the loop associated with a program statement or other assertion that we have derived. Informally, an assertion that does not contain any of the special Lucid functions can be moved into loops, and can be moved out of loops if in addition it only refers to global variables of the loop. Moreover, within a loop we can add the assertion X = **first** X for any global variable X (which states that X is *quiescent*, i.e. constant for the duration of the loop). When proving things "within a loop" we may only use statements from within the loop (which may have been brought there or have been added as above).

This might be called the technique of "nested proofs". It reduces reasoning about nested loops to reasoning about simple loops. Before we discuss generalized histories we can illustrate this sort of reasoning by deriving from the statements of Prime and the assumption **first** input > 0 the assertion

$$\text{Output} = \neg \exists L \exists K\ 2 \le K < \textbf{first}\ \text{input} \wedge L \times K = \textbf{first}\ \text{input}.$$

Proof

We will assume the only data objects are the integers, *true*, *false* and the special object *undefined*.

The first step is to prove the correctness of the inner loop. We introduce a new variable J by setting **first** J = I and **next** J = J+1 so that between the *begin* and *end* we have

```
first J = I
next J = J+1
first multiple = I×I
next multiple = multiple + I
IdivN = multiple eq N  as_soon_as  multiple ≥ N.
```

Since J does not appear elsewhere in the program, any assertion not involving J which is provable from the expanded program can be proved from the original. With J so defined we can prove

(1) $\text{multiple} = I \times J.$

The proof uses the basic Lucid induction rule:

(R1) $\underset{\sim}{\text{first}}\ P,\ P \rightarrow \underset{\sim}{\text{next}}\ P \models P$

where for any assertion A and set Γ of assertions, $\Gamma \models A$ means that the truth of A is implied by the truth of every assertion in Γ.

If we let P be "multiple = I×J" then

$$\begin{aligned}\underset{\sim}{\text{first}}\ P &= \underset{\sim}{\text{first}}\ (\text{multiple} = I\times J)\\ &= (\underset{\sim}{\text{first}}\ \text{multiple} = \underset{\sim}{\text{first}}\ I \times \underset{\sim}{\text{first}}\ J)\\ &= (I\times I = I\times I)\end{aligned}$$

which is true (we used the fact that $\underset{\sim}{\text{first}}\ I = I$ inside the inner loop).

Now we assume that P is true at some stage, i.e. we assume multiple = I×J. Then,

$$\begin{aligned}\underset{\sim}{\text{next}}\ (\text{multiple} = I\times J) &= (\underset{\sim}{\text{next}}\ \text{multiple} = \underset{\sim}{\text{next}}\ I \times \underset{\sim}{\text{next}}\ J)\\ &= (\text{multiple} + I = I\times(J+1))\\ &= (\text{multiple} + I = I\times J + I)\end{aligned}$$

which is true because of the induction assumption. We can discharge the assumption P, giving $P \rightarrow \underset{\sim}{\text{next}}\ P$, and so we have proved multiple = I×J by induction.

We also used an axiom which says that $\underset{\sim}{\text{first}}$ and $\underset{\sim}{\text{next}}$ 'commute' with conventional operations like "+": for any expression A not containing any special Lucid functions and having free variables $X_1, X_2, \ldots, X_K$ we have

(A1) $\models$ first A = A(X_1/first X_1,...,X_K/first X_K)

(A2) $\models$ next A = A(X_1/next X_1,...,X_K/next X_K),

where A(X/Q) denotes term A with free variable X replaced by term Q.

Having proved multiple = I×J we can replace any occurrence of multiple in our program by I×J. The program can then be simplified, to give program Prime_1:

```
N = first input
first I = 2
begin
   first J = I
   next J = J+1
   IdivN = I×J eq N  as soon as  I×J≥N
end
next I = I+1
output = ¬ IdivN as soon as  IdivN ∨ I×I≥N
```

and if A is any assertion without free occurrences of J then Prime $\models$ A iff $\text{Prime}_1 \models$ A.

To finish the proof of correctness of the inner loop we must determine the value of IdivN.

To do this we first introduce the function hitherto, defined by

(A3) $\models$ first hitherto P = T

$\wedge$ next hitherto P = P $\wedge$ hitherto P.

Using this we can establish by induction that

(2) hitherto (I×J<N) $\rightarrow$ ($\forall$K I≤K<J $\rightarrow$ I×K<N).

If we define firstime P to mean P $\wedge$ hitherto ¬P, we can conveniently state an axiom for the function as soon as, and a rule for firstime:

(A4) $\models$ firstime $P \to X$ as soon as $P = X$

(R2) firstime $P \to$ first Q, eventually $P \models$ first Q.

The first states that the value of X as soon as P is the value of X when P is true for the first time. The second states that if some property Q holds when P is true for the first time, and Q is quiescent and P does eventually become true, then property Q holds.

Using (2) we can establish

(3) firstime $(I \times J \geq N) \to I \times J$ eq $N = (\exists K\ I \leq K < N \wedge I \times K = N)$.

(A proof of this step can be found in [2].)

Since (A4) gives us

firstime $(I \times J \geq N) \to$ IdivN $= I \times J$ eq N

we can conclude

firstime $(I \times J \geq N) \to$ IdivN $= (\exists K\ I \leq K < N \wedge I \times K = N)$.

Since the term on the right-hand side is quiescent, to apply rule R2 we simply need to prove eventually $(I \times J \geq N)$. For this we use the following 'termination' rule for integers:

(R3) integer L, $L >$ next $L \models$ eventually $(L \leq 0)$.

To apply R3 we first prove that integer $(N - I \times J)$ and $N - I \times J >$ next $(N - I \times J)$. This is straightforward (but note that $I > 0$ must be established by induction in the outer loop, and then brought into the inner loop).

Now, applying the as soon as rule R2, we get

(4) $\qquad \mathrm{IdivN} = \exists K\ I \le K < N \wedge I \times K = N.$

Assertion (4) contains no Lucid functions and all its free variables are globals, and so it may be taken outside the inner loop. We now discard the inner loop yielding 'program' Prime_2:

```
N = first input
first I = 2
IdivN = ∃K I≤K<N ∧ I×K = N
next I = I+1
output = ¬ IdivN  as soon as  IdivN ∨ I×I≥N
```

and as before $\mathrm{Prime}_2 \models A$ implies $\mathrm{Prime} \models A$ for any assertion A. Actually Prime_2 is no longer a program but rather a hybrid object halfway between a program and statement of correctness.

Note that IdivN is always either true or false (it could be undefined if N were undefined, but we know N > 0).

Now to finish the proof that

$$\mathrm{output} = \neg \exists L \exists K\ 2 \le K < N \wedge L \times K = N$$

we must first show that

(5) $\qquad \mathrm{firstime}\ (\mathrm{IdivN} \vee I \times I \ge N) \to \mathrm{IdivN} = \exists L \exists K\ 2 \le K < N \wedge L \times K = N.$

The proof of this is similar to the proof of (3), provided we first prove that

(6) $\qquad \mathrm{hitherto}\ \neg\, \mathrm{IdivN} \to (\forall L\ L < I \to (\neg \exists K\ 2 \le K < N \wedge L \times K = N)).$

This requires a straightforward induction proof, making extensive use of properties of integers, and the property $N > 0$.

Since IdivN is always either true or false, to establish the second premise for the as soon as rule, namely, eventually $(\mathrm{IdivN} \vee I{\times}J{\geq}N)$, it is sufficient to show that eventually $(I{\times}I{\geq}N)$. This follows from the termination rule R3.

So finally, we can eliminate all the variables except output leaving 'program' Prime_3:

$$\mathrm{output} = \neg\, \exists L \exists K\ 2{\leq}K{<}\mathbf{first}\ \mathrm{input} \wedge L{\times}K = \mathbf{first}\ \mathrm{input}. \qquad \square$$

Note that $\neg\, \exists L \exists K\ 2{\leq}K{<}\mathbf{first}\ \mathrm{input} \wedge K{\times}L = \mathbf{first}\ \mathrm{input}$ is either true or false, when integer first input. Thus output is not undefined, and so program Prime terminates with the correct result.

This sample proof shows that it is possible to reason about programs knowing very little of the formal semantics, in particular knowing very little about the semantics of nested loops. But we must give a semantics for nested loops to justify the nested proof style of reasoning.

In the program Prime, the history of I can be thought of as $\langle 2,3,4,\ldots\rangle$, but the history of multiple must be $\langle\langle 4,6,8,\ldots\rangle, \langle 9,12,15,\ldots\rangle, \langle 16,20,24,\ldots\rangle,\ldots\rangle$, i.e. a two dimensional infinite sequence. A one dimensional infinite sequence can be considered as a function from the natural numbers N (including zero) to data elements, and, similarly, a two dimensional sequence is a function from $N{\times}N$ to data elements. If we write I_n instead of $I(n)$, we see that $I_n = n+2$. For two dimensions, we adopt the convention that the first subscript is the more rapidly varying time parameter, the number of iterations of the inner loop. Thus $\text{multiple}_{nm} = (m+2)(m+n+2)$. The Lucid functions (except latest) act on the first time parameter, e.g. $(\text{first multiple})_{nm} = \text{multiple}_{0m}$.

To make this work we need to do two things. Firstly, we get rid of the begin ... end notation as indicated previously, by introducing the function latest which increases the number of time dimensions, e.g. $(\text{latest } I)_{nm} = I_m$. (Note that latest I is quiescent.) Secondly, we unify the treatment of variables at different levels of loop nesting by considering all histories as depending on an infinite number of time parameters (only a finite number of which will usually be necessary for each variable).

Thus, in the rest of this paper we consider Lucid programs that use latest instead of begin ... end, and the semantics of Lucid is given in terms of functions of infinite sequences of time parameters.

As a formal system Lucid is similar, in some respects, to first order logic. On the other hand, Lucid can be viewed as a tense logic, a branch of modal logic which formalises certain kinds of reasoning about time. (The suitability of modal logic for proofs about programs has already been recognised by Burstall [3].) In Lucid a term, such as $X > Y$, need not be simply true or false. It can be true at some 'times' and false at others (and even undefined at others). As we have seen, semantically, the value of $X > Y$ depends on various time parameters because the values of the variables X and Y themselves depend on time parameters. As a result of this, certain properties of first order logic, such as the Deduction Theorem, fail to hold for Lucid, except in special circumstances.

Lucid also differs from first-order logic in that we wish to allow programs to compute truth values, and therefore we have to allow an "undefined" truth value, for sub-programs which do not terminate. (Since we have this undefined truth value, we can abolish the distinction between terms and formulas, logical connectives applied to non-truth-values acting as they would for the undefined value. This uniform treatment is not essential however - it merely simplifies the formal treatment.) The formal system must be able to deal with "undefined" within the logic. This means, for example, that the law of the excluded middle does not hold.

Nevertheless, the rules of inference for Lucid are almost identical to those for first-order logic.

Sections 1 to 3 of the paper are devoted to setting up the interpretations on which the semantics is based. Then in Section 4 we define the class of sets of terms that are Lucid programs. We show that every program has a unique minimal solution, or "meaning". In the rest of the paper we discuss a formal system for proving properties of programs, justifying the sort of reasoning used in the proofs given in [1]. In particular, in Section 7 we justify the 'nested proof' technique for proving things about programs with nested loops.

1. Formalism

The meanings of programs will be based on "computation structures", which in turn are defined in terms of simple structures. We first define a general notion of structure, and build on this in later sections.

1.1 Syntax

A Lucid alphabet Σ is a set containing the symbols "U", "$\exists$" and, for each natural number n, any number of n-ary operation symbols, including, for n=0 the nullary operation symbol T.

We also have at our disposal a set of variables, e.g. X,Y,Z.

The set of Σ-terms is defined as follows:

(a) every variable is a Σ-term;

(b) if G is an n-ary operation symbol in Σ and $A_1,\ldots,A_n$ are Σ-terms then $G(A_1,\ldots,A_n)$ is a Σ-term;

(c) if V is a variable and A is a Σ-term then $\exists V A$ is a Σ-term.

1.2 Semantics

If Σ is an alphabet then a Σ-structure S is a function which assigns to each symbol σ in Σ a "meaning" σ_S in such a way that U_S is a set, $\exists_S$ is a function from subsets of U_S to elements of U_S and, if G is an n-ary operation symbol, G_S is an n-ary operation on U_S.

An S-interpretation I extends S to assign to each variable V an element V_I of U_S.

If I is an S-interpretation, V is a variable and α is an element of U_S, then $I(V/\alpha)$ denotes the S-interpretation differing from I only in that it assigns α to V.

If A is a Σ-term, S a Σ-structure and I an S-interpretation then we define an element $|A|_I$ of U_S (the "meaning" of A) as follows:

(a) for variable V, $|V|_I$ is V_I

(b) for Σ-terms $A_1, A_2, \ldots, A_n$ and n-ary operation symbol G of Σ,

$$|G(A_1,\ldots,A_n)|_I \text{ is } G_S(|A_1|_I, |A_2|_I, \ldots, |A_n|_I)$$

(c) for Σ-term A and variable V

$$|\exists\, V\, A|_I = \exists_S(\{|A|_{I(V/\alpha)} : \alpha \in U_S\})$$

We say: $\models_I$ A (I <u>satisfies</u> A or A is <u>valid</u> in I) iff $|A|_I = T_S$; if Γ is a set of terms then $\models_I \Gamma$ iff $\models_I B$ for each B in Γ; $\Gamma \models_S A$ iff $\models_I \Gamma$ implies $\models_I A$ for all S-interpretations I.

2. <u>Basic results</u>

Even in general structures we can establish several useful properties.

2.1 <u>Substitution</u>

An occurrence of a variable V in a Σ-term A is <u>bound</u> if and only if the occurrence is in a sub-term of A of the form $\exists$ V B, otherwise the occurrence is <u>free</u>. If A and Q are Σ-terms and V is a variable then A(V/Q) is the term formed by replacing all free occurrences of V in A by Q. In this situation V is said to be <u>free for Q in A</u> iff this substitution does not result in a free variable in Q becoming bound in A(V/Q), i.e. iff V does not occur free in A in a subterm of the form $\exists$ W B for some variable W occurring free in Q.

<u>Lemma 1</u> For Σ-structure S, S-interpretation I, Σ-terms A and Q and variable V, if V is free for Q in A then

$$|A(V/Q)|_I = |A|_{I(V/|Q|_I)}$$

Proof The proof of the analogous result for first-order logic carries over directly. $\square$

2.2 Power structures

One way of building structures out of simpler structures is by a generalised Cartesian product.

2.2.1 For any Σ-structure S and any set X, S^X is the unique Σ-structure C such that

(a) U_C is the set of all functions from X to U_S. If $x \in X$ and $\alpha \in U_C$ we will write α_x instead of $\alpha(x)$.

(b) If G is an operation symbol in Σ and $\alpha,\beta,\ldots \in U_C$ and $x \in X$ then $(G_C(\alpha,\beta,\ldots))_x = G_S(\alpha_x,\beta_x,\ldots)$.

(c) If K is a subset of U_C and $x \in X$ then $(\exists_C(K))_x = \exists_S(\{\alpha_x : \alpha \in K\})$.

Thus S^X carries over the operations and quantifiers of S by making them work 'pointwise' on the elements of U_C. Thus all nullary operation symbols are assigned constant functions. In particular, T_C is the constant function on X with value T_S.

2.2.2 For S^X-interpretation I and $x \in X$, I_x denotes the unique S-interpretation which assigns each variable V the value $(V_I)_x$.

The following lemma establishes that every Σ-term acts "pointwise" in S^X, even those terms containing quantifiers.

Lemma 2 For any Σ-structure S, S^X-interpretation I, Σ-term A and element $x \in X$,

$$(|A|_I)_x = |A|_{I_x}.$$

<u>Proof</u> Let C be the structure S^X. The proof proceeds by structural induction on A.

(a) If A is a variable the result is immediate.

(b) If A is $G(A_1,\dots,A_n)$ for n-ary operation symbol G in Σ, and Σ-terms $A_1,\dots,A_n$, then

$$
\begin{aligned}
(|A|_I)_X &= (|G(A_1,\dots,A_n)|_I)_X \\
&= G_S((|A_1|_I)_X,\dots,(|A_n|_I)_X) \\
&= G_S(|A_1|_{I_X},\dots,|A_n|_{I_X}) \\
&= |G(A_1,\dots A_n)|_{I_X}.
\end{aligned}
$$

(c) If A is $\exists$ V B for some variable V and Σ-term B then

$$
\begin{aligned}
(|\exists\ V\ B|_I)_X &= (\exists_C(\{|B|_{I(V/\alpha)} : \alpha \in U_C\}))_X \\
&= \exists_S(\{(|B|_{I(V/\alpha)})_X : \alpha \in U_C\}) \\
&= \exists_S(\{|B|_{I_X(V/\alpha_X)} : \alpha \in U_C\})
\end{aligned}
$$

(since if $I' = I(V/\alpha)$ then $I'_X = I_X(V/\alpha_X)$)

$$
= \exists_S(\{|B|_{I_X(V/a)} : a \in U_S\})
$$

(since as α ranges over U_C, α_X ranges over U_S)

$$
= |\exists\ V\ B|_{I_X}.
$$

□

It follows that S and S^X have the same theory:

<u>Corollary 2.1</u> For any Σ-structure S, any set X, any Σ-term A and set Γ of Σ-terms, if $C = S^X$ then

$$\Gamma \models_S A \text{ iff } \Gamma \models_C A$$

<u>Proof</u> Suppose first that $\Gamma \models_S A$. Let J be a C-interpretation such that $\models_J \Gamma$. Then for any B in Γ and any x in X, $T_S = (T_C)_x = (|B|_J)_x = |B|_{J_x}$ by Lemma 2. Thus $\models_{J_x} \Gamma$ and so $\models_{J_x} A$; hence $|A|_{J_x} = T_S$. Therefore $(|A|_J)_x = |A|_{J_x} = T_S = (T_C)_x$. Since x was arbitrary, $|A|_J = T_C$ and so $\models_J A$.

Now suppose $\Gamma \models_C A$. Let I be an S-interpretation such that $\models_I \Gamma$. Define the C-interpretation J by setting $(V_J)_x = V_I$ for each x in X and each variable V, i.e. $J_x = I$ for each x. Then, for any B in Γ, $(|B|_J)_x = |B|_{J_x} = |B|_I = T_S = (T_C)_x$ for any x in X and so $\models_J \Gamma$. Therefore, $\models_J A$ and, choosing any x in X, $T_S = (|A|_J)_x = |A|_{J_x} = |A|_I$ and so $\models_I A$. $\square$

3. Models of Computation

We now build up the structures necessary to give meaning to programs. These will be power structures, based on certain elementary structures called standard structures.

We define Spec to be the set of special Lucid function symbols {first, next, as soon as, hitherto, latest, followed by}.

3.1 Standard structures

An alphabet Σ is standard if in addition to T and $\exists$ it contains the nullary operation symbols $\perp$ and F, the unary operation symbol $\neg$, the binary operation symbols $\vee$ and $=$ and the ternary operation symbol if then else, but none of the special Lucid symbols in Spec. (Σ may contain numerals 0,1 etc. as nullary operation symbols.)

A standard structure is a structure S whose alphabet is standard and such that

(a) T_S, F_S and $\perp_S$ are true, false, and undefined, respectively.

(b) $\neg_S$ yields true if its argument is false, false if its argument is true, undefined otherwise.

(c) $\vee_S$ yields true if at least one argument is true, false if both are false, undefined otherwise.

(d) $=_S$ yields true if its arguments are identical, false otherwise.

(e) if then else$_S$ yields its second argument if its first is true, its third if its first is false, undefined otherwise.

(f) for any subset K of U_S, $\exists_S(K)$ is true if true $\in$ K, false if K = {false}, undefined otherwise.

(g) all other operations of S are monotonic, for the ordering on U_S defined by $x \sqsubseteq y$ iff x = y or x = undefined. (Note then that the only non-monotonic operation is $=_S$.)

Standard structures are our basic domains of data objects and correspond most closely to ordinary first-order structures. Note that if we restrict $\neg_S$ and $\vee_S$ to true, false and undefined, they agree with the corresponding operators in the three-valued logic of Lukasiewicz.

3.2 Computation structures

Programs will use the special Lucid functions Spec, and these functions are interpreted over certain types of power structures.

3.2.1 Comp(S)

If S is a standard Σ-structure, then Comp(S) is the unique ($\Sigma \cup$ Spec)-structure C which extends S^{N^N} * to the larger alphabet as follows:

For $\alpha, \beta \in U_C$ and $\bar{t} = t_0 t_1 t_2 \ldots \in N^N$

i) $(\text{first}_C\ (\alpha))_{\bar{t}} = \alpha_{0 t_1 t_2 \ldots}$

ii) $(\text{next}_C\ (\alpha))_{\bar{t}} = \alpha_{t_0+1\ t_1 t_2 \ldots}$

iii) $(\alpha\ \text{as soon as}_C \beta)_{\bar{t}} = \alpha_{s t_1 t_2 \ldots}$ if there is a unique s such that $\beta_{s t_1 t_2 \ldots}$ is true and $\beta_{r t_1 t_2 \ldots}$ is false for all $r < s$, undefined if no such s exists.

iv) $(\text{hitherto}_C (\alpha))_{\bar{t}}$ = true if $\alpha_{s t_1 t_2 \ldots}$ is true for all $s < t_0$,
false if $\alpha_{s t_1 t_2 \ldots}$ is false for some $s < t_0$,
undefined otherwise.

v) $(\text{latest}_C (\alpha))_{\bar{t}} = \alpha_{t_1 t_2 \ldots}$.

vi) $(\alpha\ \text{followed by}_C\ \beta)_{0 t_1 t_2} = \alpha_{0 t_1 t_2 \ldots}$

$(\alpha\ \text{followed by}_C\ \beta)_{t_0+1\ t_1 t_2 \ldots} = \beta_{t_0 t_1 t_2 \ldots}$

* N is the set of natural numbers and N^N is the set of functions from N to N i.e. the set of infinite sequences of natural numbers.

Note that all other operations are pointwise extensions of the corresponding operations in S.

3.2.2 The function latest is used to formalise nested loops. If we have no nested loops, a simpler structure suffices.

Loop(S)

If Σ and S are as above and Σ' is the alphabet of Comp(S), omitting latest, then Loop(S) is the unique Σ'-structure C' which extends S^{N} to Σ' in such a way that $\text{first}_{C'}$, $\text{next}_{C'}$ etc. are defined as for Comp(S), but with $t_1 t_2 \ldots$ omitted. For example

$$(\text{first}_{C'}(\alpha))_{t_0} = \alpha_0 \text{ and } (\text{next}_{C'}(\alpha))_{t_0} = \alpha_{t_0+1}.$$

The usefulness of Loop(S) lies in the fact that Loop(S) is easier to understand and Loop(S) and Comp(S) have the same theory for terms not involving latest:

Theorem 1 For any standard structure S and any term A and set of terms Γ all in the language of Loop(S),

$$\Gamma \models_{\text{Comp}(S)} A \text{ iff } \Gamma \models_{\text{Loop}(S)} A$$

Proof Let C' be the restriction of Comp(S) to the language of Loop(S).

It is easily verified that C' is isomorphic to $\text{Loop}(S)^{N^N}$ and so the result follows from Corollary 2.1. $\Box$

3.2.3 Note that if S is a standard structure and C is an extension of S^X for some set X, then $=_C$ is not the identity relation on C. Nevertheless $\models_C A = B$ iff $|A|_C$ and $|B|_C$ are identical.

3.2.4 Quiescence and constancy

Let $C = \text{Comp}(S)$ and $\alpha \in U_C$. Then α is a function from infinite sequences $t_0t_1t_2\ldots$ of natural numbers to U_S. If α is the value of a variable V, then, intuitively, the value of V depends on the time parameters $t_0t_1t_2\ldots$, where t_0 is the number of iterations of the loop defining V, t_1 is the number of iterations of the next outer loop, and so on. If $\alpha_{\bar{t}}$ is independent of the first element of $\bar{t}$ (i.e. $\alpha_{t_0t_1t_2\ldots} = \alpha_{0t_1t_2\ldots}$ for all t_0) then we say α is quiescent. A term A is quiescent (in C) if $\models_C$ A = first A. Note that for terms A and B, first A, latest A and A as soon as B are all quiescent.

If $\alpha_{\bar{t}}$ is independent of $\bar{t}$, then α is said to be constant. Note that G_C is constant for any nullary operation symbol G.

In Loop(S) we can use the same definitions, but then there is no difference between quiescence and constancy.

4. Programs

We impose minimal syntactic restrictions on programs, to simplify the formal treatment. In practice, other restrictions would probably be required.

4.1 Syntax

A Σ-program P is a set of $(\Sigma \cup \text{Spec})$-terms such that

(a) each element of P is an equation of the form $\phi = \psi$, where ψ is a quantifier-free term having no occurrences of =, and ϕ is of the form X, first X, next X or latest X for some variable X.

(b) The variable input may not occur on the left hand side of any equation in P.

(c) Every other variable X occurring in P, when appearing on the left hand side of an equation, may only do so as part of a definition of X. X must be defined exactly once, in one of the following ways:

directly	i.e. $X = \psi_1$
indirectly	i.e. latest $X = \psi_2$
iteratively	i.e. first $X = \psi_3$
	next $X = \psi_4$.

In the above, the terms ψ_2 and ψ_3 must be syntactically quiescent in P, a property which is defined as follows:

(i) first ϕ, latest ϕ and ϕ as soon as ψ are syntactically quiescent in P.

(ii) if $\phi_1, \phi_2, \ldots, \phi_n$ are syntactically quiescent in P and G is an n-ary operation symbol in Σ, then $G(\phi_1, \phi_2, \ldots, \phi_n)$ is syntactically quiescent in P.

(iii) if $Y = \phi$ is in P and ϕ is syntactically quiescent in P, then Y is syntactically quiescent in P.

4.2 Semantics

The meanings of programs are specified by Comp(S)-interpretations, where S is the standard structure corresponding to the domain of data.

4.2.1 Solutions

For any Σ-program P and standard Σ-structure S, if $C = \mathrm{Comp}(S)$ and α is an element of U_C then a (S,α)-solution of P is a C-interpretation I such that $\mathrm{input}_I = \alpha$ and $\models_I$ P.

4.2.2 Theorem 2

For any Σ-program P and standard Σ-structure S, if $C = \mathrm{Comp}(S)$ and $\alpha \in U_C$ then there is a (S,α)-solution I of P that is minimal, i.e. for any (S,α)-solution I' of P, for all $\bar{t} \in N^N$ and all variables V in Σ, $(V_I)_{\bar{t}} \sqsubseteq (V_{I'})_{\bar{t}}$.

Proof (sketch)

The first step is to transform P into a set of simple equations. This is done by replacing each pair of equations of the form first X = ϕ, next X = ϕ' by the single simple equation X = ϕ followed by ϕ', and replacing each equation of the form latest X = ϕ by the simple equation X = $\mathrm{latest}^{-1}\phi$. The operation latest_C^{-1} is defined by $(\mathrm{latest}_C^{-1}(\alpha))_{t_0 t_1 \ldots} = \alpha_{0 t_0 t_1 \ldots}$

This transforms the program P into a 'program' P' of the form $\bar{X} = \tau(\bar{X})$, where $\bar{X}$ is the vector of all the variables in P other than input.

We now note that the 'programs' P and P' have the same solutions. That every solution of the original program P is a solution of the transformed program P' is clear, and the converse follows from the quiescence restrictions on P, as follows.

Suppose that $\models_I$ $X = \phi$ followed by ϕ'. Then by the definition of followed by, $\models_I$ first X = first ϕ and $\models_I$ next $X = \phi'$. But if $X = \phi$ followed by ϕ' in P' came from first $X = \phi$ and next $X = \phi'$ in P, then the syntactic quiescence of ϕ ensures that $\models_I$ ϕ = first ϕ, so $\models_I$ first $X = \phi$. Similarly, $\models_I$ X = latest^{-1} ϕ implies $\models_I$ latest X = first ϕ, and so by syntactic quiescence $\models_I$ latest $X = \phi$.

Now we note that the ordering on U_C given in the statement of the theorem makes U_C into a cpo (complete partial order), and it is easily verified that all the operations in C that are used in the 'term' τ are continuous. Moreover, although $=_C$ is not equality on U_C, by 3.2.3 the solutions of P' are fixpoints of τ.

Therefore, the transformed program P' has a unique minimal (S,α)-solution I, and hence so does P. (In fact $\bar{X}_I = \bigsqcup_{i=0}^{\infty} |\tau^i(\bar{I})|_I$; see, e.g.,[5].) □

4.3 Syntactic Enrichment

To facilitate the writing of programs we introduce 'nesting' in programs, as a syntactic abbreviation. We say that the expression

```
begin
    $\phi_1$
    $\phi_2$
    ⋮
    $\phi_n$
end
```

is shorthand for the set of terms

$$\phi_1'$$
$$\phi_2'$$
$$\vdots$$
$$\phi_n' \quad ,$$

where ϕ_i' is obtained from ϕ_i by replacing each 'global' variable V by latest V. A global variable is one which occurs within the rest of the program enclosing the original begin ... end expression. The symbols begin and end are used to delimit inner loops, and the formulation using latest shows that within inner loops global variables become quiescent. Loops can be nested to any depth. Note that for a program using begin ... end to be legal, the result of removing the begin ... end's must be a legal program according to 4.1.

5. Axioms

We now describe the formal system of axioms and rules of inference used for proving properties of Lucid programs.

5.1 The following abbreviations will be used in the rest of the paper:

$A \wedge B$ means $\neg(\neg A \vee \neg B)$

$A \rightarrow B$ means $\neg(A = T) \vee B$

$\forall V\, A$ means $\neg \exists V \neg A$

Note that, in standard structures, $\wedge$ agrees with the three-valued logic of Lukasiewicz, but $\rightarrow$ does not. In particular, we have $\models \bot \rightarrow F$, but in standard three-valued logic $\bot \rightarrow F$ would be $\bot$. This difference is crucial, and allows, for example, the use of the deduction theorem in standard structures. (However, $\rightarrow$ is defined in terms of $=$, and therefore it may not be used in programs.)

5.2 Parentheses will be (and have been) dropped from terms by using the following ranking of priorities for operators (from highest to lowest):

first, next, latest, hitherto, $\neg$, $\wedge$, $\vee$, if then else, as soon as, followed by, $=$, $\rightarrow$.

Note the low priority of as soon as, and the even lower priorities of $=$ and $\rightarrow$. Thus $A \rightarrow B = C$ as soon as $D \wedge E$ means $A \rightarrow (B = (C$ as soon as $(D \wedge E)))$.

5.3 *Theorem 3* The following are valid in Comp(S) for any standard Σ-structure S, and $(\Sigma \cup \mathrm{Spec})$-terms X, Y and P

(a) $(X = Y) \vee \neg(X = Y)$

(b) $((A = T) = (\neg\neg A = T)) \wedge ((A = F) = (\neg A = T))$

(c) $(\mathbf{first}\ \mathbf{first}\ X = \mathbf{first}\ X) \wedge (\mathbf{next}\ \mathbf{first}\ X = \mathbf{first}\ X)$

(d) $(\mathbf{first}(X\ \mathbf{followed\ by}\ Y) = \mathbf{first}\ X) \wedge (\mathbf{next}(X\ \mathbf{followed\ by}\ Y) = Y)$

(e) $(\mathbf{first}\ \mathbf{hitherto}\ P = T) \wedge (\mathbf{next}\ \mathbf{hitherto}\ P = P \wedge \mathbf{hitherto}\ P)$

(f) $X\ \mathbf{as\ soon\ as}\ P = \underline{\mathrm{if}}\ \mathbf{first}\ P\ \underline{\mathrm{then}}\ \mathbf{first}\ X\ \underline{\mathrm{else}}\,(\mathbf{next}\ X\ \mathbf{as\ soon\ as}\ \mathbf{next}\ P)$

(g) $X\ \mathbf{as\ soon\ as}\ P = X\ \mathbf{as\ soon\ as}\ P \wedge \mathbf{hitherto}\ \neg P$

(h) $\mathbf{first}(X\ \mathbf{as\ soon\ as}\ P) = X\ \mathbf{as\ soon\ as}\ P$

(i) $P \wedge \mathbf{hitherto}\ \neg P \rightarrow (X\ \mathbf{as\ soon\ as}\ P) = X$

(j) $T\ \mathbf{as\ soon\ as}\ P \rightarrow\ \mathbf{first}\ X\ \mathbf{as\ soon\ as}\ P = \mathbf{first}\ X$

(k) $(\underline{\mathrm{if}}\ T\ \underline{\mathrm{then}}\ X\ \underline{\mathrm{else}}\ Y = X) \wedge (\underline{\mathrm{if}}\ F\ \underline{\mathrm{then}}\ X\ \underline{\mathrm{else}}\ Y = Y)$

Proof These results (for *variables* X, Y and P) are easily verified in Loop(S) and carry over to Comp(S) by Theorem 1. The variables can then be replaced by $(\Sigma \cup \mathrm{Spec})$-terms. □

If we define **eventually** P to be T **as soon as** P (with the same priority as **as soon as**), we have the following corollary.

Corollary 3.1 With S, X and P as above, the following are valid in Comp(S):

(a) $\mathbf{eventually}\ P \rightarrow\ \mathbf{first}\ X\ \mathbf{as\ soon\ as}\ P = \mathbf{first}\ X$

(b) $\mathbf{eventually}\ P = \mathbf{eventually}\ P \wedge \mathbf{hitherto}\ \neg P$

(c) $\mathbf{eventually}\ P = \underline{\mathrm{if}}\ \mathbf{first}\ P\ \underline{\mathrm{then}}\ T\ \underline{\mathrm{else}}\ \mathbf{eventually}\ \mathbf{next}\ P.$

Proof These follow from the axioms of Theorem 3. □

5.4 The next theorem justifies 'pushing' **first** and **next** past quantifiers and non-Lucid operations.

Theorem 4 For any standard Σ-structure S and any Σ-term A in which $X_1, X_2, \ldots, X_k$ are the variables occurring freely:

(a) $\text{first}\ A = A(X_1/\text{first}\ X_1, X_2/\text{first}\ X_2, \ldots)$ is valid in Comp(S),

along with corresponding equations for next and latest.

(b) $\text{eventually}\ P \rightarrow A\ \text{as soon as}\ P = A(X_1/X_1\ \text{as soon as}\ P,\ X_2/X_2\ \text{as soon as}\ P, \ldots$

is valid in Comp(S).

Proof We will consider only Loop(S). The results carry over to Comp(S) by Theorem 1. Let I be a Loop(S)-interpretation and let t be any natural number. Then, if $\bar{X}$ denotes $X_1, X_2, \ldots, X_k$,

$$(|\text{first}\ A|_I)_t = (|A|_I)_0$$

$$= |A|_{I_0} \text{ (by Lemma 2) } = |A|_{I_0(\bar{X}/(\bar{X}_I)_0)}$$

$$= |A|_{I_0(\bar{X}/(|\text{first}\ \bar{X}|_I)_t)} = |A|_{I_t(\bar{X}/(|\text{first}\ \bar{X}|_I)_t)}$$

(since A has no other free variables)

$$= (|A|_{I(\bar{X}/|\text{first}\ \bar{X}|_I)})_t \text{ (by Lemma 2) } = (|A(\bar{X}/\text{first}\ \bar{X})|_I)_t$$

(by Lemma 1).

Similar reasoning verifies the other results. $\square$

6. Rules of Inference

Lucid cannot be a complete formal system because the Lucid functions are powerful enough to characterise unsolvable problems that are not even partially decidable. All we can do is add to Lucid whatever axioms and rules of inference seem natural and useful. In this section we give rules of inference for the logical connectives, and useful rules for the special Lucid functions. The 'logical' rules of inference are those of a simple natural deduction system (see, for example [4]).

6.1 Natural Deduction Rules

6.1.1 Theorem 5 The following rules are valid for standard Σ-structure S, Σ-terms A,B,C,D,P,Q, finite sets Γ and Δ of Σ-terms and variable V, provided V does not occur freely in Γ or D, and is free for P and Q in A:

($\wedge$I)	$A, B \models_S A \wedge B$	($\wedge$E)	$A \wedge B \models_S A$
			$A \wedge B \models_S B$
($\vee$I)	$A \models_S A \vee B$	($\vee$E)	$A \to C,\ B \to C, A \vee B \models_S C$
	$B \models_S A \vee B$		
(FI)	$A, \neg A \models_S F$	(FE)	$F \models_S B$
($\to$I)	if $\Delta, A \models_S B$ then $\Delta \models_S A \to B$	($\to$E)	$A \to B, A \models_S B$
($\forall$I)	if $\Gamma \models_S A$ then $\Gamma \models_S \forall V A$	($\forall$E)	$\forall V A \models_S A(V/Q)$
($\exists$I)	$A(V/Q) \models_S \exists V A$	($\exists$E)	if $\Gamma \models_S A \to D$ then $\Gamma, \exists V A \models_S D$
(=I)	$\models_S V = V$	(=E)	$A(V/P), P = Q \models_S A(V/Q)$.
(TI)	$A \models_S A = T$	(TE)	$A = T \models_S A$

Proof The validity of the rules can be established by straightforward calculation from the definitions. $\square$

There are no rules for $\neg$ because we do not have the law of the excluded middle: $A \vee \neg A$ is not valid in general, because A may not be truth-valued. This means that some of the tautologies and derived rules of first-order logic are not valid in standard structures. For example $(A \to B) \to \neg A \vee B$ is not valid, and if we were to define $A \leftrightarrow B$ to mean $(A \to B) \wedge (B \to A)$, then we would not have substitutivity of $\leftrightarrow$ (note, for example, that $\perp \leftrightarrow F$).

6.1.2 Most of the rules of Theorem 5 hold also for Comp(S):

<u>Theorem 6</u> All the rules of Theorem 5, except $(\to I)$, are valid for C in place of S (where C is Comp(S)), and $\Sigma \cup$ Spec in place of Σ.

<u>Proof</u> Apart from the quantifier rules, and $(\to I)$, all rules are of the form $\phi \models \psi$ and carry over directly because of the point wise definition of the connectives. We illustrate this for the $(\vee E)$ rule. Consider any C-interpretation I for which $\models_I A \to C$, $\models_I B \to C$ and $\models_I A \vee B$. Then, for all $\bar{t} \in N^N$, $(|A \to C|_I)_{\bar{t}}$, $(|B \to C|_I)_{\bar{t}}$ and $(|A \vee B|_I)_{\bar{t}}$ are all <u>true</u>. By definition of C, we then have $(|A|_I)_{\bar{t}} \to_S (|C|_I)_{\bar{t}}$, $(|B|_I)_{\bar{t}} \to_S (|C|_I)_{\bar{t}}$ and $(|A|_I)_{\bar{t}} \vee_S (|B|_I)_{\bar{t}}$ are all <u>true</u>. By the $(\vee E)$ rule for S (Theorem 1) we then have $(|C|_I)_{\bar{t}} =$ <u>true</u>. This holds for all $\bar{t} \in N^N$, so $\models_I C$.

We illustrate the proof for the quantifier rules by considering $(\forall E)$ and $(\exists E)$.

<u>$(\forall E)$</u>: Let I be any C-interpretation for which $|\forall\ V\ A|_I = T_I$. Then for all $\bar{t} \in N^N$

$$\underline{\text{true}} = (|\forall\ V\ A|_I)_{\bar{t}}$$

$$= \forall_S\{(|A|_{I(V/\alpha)})_{\bar{t}} : \alpha \in U_C\}.$$

Therefore for all $\bar{t} \in N^N$ and all $\alpha \in U_C$ we have $(|A|_{I(V/\alpha)})_{\bar{t}} =$ <u>true</u>. Now $|A|_{I(V/|Q|_I)} = |A(V/Q)|_I$, by Lemma 1, and so, for all $\bar{t} \in N^N$, $(|A(V/Q)|_I)_{\bar{t}} =$ <u>true</u>, that is $\models_I A(V/Q)$.

<u>$(\exists E)$</u>: Assume $\Gamma \models_C A \to D$ and consider any C-interpretation I for which $\models_I B$, for all $B \in \Gamma$, and $\models_I \exists\ V\ A$. Consider any $\bar{t} \in N^N$. By the definition of $\exists_C$, there is some $\alpha \in U_C$ such that $(|A|_{I(V/\alpha)})_{\bar{t}} =$ <u>true</u>. Now $I(V/\alpha)$ is a C-interpretation and $\models_{I(V/\alpha)} \Gamma$, since V is not free in Γ. Thus $\models_{I(V/\alpha)} A \to D$, and so $(|D|_{I(V/\alpha)})_{\bar{t}} =$ <u>true</u>. Since V is not free in D, we then have $(|D|_I)_{\bar{t}}$. We chose $\bar{t}$ arbitrarily, so $\models_I D$. $\square$

6.2 Lucid Rules

6.2.1 One of the most important rules is that a standard Σ-structure S and Comp(S) have the same theory, when restricted to Σ-terms, so any "elementary" property can be used directly in any proof about a program.

Theorem 7 For any standard Σ-structure S, any Σ-term A and any set Γ of Σ-terms, $\Gamma \models_S A$ iff $\Gamma \models_C A$, where $C = \text{Comp}(S)$.

Proof Since Γ and A are in the language of S and since Comp(S) is an extension of S^N the result follows immediately from Corollary 2.1. □

6.2.2 Other Lucid rules including induction and termination are given by the following theorem.

Theorem 8 For any standard Σ-structure S, if $C = \text{Comp}(S)$ then

(a) $P \models_C$ first P and $P \models_C$ next P

(b) first P, $P \rightarrow$ next $P \models_C P$ (Induction)

(c) $P \wedge$ hitherto $\neg P \rightarrow$ first Q, eventually $P \models_C$ first Q

(d) $P \rightarrow \neg$ hitherto $(P = F) \models_C X$ as soon as $P = \bot$

(e) integer Y, $Y >$ next $Y \models_C$ eventually $Y \leq 0$ (Termination)

(f) $X =$ next $X \models_C X =$ first X

where in (e) we assume S includes the integers.

Proof By calculation from the definitions. □

6.3 Recovering the Deduction Theorem

We have seen that the ($\rightarrow$I) rule is not valid in Comp(S). However, we can recover this rule, at the expense of weakening the (=E) rule, by a form of reasoning which intuitively corresponds to confining oneself to a particular moment during the execution of a program.

6.3.1 Definition of $|\approx$

If S is a Σ-structure, C = Comp(S) and A is a term and Γ a set of terms on the alphabet of C, then we define $\Gamma \;|\!\approx_C A$ to mean that for any C-interpretation I, if $(|B|_I)_{\bar{t}}$ = true for every B in Γ, then $(|A|_I)_{\bar{t}}$ = true.

Thus $\Gamma \;|\!\approx_C A$ means that, at any time, if all the terms in Γ are true, then A is true. It is immediate that $\models_C A$ implies $|\!\approx_C A$, and that $\Gamma \;|\!\approx_C A$ implies $\Gamma \models_C A$ but not vice versa, e.g. P $\models_C$ next P but not P $|\!\approx_C$ next P.

6.3.2 Theorem 9 For any standard Σ-structure S, if C = Comp(S)

(a) every rule of Theorem 5 except the (=E) rule remains valid if $\models_S$ is replaced by $|\!\approx_C$.

(b) for A,P,Q and V as in Theorem 5, if A contains no Lucid functions then A(V/P), P = Q $|\!\approx_C$ A(V/Q).

(c) Theorem 7 is valid for $|\!\approx_C$ in place of $|\!=_C$.

Proof Let Δ,A,D and V be as in Theorem 5.

(a) We will illustrate the proof by considering the ($\exists$E) rule and the ($\rightarrow$I) rule

(i) Assume $\Gamma \;|\!\approx_C A \rightarrow D$ and $\Gamma \;|\!\approx_C \exists$ V A. Let $\bar{t} \in N^N$ and let I be a C-interpretation such that $(|B|_I)_{\bar{t}}$ for every B in Γ. Then by the second assumption $(|\exists \text{ V } A|_I)_{\bar{t}}$ = true and so $(|A|_{I(V/\alpha)})_{\bar{t}}$ = true for some α in U_C by the definition of $\exists_C$. Since V does not occur in any B in Γ, $(|B|_{I(V/\alpha)})_{\bar{t}} = (|B|_I)_{\bar{t}}$ = true for any such B, and so by the first assumption $(|D|_I)_{\bar{t}}$ = true. Therefore $\Gamma \;|\!\approx_C$ D.

(ii) Let $\bar{t} \in N^N$ and suppose that every C-interpretation which makes A and everything in Γ true at $\bar{t}$ also makes B true at $\bar{t}$. Suppose now that C-interpretation I makes everything in Γ true at $\bar{t}$. If I makes A true at $\bar{t}$ then it must make B true at $\bar{t}$ and so makes

$A \rightarrow B$ true at $\bar{t}$. On the other hand, if I makes A other than true at $\bar{t}$ then $A \rightarrow B$ will be true at $\bar{t}$ regardless of the value I assigns B at $\bar{t}$. In either case $A \rightarrow B$ is true at $\bar{t}$ and so $\Gamma \mid\approx_C A \rightarrow B$.

(b) Suppose that $(|A(V/P)|_I)_{\bar{t}}$ = true and $(|P{=}Q|_I)_{\bar{t}}$ = true. Now $|A(V/P)|_I = |A|_{I(V/|P|_I)}$ by Lemma 1 and $(|A|_{I(V/|P|_I)})_{\bar{t}} =$ $|A|_{I_{\bar{t}}(V/(|P|_I)_{\bar{t}})}$ since A contains no Lucid functions. But $(|P{=}Q|_I)_{\bar{t}}$ = true implies $(|P|_I)_{\bar{t}} = (|Q|_I)_{\bar{t}}$. Thus

$$|A|_{I_{\bar{t}}(V/(|P|_I)_{\bar{t}})} = |A|_{I_{\bar{t}}(V/(|Q|_I)_{\bar{t}})}$$
$$= (|A|_{I(V/|Q|_I)})_{\bar{t}}$$
$$= (|A(V/Q)|_I)_{\bar{t}} .$$

Therefore $(|A(V/Q)|_I)_{\bar{t}}$ = true.

(c) Assume $\Gamma \mid=_S A$ and let I be a C-interpretation. For any $\bar{t} \in N^N$, if $(|B|_I)_{\bar{t}}$ is true for all B in Γ, then by Lemma 2 $|B|_{I_{\bar{t}}}$ is true for all B in Γ; i.e. $\mid=_{I_{\bar{t}}} \Gamma$. Hence $\mid=_{I_{\bar{t}}} A$, and so $(|A|_I)_{\bar{t}}$. Conversely, assume $\Gamma \mid\approx_C A$. Therefore, $\Gamma \mid=_C A$, and hence $\Gamma \mid=_S A$, by Theorem 7. $\square$

We call the rule in Theorem 9(b) the (weak =E) rule. To illustrate that (=E) does not work for $\mid\approx$, note that next P, $P = Q \mid\approx$ next Q is not valid (informally, if P equals Q at some time when P will be true at the next step, it does not necessarily follow that Q will be true at the next step).

We use $\mid\approx$ in the following way. Suppose we wish to prove $\Gamma \mid=_C A \rightarrow B$. We assume Γ and A, and try to prove B using only axioms and Theorem 7 and the natura deduction rules of Theorem 5, but with the (weak =E) rule instead of the (=E) rule. (We may not use Theorem 8.) If we manage to do this we have $\Gamma, A \mid\!\approx_C B$ and we can use $(\rightarrow I)$ to get $\Gamma \mid\!\approx_C A \rightarrow B$. Thus $\Gamma \models_C A \rightarrow B$. We see that to use the deduction theorem we must not use any of the Lucid rules in Theorem 8, and use only the weak version of the (=E) rule.

6.3.3 There is another way in which we can regain the deduction theorem. If we are reasoning about a simple loop, and we have made an assumption that is quiescent, then the assumption can be cancelled:

Theorem 10 For any Σ-structure S, if C = Comp(S) and A and B are terms and Γ a set of terms on the alphabet of C omitting latest, then

$$\Gamma, \text{first } A \models_C B \text{ implies } \Gamma \models_C \text{first } A \rightarrow B$$

Proof The theorem holds for Loop(S) in place of Comp(S), because if first A is ever true it is always true. The result carries over to Comp(S) by Theorem 1. □

7. Proofs within Loops

The structuring of programs that is made possible by the use of begin and end also allows "structured proofs". We will show that

(i) Within a begin .. end loop, all the rules of inference are valid and so is the assumption that $X = \text{first } X$ for every global variable X. Anything that follows by introducing latest also follows without latest, in this fashion.

(ii) Any assertion about the globals of a begin ... end loop, that does not use Lucid functions, can be moved into or out of the loop.

Theorem 11 For any standard Σ-structure S, if $C = \text{Comp}(S)$, then for any term A and set of terms Γ in the alphabet of C, and any finite set of variables $\bar{X}$,

(a) $\bar{X} = \text{first } \bar{X},\ \Gamma \models_C A$ iff

$$\Gamma(\bar{X}/\text{latest } \bar{X}) \models_C A(\bar{X}/\text{latest } \bar{X}).$$

(b) If A is a Σ-term and $\bar{X}$ is the set of variables occurring freely in A, then

$\Gamma \models_C A$ iff $\Gamma \models_C A(\bar{X}/\text{latest } \bar{X})$.

Proof (a) Assume $\bar{X} = \text{first } \bar{X},\ \Gamma \models_C A$, and that, for C-interpretation I, $\models_I \Gamma(\bar{X}/\text{latest } \bar{X})$. Let $\bar{\alpha}$ be $|\text{latest } \bar{X}|_I$ and $I' = I(\bar{X}/\bar{\alpha})$. Then $\models_{I'} \bar{X} = \text{first } \bar{X}$ and $\models_{I'} \Gamma$, therefore $\models_{I'} A$, and so $\models_I A(\bar{X}/\text{latest } \bar{X})$.

Conversely, assume that $\Gamma(\bar{X}/\text{latest } \bar{X}) \models_C A(\bar{X}/\text{latest } \bar{X})$ and that for C-interpretation I, $\models_I \bar{X} = \text{first } \bar{X}$ and $\models_I \Gamma$. Then $|\bar{X}|_I$ is $\text{latest}_C\ \bar{\alpha}$ for some $\bar{\alpha}$ in U_C.* Let I' be $I(\bar{X}/\bar{\alpha})$. Then $\models_{I'} \Gamma(\bar{X}/\text{latest } \bar{X})$, and so $\models_{I'} A(\bar{X}/\text{latest } \bar{X})$. Hence $\models_I A$.

(b) Let I be a C-interpretation such that $\models_I \Gamma$. Then since $(|\text{latest } A|_I)_{t_0 t_1 t_2 \ldots} = (|A|_I)_{t_1 t_2 \ldots}$, $(|\text{latest } A|_I)_{\bar{t}} = \text{true}$ for all $\bar{t}$ iff $(|A|_I)_{\bar{t}} = \text{true}$ for all $\bar{t}$. Then since $\models_I \text{latest } A = A(\bar{X}/\text{latest } \bar{X})$ by Theorem 4(a), the result follows. □

* In fact $\bar{\alpha} = |\text{latest}^{-1}\ \bar{X}|_I$ (see the proof of Theorem 2, Section 4.2.2).

The theorem justifies (i) and (ii) above as follows. Consider the program Prime again.

```
     ⎧ N = first input
     ⎪ first I = 2
θ    ⎨ next I = I+1
     ⎩ output = ¬IdivN  as soon as  IdivN ∨ I×I≥N
       begin
          ⎧ first multiple = I×I
          ⎪ next multiple = multiple + I
     Γ    ⎨ first J = I
          ⎪ next J = J+1
          ⎩ IdivN = multiple eq N  as soon as  multiple ≥ N
       end
```

Prime is actually equivalent to Prime':

```
     ⎧ N = first input
     ⎪ first I = 2
θ    ⎨ next I = I+1
     ⎩ output = IdivN  as soon as  IdivN ∨ I×I≥N
     ⎧ first multiple = latest I × latest I
     ⎪ next multiple = multiple + latest I
Γ'   ⎨ first J = latest I
     ⎪ next J = J+1
     ⎩ IdivN = multiple eq latest N as soon as multiple ≥ latest N
```

For program Prime' it is possible to prove that

$$\text{IdivN} = \exists K\ 2 \le K < N \wedge I \times K = N.$$

In the introduction, a 'nested' proof of this, using Prime, proceeded by the following steps. First we proved, inside the inner loop, that multiple = $I \times J$. Then, still inside the loop, we used this to prove that $IdivN = \exists K\ 2 \le K < N \wedge I \times K = N$. For this we needed that $I \ge 0$. This had to be proved in the outer loop, and could then be brought inside the inner loop, for use in the proof, because it is a statement not involving Lucid functions. Finally, the statement $IdivN = \exists K\ 2 \le K < N \wedge I \times K = N$ could be brought out of the inner loop because it doesn't use Lucid functions, and its free variables are all globals of the inner loop.

Formally we have $\theta \models I \ge 0$ and $\Gamma, I \ge 0,\ I = \mathbf{latest}\ I,$ $N = \mathbf{latest}\ N \models IdivN = \exists K\ 2 \le K < N \wedge I \times K = N$. From these we establish $\theta, \Gamma' \models \exists K\ 2 \le k < N \wedge I \times K = N$ as follows:

By Theorem 11a) we have

$$\Gamma',\ \mathbf{latest}\ I \ge 0 \models \mathbf{latest}\ IdivN = \exists K\ 2 \le K < \text{latest}\ N \wedge \mathbf{latest}\ I \times K = \mathbf{latest}\ N.$$

But $\theta \models I \ge 0$, and so by Theorem 11b) we have $\theta \models \mathbf{latest}\ I \ge 0$. Therefore,

$$\theta, \Gamma' \models \mathbf{latest}\ IdivN = \exists K\ 2 \le K < \mathbf{latest}\ N \wedge \mathbf{latest}\ I \times K = \mathbf{latest}\ N.$$

Again using Theorem 11b) we get

$$\theta, \Gamma' \models IdivN = \exists K\ 2 \le K < N \wedge I \times K = N.$$

This same sort of reasoning can be extended for loops nested to arbitrary depth.

8. References

[1] E.A. Ashcroft and W.W. Wadge, "Demystifying Program Proving: An Informal Introduction to Lucid", to appear in the Communications of the A.C.M.

[2] E.A. Ashcroft and W.W. Wadge, "Program Proving Without Tears", Proc. Intl. Symp. on Proving and Improving Programs, Arc et Senans, July 1975.

[3] R. Burstall, "Program Proving as Hand Simulation with a Little Induction", Proceedings IFIP Congress 1974, Stockholm.

[4] Z. Manna, "Introduction to Mathematical Theory of Computation", McGraw Hill, New York, 1974.

[5] R. Milner, "Models of LCF", Memo AIM/CS 332, Stanford (1973).