The University of Warwick

THEORY OF COMPUTATION

REPORT NO.33

STRUCTURED LUCID

BY

E. ASHCROFT AND W. WADGE

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

Revised
March 1980

STRUCTURED LUCID

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

and

Bill Wadge
Department of Computer Science
University of Warwick
Coventry, England

June 1979, Revised March 1980

CS-79-21

This work was supported by the National Research Council of Canada and the Science Research Council of the United Kingdom.

STRUCTURED LUCID

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

and

Bill Wadge
Department of Computer Science
University of Warwick
Coventry, England

Abstract

Structured Lucid is an ISWIM-like language formed by combining Basic Luid and USWIM. Structured Lucid is almost a member of the USWIM family, except that a distinction is made between two kinds of function variables: those which denote functions mapping streams (histories) of data objects to streams of data objects, and those which denote streams of functions mapping data objects to data objects. The inference and manipulation rules of Structured Lucid combine Basic Lucid's facility for implicit reasoning about time with the modular or nested reasoning of USWIM.

The distinction between stream functions and streams of functions is inherent in the ideas of Basic Lucid and has a natural operational interpretation. On one hand, a module which computes a stream of functions can be thought of a defining a subcomputation which is carried out while the main computation 'waits' or is 'frozen'; on the other hand, some modules which compute stream functions can be thought of as defining separate computations carried out in parallel

with the main computation (a form of coroutine). In general, though, a module may freeze some but not all of its parameters or globals, and for these various combinations of the two interpretations are appropriate.

0. INTRODUCTION

In [3] we introduced a family of logical programming languages called USWIM, which was based on Landin's ISWIM [5]. USWIM was designed to be semantically simple and yet provide a general way of allowing scope and defined functions in programming languages. A particular language is obtained by supplying an algebra A consisting of a data domain and operations on that domain. The particular language is then called USWIM(A).

Basic Lucid, as described in [2], is a general way of allowing iteration in a logical or mathematical way. It is natural to try to combine these two approaches in a single language (or family of languages), so that the resulting language provides iteration, defined functions and scope of variables in a logical, semantically simple way. The result of doing this we shall call Lucid (or, strictly speaking, Lucid(A)).

There are several similarities between USWIM and Basic Lucid. USWIM (like ISWIM before it) is a general way of allowing structured, modularised recursion and recursive definitions, whereas Basic Lucid is a general way of allowing iteration. In both cases the languages are mathematically simple, denotationally specified and referentially transparent. In fact both languages are based on equations, and they both depend upon a given data algebra A. USWIM forms constructs from the equations (phrases, function definitions) whereas Basic Lucid stays with simple equations but works with a modified data algebra $Lu(A)$, the universe of which consists of infinite sequences of elements from the universe of

$\hat{A}$, and the operations of which are the "pointwise extensions" of the operations of $\hat{A}$, together with some new "Lucid operations". Thus USWIM and Lucid achieve their goals in somewhat 'orthogonal' ways and it should be possible to combine the languages quite cleanly.

One simple way of combining them is to consider the language USWIM(Lu($\hat{A}$)) , which we will call ULU($\hat{A}$) for simplicity. ULU is a perfectly well-defined family of languages which has an interesting operational interpretation: the iterations within phrases are synchronized with the iterations in enclosing phrases, which results in the language having the flavour of a data-flow language,with the defined functions behaving like coroutines. Unfortunately, this synchronization means that in ULU it is not possible to define subcomputations.

Another way of combining them is to extend the meaning of USWIM phrases in a pointwise fashion. The resulting language, which we call LUSWIM is, in many ways, a conventional Algol-like language but not a member of the USWIM family because the same variable means different things inside and outside a phrase, the inner occurrence being "frozen". As a result, phrases can be interpreted as defining subcomputations which return a result.

Lucid is the result of combining ULU and LUSWIM, giving a language which is a superset of both sublanguages and is generally better than either of them individually. The features of the two sublanguages do interact, but constructively, not destructively. This is a direct result of the fact that both sublanguages are mathematically defined.

We should point out that, because Lucid is based on USWIM, it is not a "higher-type" language, that is, the defined functions can neither take functions as arguments nor return functions as results. To remove this restriction in Lucid would first require the investigation of the consequences of relaxing the restrictions in USWIM that the variable result and the formal parameters of a function definition be individual variables. These consequences may not be too dire, but some complications are almost bound to result, so this extension of Lucid is not considered in this paper.

1. THE LANGUAGE ULU(A)

Given a continuous algebra A, $Lu(A)$ is the continuous algebra of A-valued 'histories' together with the Lucid operations and the pointwise extensions of the operations of A; a formal definition appears in Section 3. When confusion might occur, we will say "data functions" when we mean functions on A, and "history functions" when we mean functions on $Lu(A)$. ($Lu(A)$ is similar to $Loop(S)$ in [1]. The 'histories' are simple infinite sequences, and the Lucid operation latest is not used.)

The language $ULU(A)$ is simply $USWIM(Lu(A))$. (It is assumed throughout this paper that the reader is familiar with both [3] and [2].)

Here is a sample program† in $ULU(Q)$, where Q consists of the rationals together with the usual arithmetic operations:

```
valof
        I = 1 fby I+1
        J = 3•I
    result = valof
                S = J fby S + next J
                N = 1 fby N+1
            result = S/N
        end
end.
```

† In this and subsequent papers, we use fby for followed by and asa for as soon as.

The semantics of USWIM tells us that (since there are no global variables) the meaning of the outer phrase is the value of result in the least environment E_1 satisfying the three definitions in the phrase. It is easy to see that $E_1(\underset{\sim}{I})$ is the history $\langle 1, 2, 3, \ldots \rangle$ and $E_1(\underset{\sim}{J})$ is the history $\langle 3, 6, 9, \ldots \rangle$. The value of E_1(result) is the value of the inner phrase in the environment E_1, which is the value of result in the least environment E_2, differing from E_1 only in the values of the locals ($\underset{\sim}{S}$, $\underset{\sim}{N}$ and result) of the inner phrase, which satisfies the definitions in the inner phrase. Thus $E_2(\underset{\sim}{J})$ is $\langle 3, 6, 9, \ldots \rangle$, $E_2(\underset{\sim}{S})$ is $\langle 3, 9, 18, \ldots \rangle$, $E_2(\underset{\sim}{N})$ is $\langle 1, 2, 3, \ldots \rangle$ and E_2(result) is $\langle 3, 4.5, 6, \ldots \rangle$. Thus the meaning of the program is $\langle 3, 4.5, 6, 7.5, \ldots \rangle$.

Notice that what the inner phrase is doing is maintaining a running average of the values of $\underset{\sim}{J}$. We can use this inner phrase as the body of function called $\underset{\sim}{Avg}$, as we do in the following example:

```
valof
    Avg(X) = valof
                 S = X fby S + next X
                 N = 1 fby N+1
             result = S/N
             end
        M = Avg(S) asa I eq 10
        I = 1 fby I+1
    result = Avg((S-M)**2) asa I eq 10
end.
```

Since this program has a free variable $\underset{\sim}{S}$, we can only talk of the value of this program in an environment E which gives a value to $\underset{\sim}{S}$. Let us suppose that $E(\underset{\sim}{S})$ is some history σ . In the inner environment, $\underset{\sim}{Avg}$ will be the "running average" function, so that $\underset{\sim}{M}$ is the constant history which is everywhere the average of the first ten values of $E(\underset{\sim}{S})$, that is, the average of σ_0 through σ_9[†]. Since the value of result will be the average of the first ten values of $(\underset{\sim}{S}-\underset{\sim}{M})^2$, we see that the value of the program in E will be the constant sequence which is everywhere equal to the variance (the second moment about the mean) of the first ten values of $\underset{\sim}{S}$ in E .

Probably the best way of viewing the previous program in an operational way is to consider $\underset{\sim}{Avg}$ as a coroutine, with two invokations, $\underset{\sim}{Avg}(\underset{\sim}{S})$ and $\underset{\sim}{Avg}((\underset{\sim}{S}-\underset{\sim}{M})^2)$. These invokations are considered as running from time 0 , but only the value of $\underset{\sim}{Avg}(\underset{\sim}{S})$ at time 9 and the value of $\underset{\sim}{Avg}((\underset{\sim}{S}-\underset{\sim}{M})^2)$ at time 9 are "used". The coroutine activations have to be considered as running from time 0 so that they can keep running sums of $\underset{\sim}{S}$ and $(\underset{\sim}{S}-\underset{\sim}{M})^2$ until they are needed at time 9.

Suppose we wished to generalise this program to compute arbitrary higher moments about the mean. We would need a function $\underset{\sim}{Pow}$ where $\underset{\sim}{Pow}(\underset{\sim}{X}, \underset{\sim}{N})$ gives us the running $\underset{\sim}{N}$-th powers of the values of $\underset{\sim}{X}$. We might try to define this as follows:

---

† Here we are using the convention that, if γ is a history, its elements are $\gamma_0, \gamma_1, \gamma_2, \ldots$.

```
Pow(X, N) = valof
                I = 0 fby I+1
                P = 1 fby X•P
            result = P asa I eq N
        end.
```

Now, *Pow*(5, 2) is ⟨25, 25, 25, ...⟩ and *Pow*(6, 3) is ⟨216, 216, 216, ...⟩ but if, say, the value of *A* is ⟨1, 2, 3, ...⟩ then the value of *Pow*(*A*, 2) is ⟨2, 2, 2, ...⟩ ! Moreover, the value of *Pow*(5, *A*) is ⟨5, 5, 5, ...⟩ and the value of *Pow*(5, *A*+1) is ⟨⊥, ⊥, ⊥, ...⟩ ; if the second argument *N* is changing with time, the variable *I* in the definition of *Pow* is chasing a moving target!

Similarly, if we tried to generalise the variance program to give the variance of the first *N* values of the history *X* , we would get rubbish if *N* varies with time.

The reason for all this is that globals have the same meaning inside a phrase as outside and so phrases are synchronised with their environments. We can have no subcomputations in ULU (but we can have one computation following another, as in the variance example). We have nesting of scope but we can not have nesting of computations, i.e. there are no subloops.

2. THE LANGUAGE LUSWIM(A)

The reason that the definition of *Pow* is considered to be wrong is that we expect it to work pointwise. If **A** had an explicit exponentiation operator, say ↑ , then, according to Lu(A) , if α and β are histories $\alpha\uparrow\beta$ would be $\langle\alpha_0\uparrow\beta_0, \alpha_1\uparrow\beta_1, \ldots\rangle$. This is exactly how we would expect *Pow* to work, but in ULU it clearly doesn't.

The root of the problem is that phrases are not defined pointwise. The value at time t of a phrase, in which *G* appears as a global, depends on the value of *G* not just at time t but at other times as well. For example, at any time t , the phrase computing the average of the first t+1 values of x depends on $x_0, x_1, \ldots, x_{t-1}$ as well as x_t . This is exactly what we wanted in the "average" example, but it is disastrous in examples like the one to compute the N-th power of x .

What is required is some way of ensuring that the value of a phrase, in environment *E* , at time t depends only on the values of the globals at time t . We can ensure this by freezing *E* at time t: given a sequence α and a time t , we define α^t to be the sequence $\langle\alpha_t,\alpha_t,\alpha_t,\ldots\rangle$; then the value the frozen environment E^t assigns to nullary variable *G* is $E(G)^t$. (The value E^t assigns to non-nullary variables will be discussed later.) The value of the phrase at time t is then the value at time t of result in the least environment, differing from E^t only in the values of the locals, which satisfies all the definitions in the phrase.

LUSWIM(Á) is the language obtained by using this pointwise interpretation of phrases, i.e. it is the "pointwise extension" of USWIM(Á) . Programs in LUSWIM and ULU have the same form, and we could allow the classes of programs to be the same, but we distinguish them by requiring that the intersection of the sets of variables they use contains only the individual variable result. In our examples, variables except result, in ULU programs, will be written in bold face italics, while all variables except result, in LUSWIM programs, will be written in light face italics. (In this way, ULU and USWIM can be disjoint subsets of Lucid.)

Here is an example of a simple LUSWIM program:

```
valof
        I = 1 fby I+1
        S = 1 fby S + next M
        M = valof
                K = 1 fby K+1
                P = I fby I•P
            result = P asa K eq I
        end
    result = S asa I eq 6
end.
```

This program has no global variables, so the difference between ULU and LUSWIM is not apparent until we consider the inner phrase, which has global variable I . In the (single) environment E inside the outer

phrase, the value of I clearly is <1, 2, 3, ...> . The value of M in E is determined separately at each time step. $E(M)_0$ is the value of result at time 0 in E' where E' is the environment obtained by freezing E at time 0 and then choosing values for K, P and result which satisfy the definitions in the inner phrase. Clearly $E'(P)$ is <1, 1, 1, ...> and E'(result) is <1, 1, 1, ...> . $E(M)_1$ is the value of result at time 1 in E'' where E'' is the environment obtained by freezing E at time 1 and then choosing values for the locals. So $E''(P)$ is <2, 4, 8, ...> and E''(result) is <4, 4, 4, ...>. Continuing this process, we see that $E(M)$ is $<1^1, 2^2, 3^3, ...>$. The value of the program at any time will be $1^1 + 2^2 + 3^3 + 4^4 + 5^5 + 6^6$.

In LUSWIM then, a single outer environment, like E in the example, does not determine a single inner environment, but rather a sequence of environments, like E' , E'' etc. , one for each outer time step.

Suppose now that a non-nullary variable M is defined by a phrase, for example suppose that E is an environment satisfying the definitions

$M(X, Y)$ = valof
$I = 0$ fby $I+1$
$S = X$ fby $S + Y$
result = S asa I eq $G+1$
end
$G = 0$ fby $G+1$

and that ψ is the history function which E assigns to M. Given any histories α and β, we know that $\psi(\alpha, \beta)$ is the meaning of $M(X, Y)$ in an environment $\hat{E}$ differing from E (at most) in that $\hat{E}(X) = \alpha$ and $\hat{E}(Y) = \beta$. The value of $M(X, Y)$ at time t is the value of the phrase at time t. We have defined this to be the value of result at time t in the inner environment E'; but in this inner environment the values of X and Y are frozen, i.e. $E'(X) = \langle\alpha_t, \alpha_t, \alpha_t, \ldots\rangle$ and $E'(Y) = \langle\beta_t, \beta_t, \beta_t, \ldots\rangle$. This means that there is no 'access' inside the phrase to other than the 'current' value α_t and β_t of X and Y respectively. As a result ψ has the property that $\psi(\alpha, \beta)_t = \psi(\alpha', \beta')_t$ for any α' and β' for which $\alpha_t = \alpha'_t$ and $\beta_t = \beta'_t$, in particular $\psi(\alpha,\beta)_t = \psi(\alpha^t,\beta^t)_t$.

We will say that any history function ψ is *elementary* if, for all histories $\alpha,\beta,\ldots$ and times t, $\psi(\alpha,\beta,\ldots)_t = \psi(\alpha^t,\beta^t,\ldots)_t$. All functions which are the pointwise extensions of data functions are elementary, but not all elementary functions are of this form.

An elementary function ψ is *pointwise* if $\psi(\alpha,\beta,\ldots)^t = \psi(\alpha^t,\beta^t,\ldots)$ for all $\alpha,\beta,\ldots$ and t. Pointwise functions are especially simple because they have the property that the value at a given time depends only on the values of the arguments at the given time, whereas with elementary functions the value can also depend on the time itself.

The function ψ described above is an example of an elementary function which is not pointwise. If

$$\alpha = \langle 2, 3, 4, 2, 7, 9, \ldots\rangle$$

$$\alpha' = \langle 6, 7, 1, 2, 9, 1, \ldots\rangle$$

$$\beta = \langle 1, 3, 5, 1, 8, 6, \ldots\rangle$$

$$\beta' = \langle 7, 2, 8, 1, 0, 0, \ldots\rangle$$

then

$$\psi(\alpha, \beta) = \langle 3, 9, 19, 6, 53, 45, \ldots \rangle$$

$$\psi(\alpha', \beta') = \langle 13, 11, 25, 6, 9, 1, \ldots \rangle .$$

At time 3, α and β agree with α' and β' because $\alpha_3 = 2 = \alpha'_3$ and $\beta_3 = 1 = \beta'_3$. As predicted, the values of $\psi(\alpha, \beta)$ and $\psi(\alpha', \beta')$ agree at this time, because $\psi(\alpha, \beta)_3 = 6 = \psi(\alpha', \beta')_3$. The function ψ is not pointwise, however, since $\alpha_0 = \alpha_3$ and $\beta_0 = \beta_3$ but $\psi(\alpha, \beta)_0 = 13$ and $\psi(\alpha, \beta)_3 = 6$. (Generally, $\psi(\gamma, \delta)_t = \gamma_{t+(t+1)\cdot\delta_t}$.)

Suppose now that ϕ is an elementary function. At any time t, the value of $\phi(\alpha, \beta, \gamma, \ldots)$ depends only on t and the values $\alpha_t, \beta_t, \gamma_t, \ldots$ of the arguments. This means there must be a data function θ_t such that

$$\phi(\alpha, \beta, \gamma, \ldots)_t = \theta_t(\alpha_t, \beta_t, \gamma_t, \ldots) .$$

for any $\alpha, \beta, \gamma, \ldots$. The elementary history function ϕ is therefore completely determined by the sequence $\langle \theta_0, \theta_1, \theta_2, \ldots \rangle$ of data functions (and any such sequence, in turn determines an elementary function). This means we can think of elementary functions as histories of data functions, with pointwise functions corresponding to constant histories of data functions. From this point of view, application of elementary functions works pointwise: the value of $\phi(\alpha, \beta, \gamma, \ldots)$ at time t is the 'value' of ϕ at time t (θ_t) applied to the values of $\alpha, \beta, \gamma, \ldots$ at time t.

This view of elementary functions as histories of data functions now explains what it means to freeze a function (if it is elementary).

The value α^u of an ordinary history α frozen at time u is the history $<\alpha_u, \alpha_u, \alpha_u, \ldots>$. We define the frozen value ϕ^u of an elementary function ϕ determined by the sequence $<\theta_0, \theta_1, \theta_2, \ldots>$ to be the elementary (in fact pointwise) function corresponding to the sequence $<\theta_u, \theta_u, \theta_u, \ldots>$. In other words

$$\phi^u(\alpha, \beta, \gamma, \ldots)_t = \theta_u(\alpha_t, \beta_t, \gamma_t, \ldots)$$

$$= \phi(\alpha^t, \beta^t, \gamma^t, \ldots)_u$$

for any $\alpha, \beta, \gamma, \ldots$ and any t.

This definition of ϕ^u allows us to give a meaning to phrases which have, as globals, variables denoting elementary functions. Notice that for any elementary function ψ, $\psi(\alpha, \beta, \gamma, \ldots)^t = \psi^t(\alpha^t, \beta^t, \gamma^t)$. (Compare this with the definition of pointwise functions.)

Function definitions in LUSWIM are, therefore, syntactically restricted in such a way as to ensure that all functions are elementary. The simplest way of ensuring this is to require the definiens of a function definition to be a phrase. We can weaken this requirement by requiring the definiens to be *elementary in the formal parameters* which means we allow occurrences of formal parameters outside phrases as long as they are not within the arguments of Lucid functions. For example,

$$M(X, Y) = X + Y \bullet \underset{\sim}{\text{next}}\ G$$

is a valid definition, and in fact this definition of M is equivalent to the previous one in the context of the previous definition of G .

As a result of this restriction, LUSWIM is, in a sense, the pointwise extension of USWIM , with both phrases and function application being extended pointwise.

Although the mathematical semantics of LUSWIM is more complicated than that of ULU , simpler and more conventional operational interpretations are possible. Function evaluation as well as evaluation of phrases can be thought of as subcomputations which take place while the enclosing computation is suspended. Functions without global variables can be thought of like data functions, while functions with global variables are like Algol function procedures whose globals have different values at different times. When a function is defined outside a phrase but used inside, the evaluation of the function uses the current "frozen" global, so that inside the phrase the function is pointwise.

This freezing of globals by phrases is one of the main reasons why we chose to define USWIM rather than base Lucid on ISWIM. If we used ISWIM, to get the analogue of LUSWIM we would have to define a pointwise extension of where phrases. This is clearly possible, but then we would have to distinguish between E and E **where end**, the second being a where phrase with an empty body. This is essential because in the first case the globals (free variables) would not be frozen, whereas in the second they would. In particular,

$$\underset{\sim}{\text{next}}\ X \text{ where end} = X \neq \underset{\sim}{\text{next}}\ X\ .$$

Rather than have these aesthetically displeasing empty bodies, we chose to use valof phrases.

3. THE LANGUAGE LUCID

It is apparent that these are two very different ways of adding iteration to USWIM, both potentially useful, and both having natural if radically differing operational semantics. It seems very unlikely that one is the right interpretation and the other the wrong one; in fact it is easy to imagine programs which (operationally speaking) use both coroutines and nested computation. An example would be a coroutine which computes a running N-th moment (average of N-th powers) but uses an inner, nested loop which computes the N-th power in a subcomputation.

Clearly the two 'facilities' should be combined in the same language, but it is not immediately obvious how to do this. The two languages are superficially very different in the way they give a meaning to a phrase. In ULU the entire value of the phrase is determined all at once in terms of the entire values of the globals, whereas in LUSWIM it is put together instant by instant in terms of the instantaneous values of the globals. The difference, however, is not as great as it seems, once it is realized that the ULU semantics can also be given in a pointwise manner: the value of a ULU phrase in an environment E at time t is the value of result at time t in the least environment satisfying the definitions in the phrase and differing from E at most in the values assigned to the locals. This definition is, of course, equivalent to the general USWIM definition but makes it clear that the difference between ULU and LUSWIM is that, on each time step, LUSWIM freezes the values of the globals before determining the inner environment. In ULU , the inner environments corresponding to different times are all

the same. To combine the two semantics of phrases we need only distinguish between two classes of variables (elementary and nonelementary), one of which (the elementary) consists of variables which are subject to freezing inside phrases. In this way it should be possible to mix up elementary and nonelementary variables in definitions almost at will. (Another method is to have only one kind of variable, but require the program to specify at the head of phrase those variables which are to be frozen. We will not use this method here.)

The language which results from combining LUSWIM(A) and ULU(A) in this way is the language Lucid(A) . The Lucid semantics requires that the meanings of elementary variables must, as in LUSWIM, be elementary (so they can be frozen). Lucid therefore carries over the LUSWIM restriction on the definitions of elementary variables, namely that the formal parameters must be elementary and the definiens must be elementary in the formal parameters (in the definition of "elementary in the formal parameters", for "Lucid functions" we now read "Lucid functions or non-elementary variables").

We assume therefore that we have two disjoint sets of variables (*elementary* variables and *nonelementary* variables) each containing a countably infinite number of variables of each arity, with result an elementary variable. In our examples elementary variables are lightface, and nonelementary variables are boldface. Programs in the languages discussed all have the *form* of ULU programs and differ only in the ways in which the two kinds of variables are used.

To make this more precise, let us define ULU*(A) to be the most general language, i.e. it is USWIM(Lu(A)) in which there are both kinds of variables without distinction (we are defining here only a set of syntactic objects, and will not attempt to assign meanings to all

these objects). The syntax of the USWIM family has been described elsewhere, so that to specify the syntax of Lucid and its two subsets it is sufficient to specify exactly the restriction on the form of definitions of elementary variables. Since the restriction is that the formal parameters be elementary and that the definienda be elementary in the formal parameters, it is, in turn, enough to specify what it means for a term to be elementary in a set of elementary variables.

Suppose then that V is a set of elementary variables. The set $El(V)$ of terms *elementary in the variables in* V is the least set of ULU* terms such that

(i) any term with no free occurrences of any variable in V is in $El(V)$;

(ii) a term consisting of a constant from Σ (the signature of A) and an appropriate length sequence of operands is in $El(V)$ if all the operands are;

(iii) a term consisting of an elementary variable together with an appropriate length sequence of arguments is in $El(V)$ if all the arguments are;

(iv) any phrase is in $El(V)$.

We will say that a term is elementary in a particular variable m iff it is elementary in $\{m\}$, and that a term is *elementary* iff it is elementary in all of its free elementary variables.

For example

$$A + \underset{\sim}{H}(B) + \text{valof}$$

$$P = \underset{\sim}{\text{next}}(A + C) \ \underset{\sim}{\text{fby}} \ P + M(A)$$

$$\text{result} = B + \underset{\sim}{\text{first}}(P)$$

$$\text{end}$$

is elementary in $\{A, C, P\}$, but not in $\{B\}$ because there is an occurrence of B inside a nonelementary function. The variables A and C also occur inside the range of a nonelementary constant but these are occurrences inside a phrase, and these occurrences are allowable since they refer to the frozen values (in a sense a phrase can be considered as the application an 'anonymous' elementary function to its globals). The variable P also occurs in the scope of a nonelementary constant but this also is allowable because the occurrence is not a free occurrence in the term as a whole (because P is a local of the phrase). If t is the above term we see that a definition of the form

$$M(A, C) = t$$

is allowed but one of the form

$$M(B, D) = t$$

is not, because t is not elementary in B .

Lucid programs are therefore ULU* terms having the property that the formal parameters of any elementary variable definition occurring in the program are elementary individual variables, and the definiendum is elementary in the set of formal parameters of the definition. The sets of ULU and LUSWIM terms are also as described earlier.

Before proceeding to the semantics of Lucid it is necessary first to define precisely the algebra $Lu(A)$, the class of elementary history functions, the operation of freezing, and so on.

Suppose then that we are given a continuous algebra A with signature Σ. The set Σ is usually considered to be a set of ranked operation (constant) symbols but we will assume (for simplicity) that Σ contains two extra symbols, namely the sort symbol U for the universe of A and the relation symbol $\sqsubseteq$ for approximation in A. This assumption allows us to consider A to be simply a function with domain Σ, one which assigns to U a nonempty set, which assigns to $\sqsubseteq$ a partial order on $A(U)$ which makes $\langle A(U), A(\sqsubseteq)\rangle$ a cpo, and which assigns to each n-ary operation symbol an $A(\sqsubseteq)$-continuous operation over $A(U)$. We also assume that Σ contains the nullary symbols **true**, **false** and Ω, with $A(\Omega)$ the $A(\sqsubseteq)$-least element of $A(U)$.

We require that Σ be 'normal' in that it does not contain the special Lucid operation symbols, which for our purposes we take to be the unary symbols first and next and the binary symbols asa and fby. This allows us to define $Lu(\Sigma)$ to be the result of adding these symbols to Σ, and to define $Lu(A)$ to be the function H with domain $Lu(\Sigma)$ such that

(i) $H(U)$ is the set of all infinite sequences of elements of $A(U)$, i.e. the set of all functions from the set $\{0, 1, 2, \ldots\}$ of natural numbers to $A(U)$ (elements of $H(U)$ will be called A-histories or simply histories).

(ii) $H(\Xi)$ is the pointwise extension of $A(\Xi)$, i.e. given any A-histories α and β, $\langle\alpha, \beta\rangle \in H(\Xi)$ iff $\langle\alpha_i, \beta_i\rangle \in A(\Xi)$ for all i;

(iii) for any operation symbol k in Σ, $H(k)$ is the pointwise extension of $A(k)$, i.e.

$$(H(k)(\alpha, \beta, \gamma, \ldots))_t = A(k)(\alpha_t, \beta_t, \gamma_t, \ldots)$$

(a function over $H(U)$ will be called an A-history function, or simply a history function);

(iv) $H(\underline{\text{first}})$, $H(\underline{\text{next}})$, $H(\underline{\text{fby}})$ and $H(\underline{\text{asa}})$ are the unary history functions first and next and the binary functions fby and asa where

(a) $\text{first}(\alpha) = \langle\alpha_0, \alpha_0, \alpha_0, \ldots\rangle$

(b) $\text{next}(\alpha) = \langle\alpha_1, \alpha_2, \alpha_3, \ldots\rangle$

(c) $\text{fby}(\alpha, \beta) = \langle\alpha_0, \beta_0, \beta_1, \ldots\rangle$

(d) $\text{asa}(\alpha, \beta) = \langle\alpha_i, \alpha_i, \alpha_i, \ldots\rangle$ where i is the least number for which $\beta_i = A(\text{true})$ and $\beta_j = A(\text{false})$ for all $j < i$

$= \langle A(\Omega), A(\Omega), A(\Omega), \ldots\rangle$ $(= H(\Omega))$. if no such i exists

for all histories α and β and all natural numbers t.

We can now proceed to give the semantics of Lucid. A *Lucid environment* is a function from the set of variables (of both types) to the set of history functions which assigns n-ary history functions to n-ary variables, and which in particular assigns elementary history functions to elementary variables. Given a Lucid environment E and a time s, the frozen environment E^s is the unique Lucid environment such that $E^s(m) = E(m)^s$ for any elementary variable m, and $E^s(\mathfrak{f}) = E(\mathfrak{f})$ for any nonelementary variable $\mathfrak{f}$.

We now define the meaning of a Lucid term t in a Lucid environment E, by induction on the structure of t ;

(i) if t consists of the Lu(E) operation symbol k together with operands $u_0, u_1, \ldots, u_{n-1}$, then the meaning of t in E is the result of applying $H(k)$ (i.e. Lu(A)(k)) to the meaning of the u_i in E ;

(ii) if t consists of a variable g (of either kind) together with actual parameters $u_0, u_1, \ldots, u_{n-1}$, then the meaning of t in E is the result of applying $E(g)$ to the meanings of the u_i in E ;

(iii) if t is a phrase then the meaning of t in E is α where at any time s, α_s is the meaning of result in E'_s at time s where E'_s is the least environment which satisfies the definitions in the phrase and agrees with E^s except possibly for the values E'_s assigns to the locals of t .

Finally, given a definition d of the form

$$g(x_0, x_1, \ldots, x_{n-1}) = a ,$$

we say that d is satisfied by E iff the meaning of $g(x_0, x_1, \ldots, x_{n-1})$ in $\hat{E}$ is equal to the meaning of a in $\hat{E}$ for any environment $\hat{E}$ agreeing with E except possibly for some of the values assigned to the formal parameters $x_0, x_1, \ldots, x_{n-1}$.

Notice that the definition differs from the USWIM definition only in part (iii), which makes use of the special structure of Lu(A) . In particular, notice that no special evaluation rule (calling mechanism)

is required for the elementary variables.

A Lucid program is simply a term, the free variables of which are called the input variables. An input to a program is simply an association of input values with input variables, and the meaning (or "output") of the program is its value in the appropriate environment. For example, suppose that A , B and C are the input variables of a program; then the meaning of the program given input α , β and γ is its meaning as a term in an environment in which $E(A) = \alpha$, $E(B) = \beta$, and $E(C) = \gamma$.

The above is not strictly speaking a valid definition, because an assumption is made which is not obviously true; namely, that a least Lucid environment E'_s always exists. It is possible to give a more indirect but obviously valid definition, and then to prove that the above is a true statement about the meaning so defined. We will not do this here.

We mentioned that the set of LUSWIM programs is syntactically a subset of the set of Lucid programs. The semantics of LUSWIM is determined by specifying that this inclusion hold semantically as well, in other words, the meaning of a LUSWIM program is its meaning considered as a Lucid program. It is apparent that this definition of LUSWIM conforms with the informal one given earlier because in a LUSWIM program all the globals of a phrase, being elementary, are frozen inside the phrase.

We do not have the choice of defining the semantics of ULU in this way because ULU is a member of the USWIM family and its meaning is determined by the 'generic' semantics of USWIM. Nevertheless, the

inclusion is still valid, i.e. the meaning of a ULU program is its meaning considered as a Lucid program; since all the globals of a ULU phrase are nonelementary, none will be frozen.

If we consider the USWIM variables to be elementary, then USWIM itself is a sublanguage of Lucid (or, to be more precise, USWIM(A) is a sublanguage of Lucid(A)). The same is true even if we consider the USWIM variables other than result to be nonelementary. In the absence of Lucid operations, freezing is pointless and every variable denotes a pointwise history function.

4. OPERATIONAL INTERPRETATIONS OF LUCID

We will illustrate possible operational views of Lucid by looking at several examples of Lucid programs.

The first example combines the two forms of iteration. The phrase has one nonelementary 'input' (free) variable $\underset{\sim}{X}$ and the program's value at time t is the 10th moment of the first t+1 values:

```
valof
        S = P fby S + next P
        X = X
        P = valof
                Y = 1 fby X•Y
                I = 0 fby I + 1
            result = Y asa I eq 10
            end
        J = 1 fby J + 1
    result = S/J
end.
```

The definition $X = \underset{\sim}{X}$ allows the inner phrase to freeze each particular value of $\underset{\sim}{X}$ in order to compute the 10-th power of that particular value. Using the nonelementary variable $\underset{\sim}{X}$ instead of X inside the inner phrase would give a completely different result. In general the value of the altered phrase

```
valof
        Y = 1 fby Y•X
        I = 0 fby I + 1
   result = Y asa I eq 10
end
```

at any time is the value of the product of the first 10 values of $\underset{\sim}{X}$: If this phrase were used in the previous program its value would also be at any time the product of the first 10 values of $\underset{\sim}{X}$.

In writing the above program we could, if we wanted, define a general elementary function *Pow* and just 'call' it to give the 10-th power of $\underset{\sim}{X}$. This program

```
valof
        S = P fby S + next P
        P = Pow(X, 10)
        N = 1 fby N + 1
   result = S/N
Pow(B,K) = valof
                Y = 1 fby B•Y
                I = 1 fby I + 1
           result = Y asa I eq K
           end
end
```

has the same value. Notice that it is *not* necessary to apply *Pow* to a lightface version of $\underset{\sim}{X}$. The restriction that the formal parameters

of an elementary function definition be elementary is not a type restriction, i.e. it does not restrict the class of possible actual parameters; nor is it an indication that elementary functions use some different 'calling conventions'. Elementary functions are simply history functions possessing a special property that makes them freezable, and the restrictions on the form which the definition of an elementary variable may take ensure that the denoted value will be elementary.

The examples just given used both kinds of iteration, but separately: the outer phrase had only a nonelementary function as its global and could be thought of as a ULU phrase, while the inner one had only an elementary variable as its global and could therefore be thought of as an ordinary LUSWIM phrase. Here is an example where the features are combined in one (the inner) phrase. The program computes the running variance of its input variable $\underset{\sim}{X}$, i.e. its value at time t is the variance of the first t+1 values of $\underset{\sim}{X}$:

```
valof
    Avg(V) = valof
                 S = V fby S + next V
                 I = 1 fby I + 1
             result = S/I
         end
      M = Avg(X)
  result = valof
             result = Avg((X - M)**2)
         end
end.
```

The outermost phrase is a pure ULU phrase because its only global is nonelementary, and the same is true of the phrase defining the function *Avg* (whose value is the running average of its argument). The second inner phrase, however, is quite different: one of its globals is the elementary variable M, but the other is the nonelementary variable $\underset{\sim}{X}$. The result is that the value of M inside this phrase is always its frozen outer value, whereas that of $\underset{\sim}{X}$ depends on the inner time.

The value of this 'mixed' phrase at time 2, for example, depends on the value of M at time 2 only, but on the values of $\underset{\sim}{X}$ at times 0, 1, and 2. If the value of $\underset{\sim}{X}$ begins $\langle 6, 8, 10, \ldots \rangle$ then the value of M begins $\langle 6, 7, 8, \ldots \rangle$ and the value of the phrase at time 2 is $((6 - 8)^2 + (8 - 8)^2 + (10 - 8)^2)/3$ which is $8/3$, as required. A mixed phrase is used because this (admittedly naive) algorithm implements directly the definition of variance which requires that the *present* average be subtracted from all previous values.

There are two conceptually different ways of giving an operational interpretation to 'mixed' phrases with both elementary and nonelementary globals. One way is to consider such a phrase as 'basically' an ordinary LUSWIM phrase which has additional special (nonelementary) variables which can be thought of as being 'restarted' at the beginning of every subcomputation. The phrase just discussed can be understood in this way, as can the phrase defining *isprime* in the program

```
valof
        P~ = 2 fby nxprime(P~)
    nxprime(Q) = valof
                     N = Q + 1 fby N + 1
                 result = N asa isprime(N)
                 end
    isprime(M) = valof
                 result = P~^2 < M asa P~^2 ≥ M ∨ P~ | M
                 end
        result = P~
end
```

whose value is the sequence $\langle 2, 3, 5, \ldots \rangle$ of all primes. The function *isprime* can be thought of as testing its argument for primeness using a simple loop which runs through all the primes starting with the first and checks whether one whose square is less than M actually divides M .

The other way of viewing a mixed phrase is to consider it as a parameterised set of ULU phrases, with each "freezing" of the global elementary variables yielding a ULU phrase. For example the following program

```
valof
        S = P fby S + next P
        P = X~ ↑ N
        I = 1 fby I + 1
    result = S/I
end
```

has as its value at time t the N-at-time-t-th moment of the values of $\underset{\sim}{X}$ up to time t (assuming the data function $\uparrow$ is exponentiation). The variable N is the parameter, and each numeric value of N yields an ordinary ULU phrase. In an environment in which N is constantly 2, the phrase is equivalent to the ordinary ULU phrase

```
valof
        S = P fby S + next P
        P = X ↑ 2
        I = 1 fby I + 1
    result = S/I
end.
```

In an environment in which N is constantly 3, it is equivalent to an ordinary ULU phrase which computes the third moment. In an environment in which the value of N changes irregularly with time between 2 and 3, the phrase can be considered as sampling the appropriate outputs of two different simultaneously running 'coroutines' computing running 2nd and 3rd moments respectively.

The same interpretations can be applied to nonelementary *functions*, the definitions of which use both elementary and nonelementary formal parameters and globals. For example, given the definition

```
Mom2(X, M) = valof
                 S = T fby S + next T
                 T = (X - M)^2
                 I = 1 fby I + 1
             result = S/I
             end
```

the value of *Mom2*(*A*, *N*) (in an appropriate environment) at time t is the 2nd moment of the first t+1 values of *A* about the value of *N* at time t . This function can be understood as an ordinary elementary function except that its special first argument is restarted every time the function is called.

It is, of course, possible to use two different viewpoints of the same object. We could, for example, also regard *Mom2* as an ordinary ULU function with a parameter *M* , so that

Mom2(*A*, 0)

is the running second moment, and

Mom2(*A*, *Avg*(*A*))

is the running variance. (*Mom2*(*A*, *Avg*(*A*)) can be viewed as a possibly infinite set of simultaneously running coroutines, one for each different value of the running average of *A* . The value of *Mom2*(*A*, *Avg*(*A*)) at time t is the value, at time t , of the coroutine corresponding to the running average of *A* at time t .)

These operational interpretations may be of help in visualising

the "running" of Lucid programs, but an actual implementation may work completely differently.

5. PROGRAM MANIPULATION RULES

The program manipulation rules of Lucid are exactly analogous to those of USWIM, allowing modular or nested proofs in the same way. The use of the manipulation rules must be restricted because of the way in which elementary variables are frozen inside phrases, and because of the way in which elementary and nonelementary variables are treated differently, the latter being 'immune' to freezing. (The restrictions given in this section do not just "seem to work" - they have all been justified from the formal semantics of Lucid.)

To understand the nature and necessity of these restrictions, consider the following simple Lucid program

```
valof
        I = 0 fby I + 1
        G = H + I
    result = valof
                P = 1 fby P•G
                K = 0 fby K + 1
            result = P asa K eq I
            end
end.
```

It is easy to see that if an environment E assigns the value η to H,

then the value of the term in E at time t is $(\eta_t + t) \uparrow t$.

Let us consider the USWIM import rule which, together with the addition and deletion rules, effectively allows us to move definitions in and out of phrases whenever no clashes of variables result. If we were to consider the above as a USWIM phrase, the rules would conclude that the definitions of I and G could be added to the inner phrase, and that the definitions of P and $\underset{\sim}{K}$ could be moved outside to the enclosing phrase. In fact only one of these changes preserves the meaning of the program, as we shall see.

Suppose first that the definition of I were added to the inner phrase. The result is that I, inside the phrase, would no longer be a frozen variable; in fact it would change in step with $\underset{\sim}{K}$, the test of the asa would be true at time 0 , and the value of the whole term would be constantly 1.

Similarly, if the definition if G were moved inside the phrase, it too would no longer be constant because $\underset{\sim}{H}$ would be considered to be 'restarted' in each inner subcomputation. A little calculation shows that the value of the whole term at time t would be

$$(\eta_0 + t)\cdot(\eta_1 + t)\cdot\ldots\cdot(\eta_{t-1} + t) \quad .$$

In the same way, if the definition of P were moved outside, P would be the running product of the expression $\underset{\sim}{H} + I$ and the value of the whole term at time t would be

$$(\eta_0 + 0)\cdot(\eta_1 + 1)\cdot\ldots\cdot(\eta_{t-1} + (t-1)) \quad .$$

Of these four the only change which preserves the meaning of the program is the moving of the definition of $\underset{\sim}{K}$ to the outer phrase. Since $\underset{\sim}{K}$ is a nonelementary variable, it is 'restarted' inside the inner phrase and the result is the same.

Unrestricted use of the **USWIM** import rule is obviously not safe, and the problem is to specify simple restrictions which define special circumstances in which it is valid.

The first restriction we will impose is that the definiendum of the definition to be moved must be elementary in its free elementary variables that are not formal parameters, for otherwise 'inner' and 'outer' times will get confused. For example, if the definition A = next B is outside a phrase, then it is not necessarily true as an assertion inside the phrase (because both A and B are constant inside). What is true inside the phrase is that the current (constant) value of A is equal to the (constant) value B will have on the next invocation of the loop. Since the definition may not be true as an assertion inside the phrase (i.e. may not be true in the inner environments), moving it inside could clearly change the meaning of the phrase.

The restriction that a definition be elementary in its free elementary variables prevents us from moving the definition of I and P (in the sample program) but not the definitions of G and K. Moving the definition of G causes changes because the definition mixes elementary and nonelementary variables. When it is moved some are frozen while others are not, and the result is that the values of variables being combined come from different times.

The second restriction we impose on the import rule, therefore, is that the definition be *uniform*, that is, it does not have free occurrences of both elementary and nonelementary variables.

Since the definiendum always occurs free in a definition, the combined effect of the two restrictions can be restated as follows:

(i) a definition of an elementary variable can be imported if the definiens has no free occurrences of nonelementary variables and is elementary (recall that it is required to be elementary in its formal parameters anyway);

(ii) a definition of a nonelementary variable can be imported if the definiens has no free occurrences of any elementary variables other than formal parameters.

Definitions of nonelementary variables can of course (as indicated) be imported even if the definiens are not elementary in their formal parameters. This means, for example, that the definition $F(X, N) = X$ + next N can be imported.

Most of the remaining USWIM rules involve substitution and therefore include the implicit restriction that the required substitutions be permitted. USWIM forbids substitutions which cause a clash of variables (scope), but Lucid must also prevent a 'clash' of inner and outer times. The required additional restriction is that a substitution cannot be applied to a phrase unless the assignments satisfy exactly the same requirements (just described) that definitions must satisfy if they are to be imported. Thus the substitution $\{F(X) \leftarrow X$ + next $H, M(Y) \leftarrow A + B \cdot Y\}$ can be applied to a phrase (assuming the ordinary USWIM requirements are satisfied) but one that contains one

of thc assignments $H \leftarrow K + A$, $B \leftarrow C +$ next D , or $P \leftarrow Q + R$ cannot.

We can now run through the remaining USWIM rules, indicating when they are valid Lucid rules.

The calling rule can be applied in Lucid whenever the substitutions are permitted (i.e. subject to the additional Lucid restrictions on substitution).

The local renaming rule is subject only to the USWIM restrictions provided the new variables are of the same type (elementary or nonelementary). (If a local does not occur free inside an inner phrase, the type may be changed in a renaming.)

The formal parameter renaming rule is also subject only to the USWIM restrictions if the types remain the same. (In a definition of a nonelementary function, formal parameters not appearing free in any phrase in the definiens may change type.)

The amalgamation rule requires the additional restriction that the expression e be elementary and that the v_i be elementary.

The definition addition rule requires no extra conditions (the definitions must of course be valid Lucid definitions).

The basis rule is valid without extra restrictions (the equations can use Lucid symbols if the equations are true in $Lu(A)$).

One consequence of the fact that the USWIM manipulation rules carry over almost directly to Lucid is that binding and calling "work" just as they do in USWIM . The "parameter passing mechanism", therefore, is *not* call by value, and binding is static, not dynamic.

6. INFERENCE RULES

In [3] we described how a first order language based on Σ could be used to make assertions about and annotate terms in USWIM($\dot{A}$) . In the same way a first-order language based on $Lu(\Sigma)$ can be used to annotate ULU programs, with all variables denoting histories. The same language could be used to annotate Lucid programs, but the rules for manipulating annotations (like those for manipulating programs) require extra restrictions. For example, an assertion about the globals of a phrase which is an annotation of an enclosing phrase can be added to the inner phrase if it is elementary and uniform (we should define what these terms mean for assertions).

Such a system is, however, very unnatural because of the radically different logics used in the program and in the annotation language. In the language itself a term like $X < Y$ denotes a history of truth values, while in the annotation language a formula like $X < Y$ must denote a single truth value. Assertions in the annotation language must explicitly refer to whole histories of program truth values as single truth values.

A more natural approach is to modify the logic of the annotation language and let the truth of formulas themselves also depend on time, i.e. adopt a form of modal or tense logic. Such a logic was developed in [1] and could be adapted to Structured Lucid, but we will not enter into details here.

We can, for the time being, avoid the difficulties connected with three-valued and modal logic by using an equational assertion language (as described in [3]). The assertions are universally

quantified equations between Lucid terms, the equation

$$\forall x_0\ \forall x_1 \ldots \forall x_{n-1}(t_0 = t_1)$$

being true (in Lu(A)) in an environment E iff t_1 and t_2 have the same value in any environment differing from E at most in the values given to some of the x_i .

The equational rules of substitution and replacement (as described in [3]) are sound (provided of course that the extra restrictions on permission are taken into account). Any equation in the language of A that is true in A is also true in Lu(A) , as are equations concerning the Lucid functions such as the following

$\forall X \forall Y$(first(X fby Y) = first X)

$\forall X \forall Y$(next (X fby Y) = Y)

$\forall X$(first X fby next X = X)

$\forall X \forall P$(X asa (true fby P) = first X)

$\forall X \forall P$(X asa P = if first P then first X else

next X asa next P) .

From these many others may be deduced using only the equational rules; for example,

$\forall X$(first(first X) = first X .

Quantified variables may be renamed whenever the required substitutions are permitted.

There is one special rule which captures the idea that the elementary globals of a phrase are "frozen" inside the phrase. The *freez-*

ing rule says that if elementary variable δ is an n-ary global of a phrase, we can add

$$\forall x_0\ \forall x_1\ \ldots\ \forall x_{n-1}\ \underset{\sim}{\text{first}}\ \delta(x_0, x_1, \ldots, x_{n-1})$$
$$= \delta(\underset{\sim}{\text{first}}\ x_0, \underset{\sim}{\text{first}}\ x_1, \ldots, \underset{\sim}{\text{first}}\ x_{n-1})$$

as an annotation of the phrase.

Special rules of inference (like the mathematical induction rule) valid in A are not necessarily valid in $\mathrm{Lu}(A)$, although they can usually be reformulated. For example, in $\mathrm{Lu}(N)$ the following rule is valid : to prove an equation of the form

$$\forall v\ e[\{v \leftarrow \underset{\sim}{\text{first}}\ v\}]$$

from a set of assumptions, one first proves $e[\{v \leftarrow 0\}]$ and $e[\{v \leftarrow \Omega\}]$ and then proves $e[\{v \leftarrow \underset{\sim}{\text{first}}\ v + 1\}]$ from the assumptions plus the equation $e[\{v \leftarrow \underset{\sim}{\text{first}}\ v\}]$ (the variable v must not occur free in any of the assumptions). This version of the rule is valid in $\mathrm{Lu}(N)$ because it is essentially restricted to constants. It can be used, for example, to prove the equation $K = \underset{\sim}{\text{first}}\ N$ from the equations

$$I = 0\ \underset{\sim}{\text{fby}}\ I + 1$$
$$K = I\ \underset{\sim}{\text{asa}}\ I\ \text{eq}\ \underset{\sim}{\text{first}}\ N$$

and various equations true in $\mathrm{Lu}(N)$, by proving the equation

$$\forall M \forall L(\underset{\sim}{\text{first}}\ M + \underset{\sim}{\text{first}}\ L = I + \underset{\sim}{\text{first}}\ L\ \underset{\sim}{\text{asa}}\ I\ \text{eq}\ \underset{\sim}{\text{first}}\ M)$$

by induction on $\underset{\sim}{\text{first}}\ M$.

$\mathrm{Lu}(A)$ itself also has a special rule, the (computation) induction rule. In order to prove the equation $u_0 = u_1$ one proves the equation

$$\underset{\sim}{\text{first}}\ u_0 = \underset{\sim}{\text{first}}\ u_1$$

and then the equation

$$\underset{\sim}{\text{next}}\ u_0 = \underset{\sim}{\text{next}}\ u_1$$

from the assumptions together with the equation $u_0 = u_1$ itself (as the induction hypothesis). It is required that the latter derivation use only the *elementary* substitution rule, i.e. only instances of the rule in which t_1 and t_2 are elementary in ν . For example, the rule can be used to prove the equation $J = I^2$ from the equations

$$I = 0\ \underset{\sim}{\text{fby}}\ I + 1$$

$$J = 0\ \underset{\sim}{\text{fby}}\ J + 2 \cdot I + 1\ .$$

Lucid terms can be annotated exactly like USWIM terms, and the USWIM rules for manipulating annotated USWIM terms can be applied to annotated Lucid terms provided extra restrictions are imposed which avoid clash of times. These extra restrictions involve extending the notion "elementary" to apply to annotated Lucid terms as well, but we will only give here the details for the import rule.

The Lucid annotation import rule says that the equation

$$\forall x_0\ \forall x_1\ \ldots\ \forall x_{n-1}(t_1 = t_2)$$

can be imported provided

(i) there are no free occurrences in the above assertion of locals of the inner (receiving) phrase, or of formal parameters of the definition within whose definiendum the inner phrase occurs; (this is the USWIM restriction);

(ii) the equation is elementary, i.e. t_1 and t_2 are elementary in all their elementary free variables which are not among the x_i ;

(iii) the equation is uniform, i.e. does not have free occurrences of both elementary and nonelementary variables.

Thus the equation

$$\forall X\ F(X, A) = \underset{\sim}{\text{next}}\ X\ + A$$

can be imported into a phrase if F and A are not locals or formal parameters, but the equations

$$\forall X\ F(X, A) = \underset{\sim}{\text{next}}\ X\ + A \bullet \underset{\sim}{H}$$

and

$$\forall X\ F(X, A) = \underset{\sim}{\text{next}}\ X\ + \underset{\sim}{G}(A)$$

cannot.

For the export rule, the first two restrictions are retained but the third is dropped : it is sufficient that the assertion be elementary. All but the third example can (barring clashes of scope) be exported from a phrase.

7. AN EXAMPLE OF PROGRAM TRANSFORMATION

As an example of the rules for the manipulation of program and annotations, consider the following program

valof

$\underset{\sim}{Avg}(\underset{\sim}{X})$ = valof

$S = \underset{\sim}{X}$ fby S + next $\underset{\sim}{X}$

N = 1 fby N + 1

result = $\underset{\sim}{S}/N$

end

$M = \underset{\sim}{Avg}(\underset{\sim}{A})$

result = valof

result = $\underset{\sim}{Avg}((\underset{\sim}{A} - M)^2)$

end

end

interpreted in the algebra Q whose universe is the set of rational numbers plus $\perp$, and which interprets the arithmetic operation symbols in the usual way (with division by 0 yielding $\perp$). The program calculates a running variance of the input variable $\underset{\sim}{A}$ using a naive algorithm which (operationally speaking) involves going all the way back to the beginning at each step. We will outline how our rules could be used to show the program equivalent to one which is more 'efficient'.

Our first step is to prove that that $\underset{\sim}{Avg}$ is linear and that $\underset{\sim}{Avg}$ is the identity function when applied to constants,i.e. to attach to the outer phrase the equations

$$\forall \underset{\sim}{G} \forall \underset{\sim}{H}\ \underset{\sim}{Avg}(\underset{\sim}{G} + \underset{\sim}{H}) = \underset{\sim}{Avg}(\underset{\sim}{G}) + \underset{\sim}{Avg}(\underset{\sim}{H})$$

$$\forall \underset{\sim}{G} \forall \underset{\sim}{K}\ \underset{\sim}{Avg}(\underset{\sim}{\text{first}}\ \underset{\sim}{K} \cdot \underset{\sim}{G}) = \underset{\sim}{\text{first}}\ \underset{\sim}{K} \cdot \underset{\sim}{Avg}(\underset{\sim}{G})$$

$$\forall \underset{\sim}{K}\ \underset{\sim}{Avg}(\underset{\sim}{\text{first}}\ \underset{\sim}{K}) = \underset{\sim}{\text{first}}\ \underset{\sim}{K}$$

We begin by using the definition rule which allows us to attach the quantified definition of $\underset{\sim}{Avg}$ to the phrase. Then we apply the substitution rule twice to give equations for $\underset{\sim}{Avg}(\underset{\sim}{G})$ and $\underset{\sim}{Avg}(\underset{\sim}{H})$. Then replacement applied to the identity

$\underset{\sim}{Avg}(\underset{\sim}{G}) + \underset{\sim}{Avg}(\underset{\sim}{H}) = \underset{\sim}{Avg}(\underset{\sim}{G}) + \underset{\sim}{Avg}(\underset{\sim}{H})$ gives

$\underset{\sim}{Avg}(\underset{\sim}{G}) + \underset{\sim}{Avg}(\underset{\sim}{H}) =$

$$\left(\begin{array}{l}\textsf{valof}\\ \qquad S = \underset{\sim}{G}\ \underset{\sim}{\textsf{fby}}\ S + \underset{\sim}{\textsf{next}}\ \underset{\sim}{G}\\ \qquad N = 1\ \underset{\sim}{\textsf{fby}}\ N + 1\\ \quad \textsf{result} = S/N\\ \textsf{end}\end{array}\right) + \left(\begin{array}{l}\textsf{valof}\\ \qquad S = \underset{\sim}{H}\ \underset{\sim}{\textsf{fby}}\ S + \underset{\sim}{\textsf{next}}\ \underset{\sim}{H}\\ \qquad N = 1\ \underset{\sim}{\textsf{fby}}\ N + 1\\ \quad \textsf{result} = S/N\\ \textsf{end}\end{array}\right)$$

Two applications of the local renaming rule and one of the amalgamation rule (taking e to be $R1 + R2$) plus replacements gives

$$\begin{array}{l}\underset{\sim}{Avg}(\underset{\sim}{G}) + \underset{\sim}{Avg}(\underset{\sim}{H}) = \textsf{valof}\\ \qquad S1 = \underset{\sim}{G}\ \underset{\sim}{\textsf{fby}}\ S1 + \underset{\sim}{\textsf{next}}\ \underset{\sim}{G}\\ \qquad N1 = 1\ \underset{\sim}{\textsf{fby}}\ N1 + 1\\ \qquad R1 = S1/N1\\ \qquad S2 = \underset{\sim}{H}\ \underset{\sim}{\textsf{fby}}\ S2 + \underset{\sim}{\textsf{next}}\ \underset{\sim}{H}\\ \qquad R2 = S2/N2\\ \quad \textsf{result} = R1 + R2\\ \textsf{end}.\end{array}$$

To this amalgamated phrase we add the definitions $N = 1$ fby $N + 1$ and $S = (G + H)$ fby $S +$ next$(G + H)$, then add the definitions of $N, N1, N2, S, S1, S2$ as annotations, and three applications of the Lucid induction rule give us the equations $N = N1$, $N = N2$, $S = S1 + S2$. The variables N and $N1$ do not depend on (are not defined directly or indirectly in terms of) $R1$, and so with the modification rule we can use the equation $N = N1$ to modify the definition of $R1$ to make it $R1 = S1/N$. Applying a similar change to the definition of $R2$, the annotated phrase we are working with becomes (after discarding useless annotations)

valof

$$\left.\begin{aligned} S1 &= G \text{ fby } S1 + \text{next } G \\ N1 &= 1 \text{ fby } N1 + 1 \\ R1 &= S1/N \\ S2 &= H \text{ fby } S2 + \text{next } H \\ N2 &= 1 \text{ fby } N2 + 1 \\ R2 &= S1/N \\ \text{result} &= R1 + R2 \\ N &= 1 \text{ fby } N + 1 \\ S &= (G + H) \text{ fby } S + \text{next}(G + H) \end{aligned}\right| \; S = S1 + S2$$

end

We can now begin to clean up the phrase. Two applications of the calling rule turns the definition of result into result $= S1/N + S2/N$ and with the modification rule and the equation

$$\forall U \forall V \forall W \ (U/W + V/W = (U + V)/W)$$

which is true in Q the definition of result is

$$\text{result} = (S1 + S2)/N \quad .$$

None of the variables S, $S1$ or $S2$ depend on result so that the modification rule can be applied to give result = S/N . The variables $S1$, $S2$, $N1$ and $N2$ are no longer used (occur free only in their definitions) and so can be eliminated using the deletion rule. As a result our original program now has the annotation

$$\underset{\sim}{Avg}(\underset{\sim}{G}) + \underset{\sim}{Avg}(\underset{\sim}{H}) = \text{valof}$$

$$S = (\underset{\sim}{G} + \underset{\sim}{H}) \ \underset{\sim}{\text{fby}} \ S + \underset{\sim}{\text{next}}(\underset{\sim}{G} + \underset{\sim}{H})$$

$$N = 1 \ \underset{\sim}{\text{fby}} \ \underset{\sim}{N} + 1$$

$$\text{result} = S/N$$

$$\text{end.}$$

We can now use the definition of $\underset{\sim}{Avg}$ and apply it to $\underset{\sim}{G} + \underset{\sim}{H}$ to find that the above phrase is equal to $\underset{\sim}{Avg}(\underset{\sim}{G} + \underset{\sim}{H})$; replacing equals gives us

$$\underset{\sim}{Avg}(\underset{\sim}{G}) + \underset{\sim}{Avg}(\underset{\sim}{H}) = \underset{\sim}{Avg}(\underset{\sim}{G} + \underset{\sim}{H})$$

and since no assumptions were made about $\underset{\sim}{G}$ and $\underset{\sim}{H}$, we may quantify over these variables.

Similar manipulations yield the second desired equation.

These two equations have only $\underset{\sim}{Avg}$ as a free variable, and it is nonelementary. The equations are therefore elementary and uniform and so can be imported into the inner phrase. Since M is an elementary global, using the freezing rule we can add the annotation $M = \underset{\sim}{\text{first}} \ M$ to the inner phrase. In this phrase the two equations together with certain arithmetic laws like

$$\forall Y \forall Z \ ((Y - Z)^2 = Y^2 - 2 \cdot Y \cdot Z + Z^2)$$

give us (after discarding annotations)

valof

$\underset{\sim}{Avg}(\underset{\sim}{X})$ = valof

$S = \underset{\sim}{X}$ fby S + next $\underset{\sim}{X}$

N = 1 fby N + 1

result = S/N

end

$M = \underset{\sim}{Avg}(\underset{\sim}{A})$

result = valof

result = $\underset{\sim}{Avg}(\underset{\sim}{A}^2) - 2 \cdot M \cdot \underset{\sim}{Avg}(\underset{\sim}{A}^2) + M^2$

end

end.

These manipulations have allowed us to move the elementary variable M out of the range of the nonelementary variable $\underset{\sim}{Avg}$. The result is that result is now defined by the elementary term

$$\underset{\sim}{Avg}(\underset{\sim}{A}^2) - 2 \cdot M \cdot \underset{\sim}{Avg}(\underset{\sim}{A}) + M^2$$

and so the result rule can be applied to make the definition of result in the main phrase

$$\text{result} = \underset{\sim}{Avg}(\underset{\sim}{A}^2) - 2 \cdot M \cdot \underset{\sim}{Avg}(\underset{\sim}{A}) + M^2.$$

We can now use the definition of M, the calling rule, the modification rule and some properties of Q to transform the program into

```
valof
    Avg(X) = valof
                 S = X fby S + next X
                 N = 1 fby N + 1
             result = S/N
         end
    result = Avg(A**2) - Avg(A)**2
end
```

which (assuming *Avg* yields averages) clearly has the running variance of *A* as its values. (These last manipulations could not have been performed inside the inner phrase because the definition of *M*, though elementary, was not uniform, and so could not be imported.)

To obtain a more 'efficient' version we use the calling rule twice to yield

```
valof
   Avg(X) = valof
                S = X fby S + next X
                N = 1 fby N + 1
             result = S/N
          end
   result = (valof
                S = A^2 fby S + next A^2
                N = 1 fby N + 1
             result = S/N
          end) + (valof
                S = A fby S + next A
                N = 1 fby N + 1
             result = S/N
          end)^2
end.
```

Since *Avg* no longer appears free in the definition of result, its definition can be discarded. The local renaming and amalgamation rules give us (much as before)

```
valof
     result = valof
                S1 = A fby S1 + next A
                N1 = 1 fby N + 1
                R1 = S1/N
                S2 = A^2 fby S2 + next A^2
                R2 = S2/N
             result = R1 + R2^2
          end
end.
```

The amalgamation rule allows us to eliminate the enclosing phrase, and then manipulations essentially the same as those already performed yield the final form of the program

valof

$$S1 = \underset{\sim}{A} \text{ fby } S1 + \text{next } \underset{\sim}{A}$$

$$S2 = \underset{\sim}{A}^2 \text{ fby } S2 + \text{next } \underset{\sim}{A}^2$$

$$N = 1 \text{ fby } N + 1$$

$$\text{result} = (N \cdot S2 - S1^2)/N^2$$

end

which computes the running variance of $\underset{\sim}{A}$ by the more sensible method of keeping (internally) running totals of the values of $\underset{\sim}{A}$ and $\underset{\sim}{A}^2$.

We gave an outline only of the transformation from one form to the other, but it should be apparent that *every* step can be justified by one of our rules. A completely formal justification in which every step is made explicit could be very long, but generating and checking it involves for the most part exactly the sort of bookeeping and trivial manipulations that computers excel in, no more difficult than that performed by existing proofcheckers for imperative languages.
It is well within the 'state of the art' of mechanical theorem proving to construct a system which could run through the entire transformation described aided only by a few vital hints from a human.

8. CONCLUSION

We have shown that it is feasible to combine Basic Lucid and USWIM in a single language, called (Structured) Lucid, without sacrificing the fundamental properties of either. This is possible because, while both languages are based on equations, their particular features are almost completely orthogonal. The few changes to the syntax and semantics of USWIM are the result of distinguishing between inner and enclosing local time, and it is exactly the possibility of such a distinction that makes Structured Lucid the expressive language that it is.

We have explained or at least mentioned that Lucid programs can be interpreted operationally in several ways, for example as defining conventional loops, dataflow networks, or systems or coroutines. This last possibility is especially significant because with conventional imperative languages the addition of coroutine facilities usually complicates enormously the problems of semantics and verification. Structured Lucid, however, has a simple semantics and powerful inference/manipulation rules, and we were able to illustrate the latter on a nontrivial 'coroutine' program.

We believe that Structured Lucid is a good example of the power and potential of the denotationally prescriptive approach [4].

9. REFERENCES

[1] Ashcroft, E.A. and Wadge, W.W., "Lucid - A Formal Theory for Writing and Proving Programs", SIAM J. Comput. 5, No. 3, pp. 336-354.

[2] _____ "Lucid, A Nonprocedural Language with Iteration", CACM 20, No. 7, pp. 519-526.

[3] _____ "A Logical Programming Language", CS-79-20, Computer Science Department, University of Waterloo. June, 1979.

[4] _____ "R_x for Semantics", CS-79-37, Computer Science Department, University of Waterloo. August, 1979.

[5] Landin, P.J., "The Next 700 Programming Languages", CACM 9, pp. 157-164.