# DETERMINISTIC OPERATIONAL SEMANTICS FOR LUCID

by

T.A. Cargill

Research Report CS-76-19

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

June 1976

# DETERMINISTIC OPERATIONAL SEMANTICS FOR LUCID

Tom Cargill

## 0. Introduction

Lucid is a language in which programs can be expressed, and at the same time a formal system in which properties of Lucid programs can be derived. The language is defined in Ashcroft and Wadge [1], and a knowledge of this definition is assumed. We will be concerned here with Lucid purely as a programming language, and with the problem of its implementation. In particular, the purpose of this paper is to present operational semantics for Lucid, prove them correct with respect to the fixpoint semantics of [1] and indicate how the operational semantics have been used in the construction of an interpreter for Lucid.

Other work on the implementation of Lucid is in progress by C.M. Hoffmann, M. Farah (University of Waterloo) and D. May (University of Warwick, England).

Section 1 explains why the implementation of Lucid is difficult and interesting, by exposing some aspects of the language which are not immediately observed in the published examples of Lucid programs [2,3]. Section 2 gives the details of the operational semantics and verifies their correctness. Section 3 is an informal description of the interpreter as it has been implemented. The conclusion contrasts this approach with that of the other work mentioned above.

The notation of [1] is used throughout with *followed by* and *as soon as* abbreviated to *fby* and *asa*, respectively. We will use the word "time" informally when we mean an index list of natural numbers. Thus we say "x at time t" when $x_t$ is meant etc. Lastly, the symbol $\perp$ will be used ambiguously to denote the bottom element of a number of c.p.o.'s. Its exact meaning will be clear from the context.

## 1. Some Difficult Lucid Programs

The sample Lucid programs which appeared in [2] and [3] might lead one to believe that there is a natural translation of Lucid programs into an Algol-like language. This would make the implementation of Lucid relatively easy. Unfortunately, this is not the case. A few simple examples of some of the unconventional constructions that can be achieved in Lucid should convince the reader that implementation of Lucid is far from trivial. In order to highlight the significant points of interest, the examples are kept as short as possible and non-essential details are omitted. The operation symbols f,g,h,... appearing in the examples are pointwise operations.

The variables x and y of program $P_1$ in Fig.1.1 are of interest. In this program, the variable p acts as a switch to determine whether x is to be computed from z, and y from x or y is to be computed from z, and x from y.

```
p = true fby ¬p
z = 1 fby f(z)
x = if p then h(z) else h'(y)
y = if ¬p then g(z) else g'(x)
```

$P_1$ Fig.1.1

If $P_1$ were to be translated into Algol it would require a section of code similar to Fig.1.2

```
    ⋮
P := ¬p;
z := f(z);
if p then begin x := h(z); y := g'(x) end
     else begin y := g(z); x := h'(y) end;
    ⋮
```

Fig.1.2

So in this case we have a translation. But the reader is urged to consider the problem in the general case. The solution to this problem for arbitrary programs is not simple. The variables x and y appear, syntactically, to depend upon each other in a cyclic fashion. It is only when the value of p is known that the true dependency emerges.

The second example is a result of the definition of the <u>or</u> operator $\vee$. In any standard structure S, $\vee_S$, the interpretation of $\vee$, is defined such that

$$\underline{\text{true}} \vee_S \perp_S = \underline{\text{true}}$$

$$\perp_S \vee_S \underline{\text{true}} = \underline{\text{true}}$$

From the point of view of implementation, this implies that when either of the subterms of the term $\alpha \vee \beta$ has the value <u>true</u>, the value of $\alpha \vee \beta$ is <u>true</u>. The problem is that one subterm may involve a non-terminating computation while the other returns <u>true</u> after some finite time. The program $P_2$ of Fig.1.3 illustrates this.

```
    ⋮
t = (w asa x) v (y asa z)
    ⋮
```

$P_2$ Fig.1.3

Any implementation cannot decide if either side will or will not terminate. All implementations will have to resort to some degree of parallel activity to elaborate this program. [Whether such a program is "reasonable" is beyond the scope of this paper. It is a valid construction in Lucid as defined.]

Program $P_3$ of Fig.1.4 illustrates the last difficulty to be mentioned here. The problem in implementing $P_3$ is that there are potentially non-terminating computations which are not essential to the computation of the output variable and should therefore be avoided.

```
p = ...
x = 1 fby if p then g(x) else h(y)
output = x asa ...
begin
   y = ...
     ⋮
end
```

$P_3$ Fig.1.4

We observe that values of x are determined either from the previous value of x or from the separate variable y, where y is computed by an inner loop whose termination is not guaranteed. A value of x will be required at some point established by the right hand side of the asa defining output. But attempting to compute all of the x sequence may involve computing y at a point where the inner loop does not terminate, while waiting until it is known which x is required before computing any will result in programs that are far from simple nested and concatenated loops.

These three examples should suffice to convince the reader that a complete implementation of Lucid is not a simple matter if we require that Algol-like code be generated. We therefore approach the problem by looking for an interpretive means of elaborating Lucid programs.

## 2. Operational Semantics

### 2.0 Introduction

At present, the only means of overcoming the problems outlined in Section 1 is to use a "data driven" or"request driven" implementation that views Lucid as a data flow language. The principle behind such an implementation is that a set of variables with associated time parameters is established. This set contains those variable-time pairs whose values are required to compute the solution to the program. If the first member of the output sequence is to be computed, say, then this set of "requests" would initially be the singleton containing <output,0>. When variables in this request set can be determined directly from the program or from those variables for which values are already known, this value is associated with the variable in a "memory". The memory is initially empty but as more values are computed, it forms a better and better approximation to the solution of the program. When variables in the request set depend on other variables which are not in the request set, then those new variables are added. The process proceeds until the value of the original request variable is known.

It is the purpose of this section to make this idea precise and to establish its correctness.

## 2.1 A Lucid Structure

It is convenient to choose a Lucid system which will remain fixed throughout the remainder of Sections 2 & 3. Note that the choices for alphabet, structure, variables and program are arbitrary and without loss of generality. Results will be valid for any program in any structure. Any notation or concept which is used here but not defined is exactly as in [1].

Suppose then that the following is fixed:

i) A standard Lucid alphabet $\Sigma$.

ii) A standard $\Sigma$-structure S, whose Comp(S) structure is C.

iii) $F = \{\text{first}, \text{next}, \text{fby}, \text{latest}, \text{latest}^{-1}\}$

iv) E is the set of $(\Sigma \cup F)$-terms, quantifier free with no occurrence of $=$.

v) $\mathbb{N}$ is the natural numbers.

$\mathbb{N}^+$ is the set of non-empty words over $\mathbb{N}$.

$\mathbb{N}^- = \mathbb{N}-\{0\}$.

vi) G is the set of operation symbols in $\Sigma$.

vii) V is a set of variables.

viii) P is a program over V such that for $v \in V$ there is an equation $v = \phi_v$ in P with $\phi_v \in E$.

ix) $\langle\omega,\rho\rangle \in V \times N^+$ is an identified variable-time pair whose solution is sought.

x) $\Phi: E \rightarrow E$ simultaneously substitutes $\phi_v$ for each occurrence of each variable $v \in V$ in a term. $\Phi$ will not be further formalised and its obvious properties will be assumed.

xi) Let $I$ be the minimal C-interpration satisfying P. That is $I$ is the solution to P.

Remark The Lucid program under consideration has been restricted from the class of programs in [1] in two trivial ways. Firstly, asa $\notin$ F. This causes no loss of generality since all asa can be eliminated from any program (at the expense of introducing variables "who depend on their own future"). For a proof of this see the appendix. Secondly, there is no input variable and we talk simply of "the solution to the program". Again there is no loss of generality since input could be simulated by including an equation "input = ..." in the program.

## 2.2 Extending Finite Sequences to Infinite Sequences

The formal semantics of Lucid makes use of infinite sequences of natural numbers. Only finite sequences will appear in the operational semantics. In order to be able to relate the two we introduce the following function. Let $[\ ]: N^+ \rightarrow N^N$ such that $[t_0 t_1 \ldots t_n] = t_0 t_1 \ldots t_n t_n t_n \ldots$ where $[\ ](x)$ is written $[x]$. Note that $[\ ]$ is many-one. Thus a finite sequence can be thought of as representing the infinite sequence obtained by repeating its last member.

We now simplify the definition of the Lucid operators in F by generalising their properties. For $f \in F$ define $\tilde{f}_N : N^N \to N^-$ and $\mathring{f}_N : N^N \to N^N$ such that for $\alpha_1, \alpha_2 \ldots \alpha_n \in U_C$ and $t \in N^N$

$$f_C(\alpha_1, \alpha_2 \ldots \alpha_n)_t = {\alpha_{\tilde{f}_N(t)}}_{\mathring{f}_N(t)}$$

This is a generalisation of the definition of $f \in \{\mathrm{first}, \mathrm{next}, \mathrm{fby}, \mathrm{latest}, \mathrm{latest}^{-1}\}$ where $\tilde{f}_N$ takes t and selects one of the $\alpha_i$ and $\mathring{f}_N$ takes t and yields a new time.

For each $f \in F$, $\tilde{f}_N$ and $\mathring{f}_N$ are determined by the definitions of these functions in [1]. We now wish to define two further auxiliary functions. $\tilde{f}_+ : N^+ \to N^=$ and $\mathring{f}_+ : N^+ \to N^+$, to be compatible finite versions of the above, such that

$$\tilde{f}_+(t) = \tilde{f}_N([t])$$

and $$[\mathring{f}_+(t)] = \mathring{f}_N([t])$$

for all $t \in N^+$.

The reader may assure himself that the following definitions satisfy the above constraints. For $t = t_0 t_1 t_2 \ldots \in N^N$ and $t' = t'_0 t'_1 t'_2 \ldots t'_n \in N^+$ let

$$\begin{array}{llll}
\widetilde{\mathrm{first}}_N(t) & = 1 & \widetilde{\mathrm{first}}_+(t') & = 1 \\
\mathring{\mathrm{first}}_N(t) & = 0t_1t_2\ldots & \mathring{\mathrm{first}}_+(t') & = 0t'_1t'_2\ldots t'_n \\
\widetilde{\mathrm{next}}_N(t) & = 1 & \widetilde{\mathrm{next}}_+(t') & = 1 \\
\mathring{\mathrm{next}}_N(t) & = t_0{+}1t_1t_2\ldots & \mathring{\mathrm{next}}_+(t') & = t'_0{+}1t'_1t'_2\ldots t_n \\
\widetilde{\mathrm{latest}}_N(t) & = 1 & \widetilde{\mathrm{latest}}_+(t') & = 1 \\
\mathring{\mathrm{latest}}_N(t) & = t_1t_2t_3\ldots & \mathring{\mathrm{latest}}_+(t') & = \begin{cases} t'_1t'_2\ldots t'_n & n > 0 \\ t'_0 & n = 0 \end{cases}
\end{array}$$

$$\widetilde{\text{latest}}_N^{-1}(t) = 1 \qquad \widetilde{\text{latest}}_+^{-1}(t') = 1$$

$$\mathring{\text{latest}}_N^{-1}(t) = 0t_0t_1t_2\ldots \qquad \mathring{\text{latest}}_+^{-1}(t') = 0t'_0t'_1\ldots t'_n$$

$$\widetilde{\text{fby}}_N(t) = \begin{cases} 1 & t_0 = 0 \\ 2 & t_0 > 0 \end{cases} \qquad \widetilde{\text{fby}}_+(t') = \begin{cases} 1 & t'_0 = 0 \\ 2 & t'_0 > 0. \end{cases}$$

$$\mathring{\text{fby}}_N(t) = \begin{cases} 0t_1t_2\ldots & t_0 = 0 \\ t_0\text{-}1t_1t_2\ldots & t_0 > 0 \end{cases} \qquad \mathring{\text{fby}}_+(t') = \begin{cases} 0t'_1t'_2\ldots t'_n & t'_0 = 0 \\ t'_0\text{-}1t'_1t'_2\ldots t'_n & t'_0 > 0 \end{cases}$$

## 2.3 Computing the Solution by Syntactic Substitution

We show here that the solution to the program can be computed by repeated syntactic substitution of terms for variables. We obtain a sequence which starts with the variable whose value is sought, and interpret the growing terms with all variables undefined. The details are as follows.

The Kleene Recursion Theorem [4] gives a construction for the minimal solution of P. If $\bot$ is the C-interpretation which is everywhere $\bot_C$ and $\tau:(V \to U_C) \to (V \to U_C)$ such that for $\langle v,t\rangle \in V{\times}N^N$ and C-interpretation I, $v_{\tau(I)_t} = |\phi_v|_{I_t}$, then $\mathcal{I}$, the minimal solution of P, is given by the least upper bound of $\{\tau^i(\bot):i \geq 0\}$. Since the underlying ordering, $\sqsubseteq$, is flat ($x \sqsubseteq y \Rightarrow x = \bot$ or $x = y$), it follows that

$$\omega_{\mathcal{I}[\![\rho]\!]} \neq \bot \Rightarrow (\exists k \geq 0) \quad \omega_{\tau^k(\bot)[\rho]} = \omega_{\mathcal{I}[\rho]}$$

[The ambiguous use of $\bot$,$\sqsubseteq$ etc. should lead to no confusion.]

By means of the following lemma, theorem and corollary we establish that the solution can also be computed by applications of $\Phi$ to $\omega$.

Lemma 2.3.0 For C-iterpretation I and $\sigma \in E$

$$|\Phi(\sigma)|_I = |\sigma|_{\tau(I)}$$

Proof by structural induction on $\sigma$.

Basis: $\sigma \in V \Rightarrow (\forall t \in N^N)|\Phi(\sigma)|_{I_t} = |\phi_\sigma|_{I_t} = \sigma_{\tau(I)_t} = |\sigma|_{\tau(I)_t}$

$$\Rightarrow |\Phi(\sigma)|_I = |\sigma|_{\tau(I)}$$

Induction step: Assume for all $\alpha_i$ with $\sigma = g(\alpha_1,\alpha_2,\ldots,\alpha_n)$ and $g \in F \cup G$

$$\begin{aligned} |\Phi(\sigma)|_I &= |g(\Phi(\alpha_1),\ldots\Phi(\alpha_n))|_I \\ &= g_C(|\Phi(\alpha_1)|_I,\ldots|\Phi(\alpha_n)|_I) \\ &= g_C(|\alpha_1|_{\tau(I)},\ldots|\alpha_n|_{\tau(I)}) && \text{by hypothesis} \\ &= |g(\alpha_1,\ldots,\alpha_n)|_{\tau(I)} \\ &= |\sigma|_{\tau(I)}. \end{aligned}$$

□

Theorem 2.3.1 For $0 \leq j \leq k$, $|\Phi^j(\omega)|_{\tau^{k-j}(\perp)} = \omega_{\tau^k(\perp)}$.

Proof by induction on j.

Basis: Consider $j = 0$, the case is trivial.

Induction step: Assume for $j = p < k$ and consider $j = p+1$

$$\begin{aligned} |\Phi^{p+1}(\omega)|_{\tau^{k-p-1}(\perp)} &= |\Phi\cdot\Phi^p(\omega)|_{\tau^{k-p-1}(\perp)} \\ &= |\Phi^p(\omega)|_{\tau^{k-p}(\perp)} && \text{by lemma 2.3.0} \\ &= \omega_{\tau^k(\perp)} && \text{by hypothesis} \end{aligned}$$

□

Taking k = j in this theorem we have $|\Phi^k(\omega)|_\perp = \omega_{\tau^k(\perp)}$, and combining this with the earlier construction for the solution

$$\omega_{\mathcal{I}[\rho]} \neq \perp \Rightarrow (\exists k \geq 0) \quad \omega_{\tau^k(\perp)[\rho]} = \omega_{\mathcal{I}[\rho]}$$

we obtain

Corollary 2.3.2

$$\omega_{\mathcal{I}[\rho]} \neq \perp \Rightarrow (\exists k \geq 0) \quad |\Phi^k(\omega)|_{\perp[\rho]} = \omega_{\mathcal{I}[\rho]}$$

## 2.4 Memories and the functions "eval" and "req"

We now introduce some objects that will form the building blocks of the operational semantics to follow. Memories correspond to interpretations of the variables of the program, but using only finitely many time parameters. [And those memories that are required in the operational semantics will only be defined at finitely many places. There is therefore no problem in representing memories.] Eval is a function which evaluates a term at a given time and with respect to a given memory. Its value is the interpretation of the term for the time, when the variables in the term are interpreted with the values associated with them in the memory. For a term that is undefined under eval, req is a function which yields a finite set of variable-time pairs which are undefined in the memory but whose value could influence the result of eval if they were defined. This is now made formal.

A function $M: V \times N^+ \to U_S$ is a memory. For any memory M, define $\text{eval}_M: E \times N^+ \to U_S$ inductively as follows:

i) $\mathrm{eval}_M(v,t) = M(v,t)$ for $\langle v,t\rangle \in V \times N^+$

ii) $\mathrm{eval}_M(g(\alpha_1,\alpha_2,\ldots,\alpha_n),t) = g_S(\mathrm{eval}_M(\alpha_1,t),\ldots\mathrm{eval}_M(\alpha_n,t))$ for $\langle g,t\rangle \in G \times N^+$

iii) $\mathrm{eval}_M(f(\alpha_1,\alpha_2,\ldots,\alpha_n),t) = \mathrm{eval}_M(\alpha_{\tilde{f}_+(t)},\mathring{f}_+(t))$ for $\langle f,t\rangle \in F \times N^+$

A memory M is said to be <u>normal</u> iff $M(v,t) \sqsubseteq \mathrm{eval}_M(\phi_v,t)$ for all $\langle v,t\rangle \in V \times N^+$. To motivate this definition observe that for normal memories a variable is less defined than its corresponding term (on the right hand side of its defining equation). Memories that appear in the operational semantics will all be normal because variables will only be assigned values resulting from the evaluation of their corresponding terms.

For any memory M define $\mathrm{req}_M: E \times N^+ \to 2^{V\times N^+}$ as follows:

i) If $\mathrm{eval}_M(\sigma,t) \neq \perp$ then $\mathrm{req}_M(\sigma,t) = \{\ \}$ for $\langle\sigma,t\rangle \in E \times N^+$

otherwise

ii) $\mathrm{req}_M(v,t) = \{\langle v,t\rangle\}$ for $\langle v,t\rangle \in V \times N^+$

iii) $\mathrm{req}_M(g(\alpha_1,\alpha_2,\ldots,\alpha_n),t) = \bigcup_{i=1}^{n} \mathrm{req}_M(\alpha_i,t)$ for $\langle g,t\rangle \in G \times N^+$

iv) $\mathrm{req}_M(f(\alpha_1,\alpha_2,\ldots,\alpha_n),t) = \mathrm{req}_M(\alpha_{\tilde{f}_+(t)},\mathring{f}_+(t))$ for $\langle f,t\rangle \in F \times N^+$

For a memory M and $\Gamma \subseteq V \times N^+$ define <u>M updated by Γ</u>, denoted $M_\Gamma$, to be the memory

$$M_\Gamma(v,t) = \begin{cases} M(v,t) & \text{if } \langle v,t\rangle \notin \Gamma \\ \mathrm{eval}_M(\phi_v,t) & \text{if } \langle v,t\rangle \in \Gamma \end{cases}$$

For normal memories, memory updates assign more defined values to variables by evaluating the corresponding terms. It will be this process that enables the definition of a sequence of memories which form better approximations to the solution to the program.

There are a number of obvious consequences of these definitions which are established as a series of propositions.

Proposition 2.4.0 For $\langle\sigma,t\rangle \in E \times N^+$ and normal M,

$$\mathrm{eval}_M(\sigma,t) \sqsubseteq \mathrm{eval}_M(\Phi(\sigma),t)$$

Proof by structural induction on $\sigma$.

Basis: $\sigma \in V \Rightarrow \mathrm{eval}_M(\sigma,t) = M(\sigma,t) \sqsubseteq \mathrm{eval}_M(\phi_\sigma,t) = \mathrm{eval}_M(\Phi(\sigma),t)$

Induction step: Case I: $\sigma = g(\alpha_1,\alpha_2,\ldots,\alpha_n)$, $g \in G$

$$\begin{aligned}\mathrm{eval}_M(\sigma,t) &= \mathrm{eval}_M(g(\alpha_1,\alpha_2,\ldots,\alpha_n),t)\\ &= g_S(\mathrm{eval}_M(\alpha_1,t),\ldots\mathrm{eval}_M(\alpha_n,t))\\ &\sqsubseteq g_S(\mathrm{eval}_M(\Phi(\alpha_1),t),\ldots\mathrm{eval}_M(\Phi(\alpha_n),t))\\ &\quad\text{by induction hypothesis and monotonicity of } g_S\\ &= \mathrm{eval}_M(\Phi(g(\alpha_1,\alpha_2,\ldots,\alpha_n)),t)\\ &= \mathrm{eval}_M(\Phi(\sigma),t)\end{aligned}$$

Case II: $\sigma = f(\alpha_1,\alpha_2,\ldots,\alpha_n)$, $f \in F$

$$\begin{aligned}\mathrm{eval}_M(\sigma,t) &= \mathrm{eval}_M(f(\alpha_1,\alpha_2,\ldots,\alpha_n),t)\\ &= \mathrm{eval}_M(\alpha_{\tilde{f}_+(t)},\mathring{f}_+(t))\\ &\sqsubseteq \mathrm{eval}_M(\Phi(\alpha_{\tilde{f}_+(t)}),\mathring{f}_+(t)) \quad \text{by induction hypothesis}\\ &= \mathrm{eval}_M(f(\Phi(\alpha_1),\ldots\Phi(\alpha_{\tilde{f}_+(t)}),\ldots\Phi(\alpha_n)),t)\\ &= \mathrm{eval}_M(\Phi(f(\alpha_1,\alpha_2\ldots\alpha_n)),t)\\ &= \mathrm{eval}_M(\Phi(\sigma),t).\end{aligned}$$

$\square$

We extend $\sqsubseteq$ (ambiguously) to memories in the usual fashion. For memories $M_1$ and $M_2$, $M_1 \sqsubseteq M_2$ iff $M_1(v,t) \sqsubseteq M_2(v,t)$ for all $\langle v,t\rangle \in V \times N^+$.

<u>Proposition 2.4.1</u> For $\langle\sigma,t\rangle \in E \times N^+$ and normal $M,M'$

if $M \sqsubseteq M'$ then i) $\mathrm{eval}_M(\sigma,t) \sqsubseteq \mathrm{eval}_{M'}(\sigma,t)$

ii) $\mathrm{req}_{M'}(\sigma,t) \subseteq \mathrm{req}_M(\sigma,t)$

<u>Proof</u> by structural induction on $\sigma$.

Basis: Consider $\sigma \in V$

$$\mathrm{eval}_M(\sigma,t) = M(\sigma,t) \sqsubseteq M'(\sigma,t) = \mathrm{eval}_{M'}(\sigma,t) \Rightarrow \text{i)}$$

$$\left.\begin{array}{l} M'(\sigma,t) = \perp \Rightarrow M(\sigma,t) = \perp \Rightarrow \mathrm{req}_{M'}(\sigma,t) = \{\langle\sigma,t\rangle\} = \mathrm{req}_M(\sigma,t) \\ M'(\sigma,t) \neq \perp \Rightarrow \mathrm{req}_{M'}(\sigma,t) = \{\ \} \subseteq \mathrm{req}_M(\sigma,t) \end{array}\right\} \Rightarrow \text{ii)}$$

Induction step: Case I: $\sigma = g(\alpha_1,\alpha_2,\ldots,\alpha_n)$, $g \in G$

$$\begin{aligned} \mathrm{eval}_M(\sigma,t) &= \mathrm{eval}_M(g(\alpha_1,\alpha_2,\ldots,\alpha_n),t) \\ &= g_S(\mathrm{eval}_M(\alpha_1,t),\ldots\mathrm{eval}_M(\alpha_n,t)) \\ &\sqsubseteq g_S(\mathrm{eval}_{M'}(\alpha_1,t),\ldots\mathrm{eval}_{M'}(\alpha_n,t)) \quad \text{by induction hypothesis and monotonicity of } g_S. \\ &= \mathrm{eval}_{M'}(g(\alpha_1,\alpha_2,\ldots,\alpha_n),t) \\ &= \mathrm{eval}_{M'}(\sigma,t) \end{aligned}$$

This establishes i). ii) is again taken in two parts

$$\begin{aligned} \mathrm{eval}_{M'}(\sigma,t) = \perp \Rightarrow \mathrm{req}_{M'}(\sigma,t) &= \bigcup_{i=1}^{n} \mathrm{req}_{M'}(\alpha_i,t) \\ &\subseteq \bigcup_{i=1}^{n} \mathrm{req}_M(\alpha_i,t) \quad \text{by hypothesis} \\ &= \mathrm{req}_M(\sigma,t) \end{aligned}$$

$$\mathrm{eval}_{M'}(\sigma,t) \neq \perp \Rightarrow \mathrm{req}_{M'}(\sigma,t) = \{\ \} \subseteq \mathrm{req}_M(\sigma,t)$$

Case II: $\sigma = f(\alpha_1,\alpha_2,\ldots,\alpha_n)$, $f \in F$

$$\begin{aligned}
\mathrm{eval}_M(\sigma,t) &= \mathrm{eval}_M(f(\alpha_1,\alpha_2,\ldots,\alpha_n),t)\\
&= \mathrm{eval}_M(\alpha_{\tilde{f}_+(t)},\overset{\circ}{f}_+(t))\\
&\sqsubseteq \mathrm{eval}_{M'}(\alpha_{\tilde{f}_+(t)},\overset{\circ}{f}_+(t)) \qquad \text{by induction hypothesis}\\
&= \mathrm{eval}_{M'}(f(\alpha_1,\alpha_2,\ldots,\alpha_n),t)\\
&= \mathrm{eval}_{M'}(\sigma,t).
\end{aligned}$$

This establishes i).

$$\begin{aligned}
\mathrm{req}_{M'}(\sigma,t) &= \mathrm{req}_{M'}(f(\alpha_1,\alpha_2,\ldots,\alpha_n),t)\\
&= \mathrm{req}_{M'}(\alpha_{\tilde{f}_+(t)},\overset{\circ}{f}_+(t))\\
&\subseteq \mathrm{req}_{M}(\alpha_{\tilde{f}_+(t)},\overset{\circ}{f}_+(t)) \qquad \text{by induction hypothesis}\\
&= \mathrm{req}_M(f(\alpha_1,\alpha_2,\ldots,\alpha_n),t)\\
&= \mathrm{req}_M(\sigma,t)
\end{aligned}$$

This establishes ii). $\square$

<u>Proposition 2.4.2</u> For normal $M$ and $\Gamma \subseteq V \times N^+$

i) $M \sqsubseteq M_\Gamma$

ii) $M_\Gamma$ is normal.

<u>Proof</u> Consider any $\langle v,t\rangle \in V$

$$\left.\begin{aligned}
\langle v,t\rangle \in \Gamma &\Rightarrow M(v,t) \sqsubseteq \mathrm{eval}_M(\phi_v,t) = M_\Gamma(v,t)\\
\langle v,t\rangle \notin \Gamma &\Rightarrow M(v,t) = M_\Gamma(v,t)
\end{aligned}\right\} \Rightarrow \text{i)}$$

$$\begin{aligned}
\langle v,t\rangle \in \Gamma \Rightarrow M_\Gamma(v,t) &= \mathrm{eval}_M(\phi_v,t)\\
&\sqsubseteq \mathrm{eval}_{M_\Gamma}(\phi_v,t) \text{ by part i) of this Proposition}\\
&\qquad \text{and Proposition 2.4.1.}
\end{aligned}$$

$$\begin{aligned}\langle v,t\rangle \notin \Gamma \Rightarrow M_\Gamma(v,t) &= M(v,t)\\ &\sqsubseteq \mathrm{eval}_M(\phi_v,t)\\ &\sqsubseteq \mathrm{eval}_{M_\Gamma}(\phi_v,t) \text{ by Proposition 2.4.1 again.}\end{aligned}$$

These two cases establish ii). □

Proposition 2.4.3 For normal M and $\Gamma_1 \subseteq \Gamma_2 \subseteq V \times N^+$,

$$M_{\Gamma_1} \sqsubseteq M_{\Gamma_2}.$$

Proof Consider any $\langle v,t\rangle \in V \times N^+$.

$$\begin{aligned}\langle v,t\rangle \in \Gamma_1 &\Rightarrow \langle v,t\rangle \in \Gamma_2\\ &\Rightarrow M_{\Gamma_1}(v,t) = \mathrm{eval}_M(\phi_v,t) = M_{\Gamma_2}(v,t)\\ \langle v,t\rangle \notin \Gamma_1 &\Rightarrow M_{\Gamma_1}(v,t) = M(v,t)\\ &\qquad\qquad \sqsubseteq M_{\Gamma_2}(v,t) \text{ by Proposition 2.4.2} \quad □\end{aligned}$$

We now have the important lemma of section 2.4. We know that syntactic substitution of terms for variables leads to better approximations of the solution to the program. Lemma 2.4.4 shows that memory update of the variables determined by req is as good as syntactic substitution. A little consideration should convince the reader that this is exactly what would be expected.

Lemma 2.4.4 For normal M and $\langle\sigma,t\rangle \in E \times N^+$

$$\mathrm{eval}_M(\Phi(\sigma),t) \sqsubseteq \mathrm{eval}_{M_{\mathrm{req}_M(\sigma,t)}}(\sigma,t)$$

Proof by structural induction on $\sigma$.

Basis: for $\sigma \in V$ consider two cases a) $M(\sigma,t) = \perp$

b) $M(\sigma,t) \neq \perp$

a) $\mathrm{eval}_M(\Phi(\sigma),t) = \mathrm{eval}_M(\phi_\sigma,t)$

$= \mathrm{eval}_{M_{\{\langle\sigma,t\rangle\}}}(\sigma,t)$

$= \mathrm{eval}_{M_{\mathrm{req}_M(\sigma,t)}}(\sigma,t)$

b) $\mathrm{eval}_M(\Phi(\sigma),t) = \mathrm{eval}_M(\phi_\sigma,t)$

$= M(\sigma,t)$ M normal and $\sqsubseteq$ flat

$= \mathrm{eval}_M(\sigma,t)$

$= \mathrm{eval}_{M_{\{\}}}(\sigma,t)$

$= \mathrm{eval}_{M_{\mathrm{req}_M(\sigma,t)}}(\sigma,t)$

This completes the basis.

Induction step: Case I: $\sigma = g(\alpha_1,\alpha_2,\ldots,\alpha_n)$, $g \in G$. There are two subcases to consider I.a) $\mathrm{eval}_M(\sigma,t) = \perp$

I.b) $\mathrm{eval}_M(\sigma,t) \neq \perp$

I.a) $\mathrm{eval}_M(\Phi(\sigma),t) = \mathrm{eval}_M(g(\Phi(\alpha_1),\ldots\Phi(\alpha_n)),t)$

$= g_S(\mathrm{eval}_M(\Phi(\alpha_1),t),\ldots\mathrm{eval}_M(\Phi(\alpha_n),t))$

$\sqsubseteq g_S(\mathrm{eval}_{M_{\mathrm{req}_M(\alpha_1,t)}}(\alpha_1,t),\ldots\mathrm{eval}_{M_{\mathrm{req}_M(\alpha_n,t)}}(\alpha_n,t))$

by hypothesis and monotonicity of $g_S$

$\sqsubseteq g_S(\mathrm{eval}_{M_{\mathrm{req}_M(\sigma,t)}}(\alpha_1,t),\ldots\mathrm{eval}_{M_{\mathrm{req}_M(\sigma,t)}}(\alpha_n,t))$

by definition of req, Propositions 2.4.3 and 2.4.1 and monotonicity of $g_S$

$= \mathrm{eval}_{M_{\mathrm{req}_M(\sigma,t)}}(g(\alpha_1,\alpha_2,\ldots,\alpha_n),t)$

$= \mathrm{eval}_{M_{\mathrm{req}_M(\sigma,t)}}(\sigma,t)$

I.b) $\quad \mathrm{eval}_M(\sigma,t) \neq \perp$

$\mathrm{eval}_M(\sigma,t) \sqsubseteq \mathrm{eval}_M(\Phi(\sigma),t)$ by Proposition 2.4.0

$\mathrm{eval}_M(\sigma,t) \sqsubseteq \mathrm{eval}_{M_{\mathrm{req}_M(\sigma,t)}}(\sigma,t)$ by Proposition 2.4.1 and 2.4.2

From these three facts and the flatness of $\sqsubseteq$ it follows that

$$\mathrm{eval}_M(\Phi(\sigma),t) \sqsubseteq \mathrm{eval}_{M_{\mathrm{req}_M(\sigma,t)}}(\sigma,t)$$

This completes case I of the Induction Step.

Case II: $\sigma = f(\alpha_1,\alpha_2,\ldots,\alpha_n)$, $f \in F$

$$\begin{aligned}
\mathrm{eval}_M(\Phi(\sigma),t) &= \mathrm{eval}_M(f(\Phi(\alpha_1),\ldots,\Phi(\alpha_n)),t)\\
&= \mathrm{eval}_M(\Phi(\alpha_{\tilde{f}_+(t)}),\mathring{f}_+(t))\\
&\sqsubseteq \mathrm{eval}_{M_{\mathrm{req}_M(\alpha_{\tilde{f}_+(t)},\mathring{f}_+(t))}}(\alpha_{\tilde{f}_+(t)},\mathring{f}_+(t)) \text{ by hypothesis}\\
&= \mathrm{eval}_{M_{\mathrm{req}_M(f(\alpha_1,\alpha_2,\ldots,\alpha_n),t)}}(f(\alpha_1,\alpha_2,\ldots,\alpha_n),t)\\
&= \mathrm{eval}_{M_{\mathrm{req}_M(\sigma,t)}}(\sigma,t)
\end{aligned}$$

This completes the proof of Lemma 2.4.4. □

The next lemma is the formal statement of a somewhat obvious fact. It says that $\mathrm{req}_M(\Phi(\sigma),t)$ can be computed by two levels of application of req. Firstly, req is applied to $\sigma$, and then again to the terms corresponding to the variables returned by the first application.

Lemma 2.4.5 If i) M is normal

ii) $\langle\sigma,t\rangle \in E \times N^+$

iii) $\langle u,s\rangle \in \mathrm{req}_M(\Phi(\sigma),t)$

then $(\exists\langle u',s'\rangle \in \mathrm{req}_M(\sigma,t))\ \langle u,s\rangle \in \mathrm{req}_M(\phi_{u'},s')$.

<u>Proof</u> by structural induction on $\sigma$.

Basis: for $\sigma \in V$ choose $\langle u',s'\rangle = \langle\sigma,t\rangle$.

Induction step: Case I: $\sigma = g(\alpha_1,\alpha_2,\ldots,\alpha_n)$, $g \in G$

$$\begin{aligned}\langle u,s\rangle \in \mathrm{req}_M(\Phi(\sigma),t) &\Rightarrow \langle u,s\rangle \in \bigcup_{i=1}^{n} \mathrm{req}_M(\Phi(\alpha_i),t)\\ &\Rightarrow \langle u,s\rangle \in \mathrm{req}_M(\Phi(\alpha_j),t) \quad \text{for some } j,\ 1 \le j \le n\\ &\Rightarrow (\exists\ \langle u',s'\rangle \in \mathrm{req}_M(\alpha_j,t))\ \langle u,s\rangle \in \mathrm{req}_M(\phi_{u'},s')\\ &\qquad \text{by hypothesis}\end{aligned}$$

But
$$\begin{aligned}\mathrm{req}_M(\Phi(\sigma),t) \neq \{\} &\Rightarrow \mathrm{eval}_M(\Phi(\sigma),t) = \perp\\ &\Rightarrow \mathrm{eval}_M(\sigma,t) = \perp \quad \text{by Proposition 2.4.0}\\ &\Rightarrow \mathrm{req}_M(\sigma,t) = \bigcup_{i=1}^{n} \mathrm{req}_M(\alpha_i,t)\\ &\Rightarrow \mathrm{req}_M(\alpha_j,t) \subseteq \mathrm{req}_M(\sigma,t)\end{aligned}$$

Combining the above

$$(\exists\langle u',s'\rangle \in \mathrm{req}_M(\sigma,t))\ \langle u,s\rangle \in \mathrm{req}_M(\phi_{u'},s')$$

Case II: $\sigma = f(\alpha_1,\alpha_2,\ldots,\alpha_n)$, $f \in F$

$$\begin{aligned}\langle u,s\rangle \in \mathrm{req}_M(\Phi(\sigma),t) &\Rightarrow \langle u,s\rangle \in \mathrm{req}_M(\Phi(\alpha_{\tilde{f}_+(t)}),\mathring{f}_+(t))\\ &\Rightarrow (\exists\langle u',s'\rangle \in \mathrm{req}_M(\alpha_{\tilde{f}_+(t)},\mathring{f}_+(t)))\\ &\qquad \langle u,s\rangle \in \mathrm{req}_M(\phi_{u'},s') \quad \text{by hypothesis}\\ &\Rightarrow (\exists\langle u',s'\rangle \in \mathrm{req}_M(f(\alpha_1,\alpha_2,\ldots,\alpha_n),t))\\ &\qquad \langle u,s\rangle \in \mathrm{req}_M(\phi_{u'},s')\\ &\Rightarrow (\exists\langle u',s'\rangle \in \mathrm{req}_M(\sigma,t))\langle u,s\rangle \in \mathrm{req}\ (\phi_{u'},s').\end{aligned}$$

□

## 2.5 The correctness of $\text{eval}_\perp$

We have not yet shown formally that the evaluation function for terms is correct. We wish to have eval behave consistently with the definition of the interpretation of terms. The following theorem is not the strongest formulation possible of the correctness of eval but it will be sufficient here. We use $\perp$ ambiguously to denote both the undefined C-interpretation and the undefined memory.

Theorem 2.5.0 For $\langle\sigma,t\rangle \in E \times N^+$

$$|\sigma|_{\perp[t]} \sqsubseteq \text{eval}_\perp(\sigma,t)$$

Proof by structural induction on $\sigma$.

Basis: For $\sigma \in V$ the result is trivial.

Induction step: Case I: $\sigma = g(\alpha_1,\alpha_2,\ldots,\alpha_n)$, $g \in G$.

By hypothesis $|\alpha_i|_{\perp[t]} \sqsubseteq \text{eval}_\perp(\alpha_i,t)$ for $1 \leq i \leq n$

then by monotonicity of $g_S$

$$g_S(|\alpha_1|_{\perp[t]},\ldots,|\alpha_n|_{\perp[t]}) \sqsubseteq g_S(\text{eval}_\perp(\alpha_1,t),\ldots,\text{eval}_\perp(\alpha_n,t))$$

$$\Rightarrow |\sigma|_{\perp[t]} \sqsubseteq \text{eval}_\perp(\sigma,t)$$

Case II: $\sigma = f(\alpha_1,\alpha_2,\ldots,\alpha_n)$, $f \in F$

$$|\sigma|_{\perp[t]} = |f(\alpha_1,\alpha_2,\ldots,\alpha_n)|_{\perp[t]}$$

$$= f_C(|\alpha_1|_\perp,\ldots,|\alpha_n|_\perp)[t]$$

$$= |\alpha_{\tilde{f}_N([t])}|_{\perp_{\mathring{f}_N([t])}} \qquad \text{by definition of } \tilde{f}_N \text{ and } \mathring{f}_N$$

$$
\begin{aligned}
&= \left|\alpha_{\tilde{f}_+(t)}\right|_{\perp[\mathring{f}_+(t)]} \\
&\sqsubseteq \text{eval}_\perp(\alpha_{\tilde{f}_+(t)}, \mathring{f}_+(t)) \qquad \text{by hypothesis} \\
&= \text{eval}_\perp(f(\alpha_1, \alpha_2, \ldots, \alpha_n), t) \\
&= \text{eval}_\perp(\sigma, t). \qquad \Box
\end{aligned}
$$

An immediate consequence of Theorem 2.5.0 and Corollary 2.3.2 is

Corollary 2.5.1

$$\omega_{I[\rho]} \neq \perp \Rightarrow (\exists k \geq 0)\ \text{eval}_\perp(\Phi^k(\omega), \rho) = \omega_{I[\rho]} .$$

In other words, where the solution to the program is defined it can be computed by applying $\text{eval}_\perp$ to successive members of the $\Phi^i(\omega)$ sequence of terms until one delivers a defined result. We have not yet shown that applying $\text{eval}_{\perp\!\!\perp}$ to every member of this sequence is indeed an approximation to the solution. But Corollary 2.6.6 establishes that the operational semantics can never yield a result which is not an approximation to the solution.

## 2.6 Operational Semantics

We now have sufficient basis to define the operational semantics for Lucid. We do this by associating with the program a pair of sequences. (Repeating the observation from section 2.1, we note that dealing with a particular program in a particular structure leads to no loss of generality.) The "M-sequence" is a sequence of memories, each of which approximates the solution to the program. The "R-sequence" is a sequence of growing sets of variable-time pairs which have been "requested". That is, those variable-time pairs which are added to later members of the sequence are the variables whose values are required to compute values for earlier ones. We will show that when the solution to the program is defined for $\langle\omega,\rho\rangle$ the solution value will ultimately appear in the M-sequence, if the R-sequence initially contains $\langle\omega,\rho\rangle$. And further, the memories in the M-sequence are all approximations to the solution of the program.

Formally, we define two sequences $\langle M^{(i)}\rangle$, $\langle R^{(i)}\rangle$ for $i = 0,1,2,\ldots$; each $M^{(i)}$ is a memory and each $R^{(i)} \subseteq V \times N^{+}$. The sequences are defined inductively as follows:

$$R^{(0)} = \{\langle\omega,\rho\rangle\}$$

$$M^{(0)} = \perp \qquad \text{(the everywhere undefined memory)}$$

$$R^{(i+1)} = R^{(i)} \cup \bigcup_{\langle v,t\rangle\in R^{(i)}} \mathrm{req}_{M^{(i)}}(\phi_v,t)$$

$$M^{(i+1)} = M^{(i)}_{R^{(i)}}$$

We have the following immediate properties of these sequences.

<u>Proposition 2.6.0</u>

i) $i \leq j \Rightarrow M^{(i)} \sqsubseteq M^{(j)}$

ii) $i \leq j \Rightarrow R^{(i)} \subseteq R^{(j)}$

iii) $i \geq 0 \Rightarrow M^{(i)}$ is normal

<u>Proof</u> ii) is obvious. i) and iii) follow by induction using Proposition 2.4.2.

□

The next lemma shows that as the $\Phi^{i}(\omega)$ term grows, the undefined variable-time pairs required for its evaluation are found in the R-sequence.

<u>Lemma 2.6.1</u> $\text{req}_{M^{(k)}}(\Phi^{k}(\omega),\rho) \subseteq R^{(k)}$ for $k \geq 0$

<u>Proof</u> by induction on k.

Basis: for $k = 0$ it is trivial.

Induction step: Assume for $k = p$ and consider $k = p+1$

$$\langle v,t\rangle \in \text{req}_{M^{(p+1)}}(\Phi^{p+1}(\omega),\rho) \Rightarrow \langle v,t\rangle \in \text{req}_{M^{(p+1)}}(\Phi(\Phi^{p}(\omega)),\rho)$$

$$\Rightarrow (\exists\langle u,s\rangle \in \text{req}_{M^{(p+1)}}(\Phi^{p}(\omega),\rho))\langle v,t\rangle \in \text{req}_{M^{(p+1)}}(\phi_{u},s)$$

by Lemma 2.4.5

$$\Rightarrow (\exists\langle u,s\rangle \in \text{req}_{M^{(p)}}(\Phi^{p}(\omega),\rho))\langle v,t\rangle \in \text{req}_{M^{(p)}}(\phi_{u},s)$$

by Propositions 2.6.0 and 2.4.1

$$\Rightarrow (\exists\langle u,s\rangle \in R^{(p)})\langle v,t\rangle \in \text{req}_{M^{(p)}}(\phi_{u},s) \quad \text{by hypothesis}$$

$$\Rightarrow \langle v,t\rangle \in R^{(p+1)}. \qquad \square$$

The next theorem presented is the central property of the operational semantics. It establishes an approximation which has on the left the evaluation of a term in the $\Phi^{i}(\omega)$ sequence and on the right the evaluation of an earlier term in the sequence but with respect to a memory that appears later in the M-sequence. In a sense, this says that the reduction of information about the program that results from choosing an earlier term is compensated for by the better approximation to the solution that the memory in the M-sequence provides.

<u>Theorem 2.6.2</u> $\text{eval}_{\perp}(\Phi^{k}(\omega),\rho) \sqsubseteq \text{eval}_{M^{(k+j)}}(\Phi^{k-j}(\omega),\rho)$ for $0 \leq j \leq k$

<u>Proof</u> by induction on j

Basis: $j = 0$

For $k \geq 0$, $M^{(k)}$ is normal and $\perp \sqsubseteq M^{(k)}$

$$\therefore\ \text{eval}_{\perp}(\Phi^{k}(\omega),\rho) \sqsubseteq \text{eval}_{M^{(k)}}(\Phi^{k}(\omega),\rho)$$

Induction step; Assume for $j = p < k \geq 0$ and consider $j = p+1$

$$\begin{aligned}
\mathrm{eval}_{\perp}(\Phi^{k}(\omega),\rho) &\sqsubseteq \mathrm{eval}_{M^{(k+p)}}(\Phi^{k-p}(\omega),\rho) && \text{by hypothesis} \\
&= \mathrm{eval}_{M^{(k+p)}}(\Phi(\Phi^{k-p-1}(\omega)),\rho) && p < k \\
&\sqsubseteq \mathrm{eval}_{M^{(k+p)}_{\mathrm{req}_{M^{(k+p)}}(\Phi^{k-p-1}(\omega),\rho)}}(\Phi^{k-p-1}(\omega),\rho) && \\
& && \text{by Lemma 2.4.4} \\
&\sqsubseteq \mathrm{eval}_{M^{(k+p)}_{\mathrm{req}_{M^{(k-p-1)}}(\Phi^{k-p-1}(\omega),\rho)}}(\Phi^{k-p-1}(\omega),\rho) && \\
& && \text{by Propositions 2.4.1 \& 2.6.0} \\
&\sqsubseteq \mathrm{eval}_{M^{(k+p)}_{R^{(k-p-1)}}}(\Phi^{k-p-1}(\omega),\rho) && \text{by Lemma 2.6.1 and Proposition 2.4.1} \\
&\sqsubseteq \mathrm{eval}_{M^{(k+p)}_{R^{(k+p)}}}(\Phi^{k-p-1}(\omega),\rho) && \text{by Propositions 2.6.0, 2.4.1 \& 2.4.3} \\
&= \mathrm{eval}_{M^{(k+p+1)}}(\Phi^{k-p-1}(\omega),\rho). && \square
\end{aligned}$$

The interesting case to consider for this theorem is when k=j and the term on the right is reduced to $\omega$

$$\mathrm{eval}_{\perp}(\Phi^{k}(\omega),\rho) \sqsubseteq \mathrm{eval}_{M^{(2k)}}(\omega,\rho) = M^{(2k)}(\omega,\rho)$$

This can be combined with Corollary 2.5.1 to give

<u>Corollary 2.6.3</u>

$$\omega_{I[\rho]} \neq \perp \Rightarrow (\exists k \geq 0)\ M^{(k)}(\omega,\rho) = \omega_{I[\rho]}.$$

We have therefore achieved the goal of proving that the M-sequence will eventually contain the solution to the program, when the solution is defined. It only remains to prove that whenever the M-sequence is defined then the value is that of the solution. This will exclude the possibility of the M-sequence giving a value when the solution to the program is undefined. The first step towards this is given by the next lemma which shows that when a memory approximates an interpretation then eval also approximates that interpretation.

Lemma 2.6.4 If I is any C-interpretation and M is any memory such that $M(v,t) \sqsubseteq v_{I_{[t]}}$ for all $\langle v,t\rangle \in V \times N^+$ then

$eval_M(\sigma,t) \sqsubseteq |\sigma|_{I_{[t]}}$ for all $\langle \sigma,t\rangle \in E \times N^+$.

Proof by structural induction on $\sigma$.

Basis: for $\sigma \in V$ it holds trivially.

Induction step: Case I: $\sigma = g(\alpha_1,\alpha_2 \ldots \alpha_n)$, $g \in G$

$eval_M(\alpha_i,t) \sqsubseteq |\alpha_i|_{I_{[t]}}$ $1 \le i \le n$ by hypothesis

$\Rightarrow g_S(eval_M(\alpha_1,t),..,eval_M(\alpha_n,t) \sqsubseteq g_S(|\alpha_1|_{I_{[t]}},..,|\alpha_n|_{I_{[t]}})$

by monotonicity of $g_S$

$\Rightarrow eval_M(g_S(\alpha_1,\alpha_2..,\alpha_n),t) \sqsubseteq |g(\alpha_1,\alpha_2..,\alpha_n)|_{I_{[t]}}$

$\Rightarrow eval_M(\sigma,t) \sqsubseteq |\sigma|_{I_{[t]}}$

Case II: $\sigma = f(\alpha_1,\alpha_2..,\alpha_n)$, $f \in F$

$eval_M(\sigma,t) = eval_M(f(\alpha_1,\alpha_2..,\alpha_n),t)$

$= eval_M(\alpha_{\tilde{f}_+(t)},\overset{\circ}{f}_+(t))$

$= eval_M(\alpha_{\tilde{f}_N([t])},\overset{\circ}{f}_+(t))$

$\sqsubseteq |\alpha_{\tilde{f}_N}([t])|_{I_{[\mathring{f}_+(t)]}}$ by hypothesis

$= |\alpha_{\tilde{f}_N}([t])|_{I_{\mathring{f}_N([t])}}$

$= f_C(|\alpha_1|_I,\ldots,|\alpha_n|_I)_{[t]}$

$= |f(\alpha_1,\alpha_2\ldots,\alpha_n)|_{I_{[t]}}$

$= |\sigma|_{I_{[t]}}$ □

We use this lemma to prove that an update of a memory which approximates the solution to the program is still an approximation to the solution.

<u>Theorem 2.6.5</u> If M is a memory such that $M(v,t) \sqsubseteq v_{\mathcal{I}_{[t]}}$ for all $\langle v,t\rangle \in V \times N^+$ then for any $\Gamma \subseteq V \times N^+$,

$M_\Gamma(v,t) \sqsubseteq v_{\mathcal{I}_{[t]}}$ for all $\langle v,t\rangle \in V \times N^+$

<u>Proof</u> Consider any $\Gamma$ and any $\langle v,t\rangle$

$\langle v,t\rangle \notin \Gamma \Rightarrow M_\Gamma(v,t) = M(v,t) \sqsubseteq v_{\mathcal{I}_{[t]}}$ as required.

$\langle v,t\rangle \in \Gamma \Rightarrow M_\Gamma(v,t) = \mathrm{eval}_M(\phi_v,t)$

$\sqsubseteq |\phi_v|_{\mathcal{I}_{[t]}}$ by lemma 2.6.4

$= |v|_{\mathcal{I}_{[t]}}$ $\mathcal{I}$ is a solution

$= v_{\mathcal{I}_{[t]}}$ □

Now since $M^{(0)}$ is clearly an approximation to $\mathcal{I}$, Theorem 2.6.5 can be applied inductively to the M-sequence to obtain

<u>Corollary 2.6.6</u>

$M^{(k)}(\omega,\rho) \sqsubseteq \omega_{\mathcal{I}_{[\rho]}}$ for $k \geq 0$

This completes this section on the operational semantics. It should be clear that implementation of this scheme presents no problem provided some means of computing the operations of G is available. It may also be evident that this method requires that substantial amounts of redundant work be performed. The fashion in which $R^{(i+1)}$ and $M^{(i+1)}$ are defined involves computing req and eval across the whole of $R^{(i)}$, whereas a more restricted set would be sufficient. A more sophisticated algorithm is presented in the next section which is much closer to that used for the implementation in AlgolW.

## 3. Informal Description of the Interpreter

In this section we present more sophisticated operational semantics for Lucid in the form of a very high level description of an interpreter for Lucid programs. No proof of correctness of this algorithm will be given. An intuitive understanding of the validity of this interpreter may be acquired by observing the correspondence between the actions of the interpreter and the operational semantics of the previous section. The algorithm is described by an Algol-like program in Fig.3.1. A formal description of this language will not be given. The tupling, set theoretic operations and where construct have obvious interpretations. The program is explained here line by line.

1. The three global variables Q, M and G are initialised. Q is a queue of variable-time pairs (it is of type $(V \times N^{+})^{*}$) and is initialised to be of length 1 and contain the sought variable-time pair, $\langle\omega,\rho\rangle$. The "left" of the string is the "head" of the queue. Q corresponds to the R-sequence of the previous semantics. It is a list of variable-time pairs to be computed. An important distinction is that Q (considered as a set) both

grows and shrinks, whereas the R-sequence always grows. M is a memory (it is of type $V \times N^+ \rightarrow U_S$) and is initialised to be the everywhere undefined function. M corresponds to the M-sequence. G is a directed graph of variable-time pairs (it is of type $2^{(V \times N^+) \times (V \times N^+)}$) and is initially empty. When G contains an edge $\langle a,b \rangle$ (where $a,b \in V \times N^+$) it means that the evaluation of a requires the value of b, i.e. that a depends on b. It is initially empty because nothing is known of the dependencies in the program at this point.

```
$\langle Q,M,G \rangle := \langle \langle \omega,\rho \rangle,\perp,\{\ \} \rangle$;                                          1
while $M(\omega,\rho) = \perp \wedge Q \neq \lambda$ do                                              2
begin $\langle Q,\langle v,t \rangle \rangle := \langle Q',x \rangle$ where $Q = xQ' \wedge x \in V \times N^+$;                       3
    $\langle e,r \rangle := \langle eval_M(\phi_v,t),req_M(\phi_v,t) \rangle$;                           4
    if $e = \perp$ then                                                        5
            begin for $u \in r$ do begin $G := G \cup \{\langle \langle v,t \rangle,u \rangle\}$;           6
                                     if $u \notin Q$ then $Q := Qu$          7
                               end                                  8
            end                                                        9
        else                                                          10
            begin $M(v,t) := e$;                                            11
              for $\langle u,\langle v,t \rangle \rangle \in G$ do if $M(u) = \perp \wedge u \notin Q$ then $Q := Qu$   12
            end                                                       13
 end                                                                  14
```

Figure 3.1

2. The interpreter loops until one of two conditions is satisfied. The loop will terminate if $M(\omega,\rho) \neq \perp$ which means that the value for the sought variable has been found or if $Q = \lambda$ (i.e. Q is empty) which means that there are no outstanding requests for variables. On exit of this loop we will have $M(\omega,\rho) = \omega_{I[\rho]}$.

3. This assignment removes the first element of Q, assigning the variable component to v and the time component to t.

4. Eval and req are applied to the term corresponding to v using the current memory and time t. The results are saved in e and r. If $e = \perp$ then r will be the set of variable-time pairs needed to evaluate v at t.

5. A test on e decides whether eval failed and 6-9 are to be executed or eval succeeded in obtaining a value and 11-13 are to be executed.

6&7. Control is here only when $\mathrm{eval}_M(\phi_v,t) = \perp$. If $\phi_v$ cannot be evaluated with the current memory then the variable-time pairs of $\mathrm{req}_M(\phi_v,t)$ should be requested. Taking each member of $\mathrm{req}_M(\phi_v,t)$ an edge is added to G to record that the variable is needed for the computation of $\langle v,t\rangle$. Then if this variable is not already on the request queue (viewing Q as a set), it is added to the "far" end of Q. There would be no point in having two copies of a request for a variable. At this point $\langle v,t\rangle$ becomes "suspended". It does not appear in Q and will not return to Q until one of the variables requested at this point becomes defined.

11&12. Control is here only when $\mathrm{eval}_M(\phi_v,t) \neq \perp$. In this case the memory is "updated" at $\langle v,t\rangle$ with the computed value. Now when $\langle v,t\rangle \neq \langle\omega,\rho\rangle$ this value was computed to enable the computation of one or more other variables. It is therefore necessary to find all those variable-time pairs which depend on $\langle v,t\rangle$ (i.e. $\langle u,\langle v,t\rangle\rangle \in G$) and add to Q those which are still undefined and are not already in Q.

The while loop continues to request and define variables until a value for $\langle\omega,\rho\rangle$ is obtained.

This description is still far from an accurate portrayal of the implementation as written. The interpreter is approximately 1000 lines of AlgolW. It would be a major undertaking to give a detailed model of such a program. And since no mathematical analysis of the program has been attempted it would be unjustifiable to treat it in depth.

## 4. Conclusion

We have shown that Lucid can be implemented by means of a request driven interpreter. And indeed this approach has been used as the basis for an implementation of Lucid written in AlgolW for the IBM 360. May has also implemented Lucid by means of requests. In fact, his implementation extends the language to include arrays. However, he modified the language to eliminate the parallelism inherent in the original definition of $\vee$. He redefines $\perp \vee \underline{\text{true}}$ to be $\perp\!\!\perp$ and can then satisfy requests in a "depth first" fashion. Hoffmann has written a compiler for a subset of Lucid by eliminating the difficult constructions from the language.

The work of Farah is quite close to the work presented in this paper. But his is a theoretical implementation in Culik's linked forest manipulation system [5] and exploits the non-determinism of the method to handle the parallelism.

At the time of writing it is not known whether any of the above mentioned work is to be published.

## References

[1] E.A. Ashcroft and W.W. Wadge, "Lucid - A Formal System for Writing and Proving Programs", Tech. Report CS-75-01, Dept. of Computer Science, University of Waterloo, 1975 (to appear SIAM J. Computing).

[2] E.A. Ashcroft and W.W. Wadge, "Demystifying Program Proving: an Informal Introduction to Lucid", Tech. Report CS-75-02, Dept. of Computer Science, University of Waterloo, 1975.

[3] E.A. Ashcroft and W.W. Wadge, "Program Proving without Tears", IRIA Symposium on Proving and Improving Programs , G. Huet and G. Kahn (Editors), Colloques IRIA, 1975.

[4] S.C. Kleene, "Introduction to Metamathematics", Van Nostrand, 1952.

[5] K. Culik II, "A Model for the Formal Definition of Programming Languages", International J. Computer Math., Section A, vol.3, pp.315-345, 1973.

Case ii $(\forall i \geq 0)\ z_{I'_{iu}} = \underline{\text{false}}$

$\Rightarrow$ $w_{I'_{iu}} = w_{I'_{i+1\ u}}$ for $i \geq 0$ since I' satisfies (3)

$\Rightarrow$ $w_{I'_{iu}} = \perp$ for $i \geq 0$ since I' is minimal

$\Rightarrow$ $(\text{first}\ w_{I'})_t = w_{I'_{ou}} = \perp$

$\Rightarrow$ $x_{I'_t} = \perp$ since I' also satisfies (2)

$\Rightarrow$ $x_{I'_t} = (y_{I'}\ \text{asa}\ z_{I'})_t$ by definition of asa

Therefore I' satisfies (1).

This completes Part B and the proof. □

Remark Observe that, in general, programs $\pi$ and $\pi'$ of the above theorem do not have the same set of solutions. Consider the programs $\pi$ and $\pi'$ below

| | |
|---|---|
| | x = first w |
| x = 1 asa false | w = if false then 1 else next w |
| $\pi$ | $\pi'$ |

The only solution to $\pi$ is $x = \perp$. However, any quiescent value for x and w will satisfy $\pi'$. The minimal solutions do, of course, agree.

Case ii $\quad (\forall i{\ge}0)\ z_{I_{iu}} = \underline{\text{false}}$

$\Rightarrow \quad \bar{w}_{iu} = \bar{w}_{i+1\ u} \qquad$ for $i \ge 0$

$\Rightarrow \quad \bar{w}_{iu} = \perp \qquad$ for $i \ge 0$ since $\bar{w}$ is minimal

$\Rightarrow \quad \bar{w}_{ou} = \perp$

But $x_{I_t} = \perp$ by definition of asa and from (1). Therefore $x_{I_t} = \bar{w}_{ou} = (\text{first}\ \bar{w})_t$ as required.

This completes the two cases of Part A.

Part B $\quad$ We now verify that I' satisfies $\pi$. Again A is common and it is sufficient to show that $x_{I'}, y_{I'}, z_{I'}$ satisfy (1). That is, we prove

for all $t \in N^N$, $x_{I'_t} = (y_{I'}\ \text{asa}\ z_{I'})_t$

As before consider any $t \in N^N$ and let $t = t_0u$ where $t_0 \in N$ and $u \in N^N$. Again there are two cases

i) $\quad (\exists s{\ge}0)\ z_{I'_{su}} \ne \underline{\text{false}}$ and $0{\le}r{<}s \Rightarrow z_{I'_{ru}} = \underline{\text{false}}$

ii) $\quad (\forall i{\ge}0)\ z_{I'_{iu}} = \underline{\text{false}}$

Case i $\quad$ By an argument similar to case i of Part A we have

$w_{I'_{ou}} = \underline{\text{if}}\ (z_{I'_{su}} = \underline{\text{true}})\ \underline{\text{then}}\ y_{I'_{su}}\ \underline{\text{else}}\ \perp$

$\Rightarrow \quad w_{I'_{ou}} = (y_{I'}\ \text{asa}\ z_{I'})_t \qquad$ by definition of asa

$\Rightarrow \quad (\text{first}\ w_{I'})_t = (y_{I'}\ \text{asa}\ z_{I'})_t$

$\Rightarrow \quad x_{I'_t} = (y_{I'}\ \text{asa}\ z_{I'})_t \qquad$ since I' satisfies (2)

and thus I' satisfies (1).

## APPENDIX

Programs as defined in section 2.1 do not contain the asa operator. At first sight this may appear to restrict the power of the language since asa is somewhat like a minimisation operator. But we prove a theorem here that enables all occurrences of asa to be removed from any program, leaving an equivalent program.

Theorem For any Lucid program, $\pi$, containing an occurrence of asa there is a Lucid program $\pi'$ such that

1) Every variable that appears in $\pi$ also appears in $\pi'$.

2) The minimal solutions of $\pi$ and $\pi'$ agree over the variables in $\pi$.

3) There is one less occurrence of asa in $\pi'$ than in $\pi$.

Proof

Notation: Let $\mathrm{next}^k$ denote k applications of next for $k \geq 0$. We will write first, next etc. where $\mathrm{first}_C$, $\mathrm{next}_C$ would be strictly correct.

Assume, without loss of generality, that asa occurs in $\pi$ in the form $x = y$ asa $z$ where x,y and z are variables. If A is the set of the remainder of the equations in $\pi$ then $\pi$ has the form

A

$x = y$ asa $z$ (1)

Construct the following program, $\pi'$

A

$x =$ first $w$ (2)

$w =$ if $z$ then $y$ else next $w$ (3)

where w is a new variable not occurring in $\pi$.

To prove that $\pi$ and $\pi'$ have the same minimal solution over all variables but w it is sufficient to show that

A) The minimal solution of $\pi$, denoted I, satisfies $\pi'$.

B) The minimal solution of $\pi'$, denoted I', satisfies $\pi$.

The proof of each part follows separately.

<u>Part A</u> We verify that I satisfies $\pi'$ by extending I to assign a meaning to w and then showing that all the equations of $\pi'$ are satisfied. Let $\bar{w}$ be assigned to w where $\bar{w}$ is the minimal solution of

$$w = \underline{\text{if}}\ z_I\ \underline{\text{then}}\ y_I\ \underline{\text{else}}\ \mathbf{next}\ w \qquad (4)$$

Since $\bar{w}$ satisfies (4), we have immediately that $\bar{w}$, $y_I$, $z_I$ satisfy (3). The set A is common to $\pi$ and $\pi'$. It therefore only remains to show that $x_I$ and $\bar{w}$ satisfy (2). That is we must prove

$$\text{for all } t \in N^N,\ x_{I_t} = (\mathbf{first}\ \bar{w})_t$$

Consider then any $t \in N^N$. Let $t = t_0u$ where $t_0 \in N$ and $u \in N^N$. We divide the argument into two cases

i) $(\exists s \geq 0)\ z_{I_{su}} \neq \underline{\text{false}}$ and $0 \leq r < s \Rightarrow z_{I_{ru}} = \underline{\text{false}}$

ii) $(\forall i \geq 0)\ z_{I_{iu}} = \underline{\text{false}}$

<u>Case i</u> Apply $\mathbf{next}^i$ to each side of (4) for $i \geq 0$

$$\text{next}^i\ \bar{w} = \underline{\text{if}}\ \mathbf{next}^i\ z_I\ \underline{\text{then}}\ \mathbf{next}^i\ y_I\ \underline{\text{else}}\ \mathbf{next}^{i+1}\ \bar{w} \qquad i \geq 0$$

$$\Rightarrow \bar{w}_{iu} = \underline{\text{if}}\ z_{I_{iu}}\ \underline{\text{then}}\ y_{I_{iu}}\ \underline{\text{else}}\ \bar{w}_{i+1\ u}$$

$$\Rightarrow \bar{w}_{iu} = \bar{w}_{i+1\ u} \text{ for } i < s \text{ and } \bar{w}_{su} = \underline{\text{if}}\ (z_{I_{su}} = \underline{\text{true}})\ \underline{\text{then}}\ y_{I_{su}}\ \underline{\text{else}}\ \perp$$

$$\Rightarrow \bar{w}_{ou} = \underline{\text{if}}\ (z_{I_{su}} = \underline{\text{true}})\ \underline{\text{then}}\ y_{I_{su}}\ \underline{\text{else}}\ \perp$$

But, by the definition of $\mathbf{asa}$ and from (1)

$$x_{I_t} = \underline{\text{if}}\ (z_{I_{su}} = \underline{\text{true}})\ \underline{\text{then}}\ y_{I_{su}}\ \underline{\text{else}}\ \perp$$

$$\Rightarrow x_{I_t} = \bar{w}_{ou} = (\mathbf{first}\ \bar{w})_t \text{ as required.}$$