INTERMITTENT ASSERTION PROOFS IN LUCID

by

E. A. Ashcroft

Research Report CS-76-47

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

November 1976

INTERMITTENT ASSERTION PROOFS IN LUCID

by

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

Abstract

The intermittent assertion technique of Burstall can be formulated and made rigorous in the formal-system/programming-language Lucid, in a very straightforward way. This reinforces the contention that Lucid is a framework within which many sorts of proofs of program properties may be expressed.

This paper includes three proofs, all of which are the Lucid versions of intermittent assertion proofs found in the literature.

Introduction

Three years ago, Burstall presented an invited paper at IFIP74 entitled "Program Proving as Hand Simulation with a little Induction" [3]. Seldom are invited papers so original. The technique he presented, which has been called the "intermittent assertion" method by Manna and Waldinger [5], took some time to register with the program-proving community, but is now being increasingly recognised as a method of paramount importance. Not only does it subsume previous methods (Manna and Waldinger [5]), but also it is very natural to use, and could prove to be the long-sought philosopher's stone which transmutes base programmers into golden program-provers.

At the same time, Lucid was being developed [1,2]. The intention with Lucid was not so much to present new proof techniques as to rebuild the foundations of programming and program proving to give a single coherent structure. There were two guiding principles used in its construction; the programming language was to be reasonably natural and understandable, using iteration as its basic "control structure" and assignment as its basic "operation", and yet the language was to be completely denotational, with mathematical properties such as substitutivity and "referential transparency". Moreover, assignment statements were to be equations. The solution to these seemingly contradictory requirements will not be detailed here (see Ashcroft & Wadge [1,2]).

If Lucid is a general structure within which both program-writing and program-proving can be carried out, it is natural to ask (even demand) whether the intermittent assertion technique fits into Lucid. This paper shows that in fact it does, and moreover suggests that the technique can only be put on a sound semantical footing by embedding it within a modal logic with some of the properties of Lucid.

The intermittent assertion technique

I shall use the notation of Manna and Waldinger [5]. For a program P containing a label L, the statement "sometime A at L" asserts that, at some stage in the computation of P, control will be at label L with the assertion A being true at that time. Which computation of P is being considered is taken to be understood. (One can imagine the full statement is "sometime P at L in c" where c denotes the computation in question. During proofs, c never changes, and we can consider that normally the phrase "in c" is suppressed. This is quite justifiable since some sort of a modal logic is being used anyway, as pointed out by Burstall in his paper.)

Using such statements it is possible to carry out very natural-seeming proofs about programs, proving not just partial but also total correctness.

Many such proofs are based on lemmas of the form

"sometime A at L implies sometime B at M".

The way such a statement is usually understood (and this is what is normally proved) is to consider B to be true at M _after_ A is true at L. However, the above statement would also be true if B were true at M _before_ A is true at L. For programs for which the final result is the important thing, this doesn't really make any difference. But one of the attractive areas of application is in the study of continuously operating programs where "sometime" statements look promising as ways of describing desired _behaviour_.

And in this application, we really do want "sometime" to mean "sometime later", as in "sometime A at L implies sometime later B at M". If this is to be a formal statement in a proof, it is by no means clear how the word "later" is to be interpreted; is "sometime later" a single thing, or is "later" qualifying the whole "sometime" clause? And if we ask "later than what" we can only reply "after A is true at L", so that time information has crossed the implication. Another way of looking at this is to note that "sometime B at L" is either true or false, but the truth or falsity of "sometime later B at L" depends on time, the stage reached in the computation. It appears hard to formalise such statements, but formalisation is needed if we are to propose rules for reasoning about them.

Intermittent assertions in Lucid

Lucid can be looked upon as a modal logic, which reasons about time while suppressing all explicit mention of time. This is achieved by making variables and expressions in Lucid, say X, denote generalised infinite sequences, with the t-th component, X_t, representing the t-th value that X would take on during a "computation". (In general the t's are not just natural numbers, but infinite sequences of natural numbers, corresponding to numbers of iterations of various loops.) In Lucid, a statement $A \rightarrow B$ means that, for all times, if A is true at some time then B is true at the same time. Time information crosses the implication. It is not surprising that Lucid can take "sometime later" in its stride, in fact by simply considering it as a new Lucid function, which we call "later". The "sometime" of Burstall and Manna & Waldinger is rendered by a function "sometime" which is actually the same as the already existing Lucid function "eventually". The function "later" will actually be defined in terms of a further function "when next" which has already been used (with a different name) by Cargill [4] to define the Lucid function "as soon as".

Definitions

We use the notation in [1].

(Cargill's function:)

$$(\alpha \text{ when next } \beta)_{t_0 t_1 t_2 \ldots} = \alpha_{r t_1 t_2 \ldots} \text{ if } \beta_{r t_1 t_2 \ldots} \text{ is true}$$

and $\beta_{s t_1 t_2 \ldots}$ is false for all s, $t_0 \le s < r$,

undefined if no such r exists.

This function can also be defined as follows:

P when next Q = if Q then P else next (P when next Q).

(This looks like a non-terminating recursive definition, but in fact the when next on the right is looked at at a different time than the one on the left (because of the function next), and we are simply defining when next in terms of its own "future".)

We now have

later P = P when next P

sometime P = first later P

P as soon as Q = first (P when next Q).

Using these functions we can now write terms such as

$$\text{later } A \longrightarrow \text{later } B$$

which can be interpreted as saying that, at all times, if sometime in the future A is true, then sometime in the future B is true. Since this is true at all times, even the times when A is true, it follows that B is true later than A. In fact, it is a simple deduction from the above that

$$A \longrightarrow \text{later } B ,$$

namely whenever A is true, B is true sometime afterwards (or even at the same time).

Unfortunately, there are features of Lucid which slightly complicate proofs using sometime and later. The variables in Lucid denote generalised infinite sequences, even if they are used in a "terminating" loop. Thus, even if sometime Y > X say, there is no guarantee that Y becomes bigger than X before the loop using X and Y has terminated. If the test on the as soon as for the loop is P, what we want to know is whether sometime (Y > X ∧ hitherto ¬P). We introduce a more general form of sometime to express this; we say sometime Y > X before P. (We really should use sometime Y > X not later than P which more accurately conveys the meaning.) Note that sometime...before is a single function. We also have such a function corresponding to later.

Definitions

later P before Q = (Q → P) when next P ∨ Q

sometime P before Q = first later P before Q

= sometime (P ∧ hitherto ¬Q)

It is easily seen that later...before can also be defined as follows, and is in fact continuous, even though the previous definition used "→".

$(\text{later } \alpha \text{ before } \beta)_{t_0 t_1 t_2 \ldots}$ = true if

$\alpha_r t_1 t_2 \ldots$ is true

and $\alpha_s t_1 t_2 \ldots$ is false

and $\beta_s t_1 t_2 \ldots$ is false

for all s, $t_0 \le s < r$,

false if $\beta_r t_1 t_2 \ldots$ is true

and $\alpha_r t_1 t_2 \ldots$ is false

and $\alpha_s t_1 t_2 \ldots$ is false

and $\beta_s t_1 t_2 \ldots$ is false

for all s, $t_0 \le s < r$,

undefined if no such r exists.

Note that

$$\text{later } P = \text{later } P \text{ before } P$$

$$\text{sometime } P = \text{sometime } P \text{ before } P$$

Axioms and rules of inference

To formalise reasoning using these functions the following axioms and rules of inference are useful. (It is assumed that P, Q etc. are formulas, and at any time can only take the values true, false or undefined. The assertion def P means $P{=}T \vee P{=}F$.)

(0) $\neg(P{=}T) \models \neg(\text{later } P \text{ before } Q = T)$

(0') $\neg(P{=}T) \models \neg(\text{sometime } P \text{ before } Q = T)$

(1) $\models P \to \text{later } P \text{ before } Q$

(1') $\models \text{first } P \to \text{sometime } P \text{ before } Q$

(2) $P \to (\neg Q \wedge \text{next } R),\ \text{def } P \to \text{def } R$

$\models \text{later } P \text{ before } Q \to \text{later } R \text{ before } Q$

(2') $P \to (\neg Q \wedge \text{next } R),\ \text{def } P \to \text{def } R$

$\models \text{sometime } P \text{ before } Q \to \text{sometime } R \text{ before } Q$

(3) $P_1 \to P_2,\ Q_2 \to Q_1,\ \text{def } P_1 \wedge \text{def } Q_1 \to \text{def } P_2 \wedge \text{def } Q_2$

$\models \text{later } P_1 \text{ before } Q_1 \to \text{later } P_2 \text{ before } Q_2$

(3') $P_1 \to P_2,\ Q_2 \to Q_1,\ \text{def } P_1 \wedge \text{def } Q_1 \to \text{def } P_2 \wedge \text{def } Q_2$

$\models \text{sometime } P_1 \text{ before } Q_1 \to \text{sometime } P_2 \text{ before } Q_2$

(4) $P \to Q \models \text{later } P \text{ before } R \to \text{later } P \wedge Q \text{ before } R$

(4') $P \to Q \models \text{sometime } P \text{ before } R \to \text{sometime } P \wedge Q \text{ before } R$

(5) $\models$ sometime $P \wedge Q$ before $Q \rightarrow P$ as soon as Q

(6) sometime P before Q, $P \rightarrow Q \models P$ as soon as Q

(6') later P before Q, $P \rightarrow Q \models P$ as soon as Q

(7) P as soon as $Q \models$ eventually Q

(8) P as soon as Q, $P \rightarrow R \models R$ as soon as Q

Each rule (i') can be derived from the corresponding rule (i). Rule 6 is derived from rules (4') and (5).

Examples of proofs

The proofs given in sections 2 and 5 of Burstall's paper will be carried out within Lucid. Also, a proof will be given for a continuously operating program, taken from Manna & Waldinger's paper.

I) The program which computes 2^n is as follows:

```
first P = 1
first N = n
next P = 2 × P
next N = N-1
OUTPUT = P as soon as N ≤ 0
```

In the above program, and in the rest of the paper, variables written in small letters will range over constants, in this case integers. (In the next proof they will also be binary trees and lists.) Quantification over such variables does not allow undefined as a possible value. (This means that universal quantification can not be instantiated using an arbitrary term, and arbitrary terms can not be converted to existentially quantified variables. In both cases the term must first be proved to be defined and constant.) The reasoning about such variables is then very much like conventional mathematical reasoning, and we can use mathematical induction, structural induction etc.

Theorem 1. For this program

$$n \geq 0 \rightarrow \text{OUTPUT} = 2^n$$

Proof. We first prove the following lemma

Lemma 1

$\forall i \;\; 0 \leq i \leq n$, sometime $P = 2^i \wedge N = n-i$ before $N \leq 0$

Proof by induction on i

i=0 We immediately have from the program that

first $(P=2^0 \wedge N = n-0)$.

Thus, by rule (1')

sometime $P=2^0 \wedge N = n-0$ before $N \leq 0$.

This completes the base step.

Now assume that for some i, $0 \leq i < n$

$P=2^i \wedge N = n-i$.

We thus have that the termination condition is false, and we find the effect of "going round the loop": $N > 0 \wedge$ next $P = 2^{i+1} \wedge$ next $N = n-(i+1)$

i.e. $N > 0 \wedge$ next$(P=2^{i+1} \wedge N = n-(i+1))$.

Discharging our assumption (because no Lucid rules used, no substitution within Lucid functions), we get

$P=2^i \wedge N = n-i \rightarrow N > 0 \wedge$ next $(P=2^{i+1} \wedge N = n-(i+1))$.

Now it is obvious that def $(P=2^{i+1} \wedge N = n-(i+1))$ since "=" always gives true or false at any time. Thus, applying rule (2') we get

sometime $P=2^i \wedge N=n-i$ before $N \leq 0 \longrightarrow$
sometime $P=2^{i+1} \wedge N=n-(i+1)$ before $N \leq 0$

This is the induction step, completing the proof of the lemma.

□

To prove the theorem we now assume $n \geq 0$, and take i=n in the Lemma. (This instantiation is all right because n is obviously constant and defined.) We get

$$\underset{\sim}{\text{sometime}}\ P=2^n \wedge N=0\ \underset{\sim}{\text{before}}\ N \le 0.$$

Now $P=2^n \wedge N=0 \rightarrow N \le 0$, so applying rule (6) we get

$$(P=2^n \wedge N=0)\ \underset{\sim}{\text{as soon as}}\ N \le 0$$

and hence $(P=2^n)$ as soon as $N \le 0$ (by (8)). Since eventually $N \le 0$ (by (7)) we can push the as soon as inside (this rule being given in [1]), giving

$$P\ \underset{\sim}{\text{as soon as}}\ N \le 0 = 2^n$$

Since OUTPUT = P as soon as $N \le 0$, and our original assumption, $n \ge 0$, is constant, in two steps we get

$$n \ge 0 \longrightarrow \text{OUTPUT} = 2^n.$$

□

II) This program, which counts the tips of a binary tree, is as follows:

```
first (COUNT,T,STACK) = (0,t,Λ)
next (COUNT,T,STACK) =
    if T eq nil then (COUNT+1,right(hd(STACK)),tℓ(STACK))
              else (COUNT,left(T), T∘STACK)
OUTPUT = COUNT+1 as soon as T eq nil ∧ STACK eq Λ.
```

Variable STACK holds lists, with Λ denoting the empty list, hd, $t\ell$ and $\circ$ having the usual property $s = hd(s) \circ t\ell(s)$. Variable T holds binary trees, with "nil" denoting the tree consisting of single leaf, and left(T) and right (T) being the left and right subtrees of T. We will use the function tips defined by $tips(\tau) = \underline{if}\ \tau=nil\ \underline{then}\ 1\ \underline{else}\ tips(left(\tau)) + tips(right(\tau))$. We take this as our definition of what the number of tips of a tree τ actually is.

<u>Theorem 3</u> For this program

$$\text{OUTPUT} = \text{tips}(t)$$

<u>Proof</u> We first need the following Lemma:

<u>Lemma 2</u>

$\underset{\sim}{\text{sometime}}$ COUNT=c ∧ T=t' ∧ STACK = s $\underset{\sim}{\text{before}}$ (T eq nil ∧ STACK eq Λ)

⟶ $\underset{\sim}{\text{sometime}}$ COUNT=c+tips(t')-1 ∧ T=nil ∧ STACK=s

$\underset{\sim}{\text{before}}$ (T eq nil ∧ STACK eq Λ)

<u>Proof</u> By structural induction on t'.

<u>t'=nil</u>. In this case c=c+tips(t')-1 and the result is immediate.

<u>Otherwise</u> (t'≠nil, in fact t' ne nil): Assume COUNT=c ∧ T=t' ∧ STACK=s.

It is easily seen that the termination condition is false, and we go around the loop: (T ne nil ∨ STACK ne Λ) ∧ $\underset{\sim}{\text{next}}$(COUNT=c ∧ T=left(t')

∧ STACK=t'∘ s).

This gives us the implication need as a premise of rule (2'), and since <u>def</u>(COUNT=c ∧ T=left(t') ∧ STACK=t'∘ s) we get

(*) $\underset{\sim}{\text{sometime}}$ COUNT=c ∧ T=t' ∧ STACK=s $\underset{\sim}{\text{before}}$ (T eq nil ∧ STACK eq Λ)

⟶ $\underset{\sim}{\text{sometime}}$ COUNT=c ∧ T=left(t') ∧ STACK=t'∘ s $\underset{\sim}{\text{before}}$ (T eq nil ∧ STACK eq Λ).

Now, by the induction hypothesis, since left(t') is a subtree of t',

(**) $\underset{\sim}{\text{sometime}}$ COUNT=c ∧ T=left(t') ∧ STACK=t'∘ s $\underset{\sim}{\text{before}}$ (T eq nil ∧ STACK eq Λ)

⟶ $\underset{\sim}{\text{sometime}}$(COUNT=c+tips(left(t'))-1 ∧ T=nil ∧ STACK=t'∘ s

$\underset{\sim}{\text{before}}$ (T eq nil ∧ STACK eq Λ).

Now we assume that in fact

COUNT=c+tips(left(t'))-1 ∧ T=nil ∧ STACK=t'∘ s.

Again, it quickly follows, by a "hand simulation", that

$(T \text{ ne nil} \vee \text{STACK ne } \Lambda) \wedge \underset{\sim}{\text{next}}(\text{COUNT}=c+\text{tips}(\text{left}(t')) \wedge T=\text{right}(t') \wedge \text{STACK}=s)$

and we can discharge our assumption, giving an implication. Once more $\underline{\text{def}}(\text{COUNT}=c+\text{tips}(\text{left}(t')) \wedge T=\text{right}(t') \wedge \text{STACK}=s)$ and so by (2')

$$(***) \quad \underset{\sim}{\text{sometime}}\ \text{COUNT}=c+\text{tips}(\text{left}(t'))-1 \wedge T=\text{nil} \wedge \text{STACK}=t'\circ s \ \underset{\sim}{\text{before}}\ (T \text{ eq nil} \wedge \text{STACK eq } \Lambda)$$
$$\longrightarrow \underset{\sim}{\text{sometime}}\ \text{COUNT}=c+\text{tips}(\text{left}(t')) \wedge T=\text{right}(t') \wedge \text{STACK}=s \ \underset{\sim}{\text{before}}\ (T \text{ eq nil} \wedge \text{STACK eq } \Lambda) .$$

Since right(t') is a subtree of t', we apply the induction hypothesis to get

$$(****) \quad \underset{\sim}{\text{sometime}}\ \text{COUNT}=c+\text{tips}(\text{left}(t')) \wedge T=\text{right}(t') \wedge \text{STACK}=s \ \underset{\sim}{\text{before}}\ (T \text{ eq nil} \wedge \text{STACK eq } \Lambda)$$
$$\longrightarrow \underset{\sim}{\text{sometime}}\ \text{COUNT}=c+\text{tips}(\text{left}(t')+\text{tips}(\text{right}(t')))-1 \wedge T=\text{nil} \wedge \text{STACK}=s \ \underset{\sim}{\text{before}}\ (T \text{ eq nil} \wedge \text{STACK eq } \Lambda)$$

If we now chain together the implications (*), (**), (***) and (****), and use the definition of tips, we get

$$\underset{\sim}{\text{sometime}}\ \text{COUNT}=c \wedge T=t' \wedge \text{STACK}=s \ \underset{\sim}{\text{before}}\ (T \text{ eq nil} \wedge \text{STACK eq } \Lambda)$$
$$\longrightarrow \underset{\sim}{\text{sometime}}\ \text{COUNT}=c+\text{tips}(t')-1 \wedge T=\text{nil} \wedge \text{STACK}=s \ \underset{\sim}{\text{before}}\ (T \text{ eq nil} \wedge \text{STACK eq } \Lambda)$$

which is what we wanted.

$\square$

We can now use the Lemma to prove the theorem as follows. Since $\underset{\sim}{\text{first}}(\text{COUNT},T,\text{STACK}) = (0,t,\Lambda)$ we get, using (1'),

$$\underset{\sim}{\text{sometime}}\ \text{COUNT}=0 \wedge T=t \wedge \text{STACK} = \Lambda \ \underset{\sim}{\text{before}}\ (T \text{ eq nil} \wedge \text{STACK eq } \Lambda)$$

Applying the lemma we get

sometime COUNT=tips(t)-1 $\wedge$ T=nil $\wedge$ STACK = Λ

before (T eq nil $\wedge$ STACK eq Λ).

Rule (6) followed by (8) gives

(COUNT+1=tips(t)) *as soon as* (T eq nil $\wedge$ STACK eq Λ).

As in theorem 1, we now say

eventually (T eq nil $\wedge$ STACK eq Λ) and so we can push the *as soon as* inside:

(COUNT+1) *as soon as* (T eq nil $\wedge$ STACK eq Λ) = tips(t)

i.e. OUTPUT = tips(t).

$\square$

III) The following program is a Lucid version of the simple operating system program in the Manna & Waldinger paper.

```
begin
    first JOBQUEUE  = INPUT
    first PRINTOUTS = nil
    JOB = hd(JOBQUEUE)
    next PRINTOUTS = PRINTOUTS∘process (JOB)
    next JOBQUEUE = tℓ(JOBQUEUE)
    OUTPUT = PRINTOUTS as soon as JOBQUEUE eq nil
end
```

The begin and end indicate that the enclosed program is "executed" for each value of "INPUT", giving a corresponding value of "OUTPUT". Each "INPUT" value will be a list of jobs, and the corresponding "OUTPUT" value will be a list of the results of these jobs.

We can express the fact that jobs never get lost by the following theorem.

<u>Theorem 3</u> For this program

$$\mathbf{later}\ \text{JOBQUEUE} = \ell \wedge j \in \ell\ \mathbf{before}\ \text{JOBQUEUE eq nil}$$
$$\longrightarrow \mathbf{later}\ \text{JOB} = j\ \mathbf{before}\ \text{JOBQUEUE eq nil}$$

<u>Proof</u>

By structural induction on ℓ.

<u>ℓ = nil</u>. Since $(\text{JOBQUEUE} = \ell \wedge j \in \ell) \neq T$, by (0) we have

$$\mathbf{later}\ (\text{JOBQUEUE} = \ell \wedge j \in \ell)\ \mathbf{before}\ \text{JOBQUEUE eq nil} \neq T$$

and the result is immediate.

<u>otherwise</u>, assume the theorem for $t\ell(\ell)$.

Now $j = hd(\ell) \vee j \neq hd(\ell)$.

Assume $j = hd(\ell)$

Then $(\text{JOBQUEUE} = \ell \wedge j \in \ell) \rightarrow \text{JOB} = j$

and so, by (3),

$$\mathbf{later}\ \text{JOBQUEUE} = \ell \wedge j \in \ell\ \mathbf{before}\ \text{JOBQUEUE eq nil}$$
$$\longrightarrow \mathbf{later}\ \text{JOB} = j\ \mathbf{before}\ \text{JOBQUEUE eq nil}.$$

Since our assumption is constant we can discharge it:

$$j = hd(\ell) \rightarrow (\mathbf{later}\ \text{JOBQUEUE} = \ell \wedge j \in \ell\ \mathbf{before}\ \text{JOBQUEUE eq nil}$$
$$\longrightarrow \mathbf{later}\ \text{JOB} = j\ \mathbf{before}\ \text{JOBQUEUE eq nil})$$

Now assume $j \neq hd(\ell)$, and further assume $\text{JOBQUEUE} = \ell \wedge j \in \ell$. Clearly

$$\text{JOBQUEUE ne nil} \quad \wedge\ \mathbf{next}\ (\text{JOBQUEUE} = t\ell(\ell) \wedge j \in t\ell(\ell)).$$

Since <u>def</u> $(\text{JOBQUEUE} = t\ell(\ell) \wedge j \in t\ell(\ell))$ we can discharge the latter assumption, and then apply (2), to give

later JOBQUEUE $= \ell \wedge j \in \ell$ **before** JOBQUEUE eq nil

$\longrightarrow$ **later** JOBQUEUE $= t\ell(\ell) \wedge j \in t\ell(\ell)$ **before** JOBQUEUE eq nil.

Now, by our induction hypothesis, since $t\ell(\ell)$ is a sublist of ℓ,

later JOBQUEUE $= t\ell(\ell) \wedge j \in t\ell(\ell)$ **before** JOBQUEUE eq nil

$\longrightarrow$ **later** JOB $= j$ **before** **next** JOBQUEUE eq nil.

Combining these results,

later JOBQUEUE $= \ell \wedge j \in \ell$ **before** JOBQUEUE eq nil

$\longrightarrow$ **later** JOB $= j$ **before** JOBQUEUE eq nil.

We can discharge our former assumption, since it is constant, to give

$j \neq hd(\ell) \rightarrow$ (**later** JOBQUEUE $= \ell \wedge j \in \ell$ **before** JOBQUEUE eq nil

$\longrightarrow$ **later** JOB $= j$ **before** JOBQUEUE eq nil).

Thus, since $j = h(\ell) \vee j \neq hd(\ell)$, we have

later JOBQUEUE $= \ell \wedge j \in \ell$ **before** JOBQUEUE eq nil

$\longrightarrow$ **later** JOB $= j$ **before** JOBQUEUE eq nil.

This completes the induction step.

$\square$

These three proofs have closely followed the steps in the original versions. Note however that the third proof actually proved a stronger statement than was proved in the original version in the Manna & Waldinger paper. In order even to state the desired property requires a time-dependent logic, and is beyond the scope of informal methods.

Conclusion

It has been shown that proofs using Burstall's intermittent assertion method can easily be accomplished using Lucid. Moreover, more powerful and useful statements using intermittent assertions can be stated and proved in Lucid than are possible using the informal description of the method.

It is difficult to think of a way of formalising the method, to deal with these more powerful statements, that does not result in statements whose value varies with time and in logical connectives such as "implies" conveying some information about time. This means that the whole body of mathematical reasoning used will be affected with this time dependent component. At this stage it becomes apparent that a self-contained formal theory must be developed which reasons about time, without mentioning time explicitly. I suggest that that theory is the part of Lucid dealing with logical statements. Whether that part of Lucid can be used without the rest of Lucid remains to be seen.

Acknowledgement

Thanks are due to Tom Cargill for stimulating discussions in which some of the ideas presented here were worked out; in particular the definition of **when next**.

The idea of introducing special variables into Lucid proofs to denote constants, thus allowing proofs by mathematical induction, structural induction, etc. is due to Bill Wadge.

References

[1] Ashcroft, E.A. & Wadge, W.W. "Lucid, a Formal System for Writing and Proving Programs" SIAM J. on Computing Vol.5, No.3. pp.336-354.

[2] Ashcroft, E.A. & Wadge, W.W. "Lucid, a Non-Procedural Language with Iteration", to appear in CACM.

[3] Burstall, R.M. "Program Proving as Hand Simulation with a little Induction" IFIP74 Congress, Stockholm.pp 308-312.

[4] Cargill, T.A. "Deterministic Operational Semantics for Lucid". Research Report CS-76-19, Computer Science Department, University of Waterloo.

[5] Manna, Z. & Waldinger, R. "Is 'sometime' sometimes better than 'always'? Intermittent Assertions in Proving Correctness of Programs", STAN-CS-76-558, Computer Science Dept., Stanford University.