Intermittent Assertion Proofs in Lucid

E. A. Ashcroft  W. W. Wadge

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 rein-
forces 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.

1 Introduction

Three years ago, Burstall presented an invited paper at IFIP74 entitled “Program Proving as
Hand Simulation with a Little Induction” [4]. The technique he presented has been called the
“intermittent assertion” method by Manna and Waldinger [6].

At the same time, Lucid was being developed [2, 3, 1]. 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 construc-
tion; the programming language was to be reasonably natural and understandable, using iteration
as its basic “operation”, and yet the language was to be completely denotational, with mathe-
matical 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 and Wadge [2, 3, 1]).

If Lucid is a general structure within which both program writing and program proving can
be carried out, it is natural to ask 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.

2 The Intermittent Assertion Technique

We shall use the notation of Manna and Waldinger [6]. For a program $P$ containing a label $L$, the
statement “sometime $A$ at $L$” asserts that, now or at some later state 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. In fact we are using a modal logic, each “setting”
of which is a computation, the “worlds” being successive stages of the computation.

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 $M$”

If this is not dependent on other assumptions, it is then true at all stages in the computation,
including the stage when $A$ is true at $L$. We see that the statement implies that $B$ is true at $M$
after $A$ is true at $L$.

∗Published in Information Processing 77, pages 723–726, North-Holland, 1977.
3 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 generalized 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 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. This is exactly the meaning that would be attached to such a statement in the modal logic mentioned earlier. In fact the same modal logic is involved. It is not surprising that the “sometime” statements of the intermittent assertion technique can be written in Lucid, by merely using a new Lucid function called $\mathtt{sometime\ldots before}$.

It might seem sufficient to simply have a function $\mathtt{sometime}$ such that

\[
\text{(sometime } \alpha \text{)_{t_0 t_1 t_2 \ldots}} = \begin{cases} 
\text{true,} & \text{if for some } s \geq t_0, \alpha_{st_1 t_2 \ldots} \text{ is true} \\
\text{false,} & \text{if for all } s \geq t_0, \alpha_{st_1 t_2 \ldots} \text{ is false} \\
\text{undefined,} & \text{otherwise}
\end{cases}
\]

(We are using the notation in [2].)

Thus at any time, sometime $P$ is true of $P$ is true now or later, and is false of $P$ is false from now on. (The function sometime is not continuous and could not be used in any program.)

Using this function we could now write terms such as

\[
\text{sometime } A \rightarrow \text{ sometime } B
\]

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

\[
A \rightarrow \text{ sometime } B
\]

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. The variables in Lucid denote generalized 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 $P$, we can express what we want by using a more general form of sometime: we say sometime $Y > X$ before $P$. (We really should say sometime $Y > X$ not later than $P$ which more accurately conveys the meaning.) Note that sometime $\ldots$ before is a single function.

\[
\text{(sometime } \alpha \text{ before } \beta)_{t_0 t_1 t_2 \ldots} = \begin{cases} 
\text{true,} & \text{if for some } r \geq t_0, \alpha_{rt_1 t_2 \ldots} \text{ is true and} \\
\beta_{st_1 t_2 \ldots} \text{ is false for all } s, t_0 \leq s < r \\
\text{false,} & \text{if for some } r \geq t_0, \beta_{rt_1 t_2 \ldots} \text{ is true and} \\
\alpha_{st_1 t_2 \ldots} \text{ is false for all } s, t_0 \leq s < r \\
\text{undefined,} & \text{if no such } r \text{ exists}
\end{cases}
\]

Note that sometime $\ldots$ before is continuous, and

\[
\text{sometime } P \leftrightarrow \text{ sometime } P \text{ before } P
\]

4 Pronouns and Pro-functions

It is very tedious in proofs to have to carry around the phrase “before $P$” all the time. Intermittent assertion proofs are ideally suited to the use of pronouns and pro-functions (see Wadge [7]). A
pronoun is a special or reserved local variable which cannot be “renamed”. A pro-function is a function whose definition is implicit and which differs from basic data operations and Lucid functions in that the definition refers to global variables, usually pronouns.

**Program 1.** For example, consider the following simple program for computing $2^N$:

```
compute Power using N
  first P = 1
  first M = N
  next P = 2 * P
  next M = M - 1
  Halt = M \leq 0
  Result = last(P)
end
```

The variables `Halt` and `Result` are pronouns. The function `last` is a pro-function. Its implicit definition is:

```
function last(X) using Halt
  Result = X as soon as Halt
end
```

In this way, the function *as soon as* need never appear explicitly, and proofs can become very natural.

For our purposes, we shall introduce a pro-function “`sometime`” (not to be confused with `sometime`, discussed earlier) with the implicit definition

```
function sometime(P) using Halt
  Result = sometime P before Halt
end
```

The function `sometime`···*before* then need never be used in proofs.

The axioms and rules of inference are quite direct.

## 5 Axioms and Rules of Inference

For readability, we shall write `firstly P` and `lastly(P)`, when `P` is a formula, rather than `first P` and `last(P)`.

\[
\begin{align*}
\models P \rightarrow \neg\text{Halt} & \wedge \text{next } Q \quad \models \text{sometime}(P) \rightarrow \text{sometime}(Q) \\
\models P \rightarrow Q & \quad \models \text{sometime}(P) \rightarrow \text{sometime}(Q) \\
\text{firstly sometime}(P \wedge \text{Halt}) & \quad \models \text{lastly}(P) \\
\text{lastly}(P) & \quad \models \text{last}(f(e_1, \ldots, e_n)) = f(\text{last}(e_1), \ldots, \text{last}(e_n)) \quad \text{for all pointwise operations } f \\
\text{lastly}(P), A = \text{first } A & \quad \models \text{last}(A) = A
\end{align*}
\]

## 6 Examples of Proofs

In the proofs in this paper variables written all in small letters will range over defined quiescent values (for example, integers, binary trees and lists). Quantification over such variables does not allow *undefined* as a possible value. (This means that universal quantification cannot be instantiated using an arbitrary term, and arbitrary terms cannot be converted to existentially quantified variables. In both cases the term must first be proved to be defined and quiescent.)
The reasoning about such variables is then very much like conventional mathematical reasoning, and we can use mathematical induction, structural induction, etc. (see [7]).

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 and Waldinger’s paper.

Our first example will be Program 1 of Section 4.

**Theorem 1.** For this program

\[ N \geq 0 \rightarrow Power = 2^N \]

**Proof.** Inside the compute, assume \( N \geq 0 \). Note that \( N = \text{first} \ N \). Thus \( N \) is quiescent and defined. Assume sometime\( (P = 1 \land M = N) \). We now need:

**Lemma 1.** \( \forall i \ 0 \leq i \leq N \) sometime\( (P = 2^i \land M = N - i) \)

**Proof.** By induction on \( i \).

\[ i = 0. \text{ Immediate, since we have sometime}(P = 1 \land M = N). \]

\[ i = I. \text{ Assume } 0 \leq I < N, \text{ and assume } P = 2^I \land M = N - I. \text{ We see that } M > 0 \text{ and so } \neg\text{Halt}, \text{ and also next}(P = 2^{I+1} \land M = N - (I + 1)), \text{ since next } P = 2 * P \text{ and next } M = M - 1. \]

Thus, by Rule (2), we get

\[ \text{sometime}(P = 2^I \land M = N - I) \rightarrow \text{sometime}(P = 2^{I+1} \land M = N - (I + 1)) \]

This completes the induction step, and the proof of the lemma.

Now, instantiating in the lemma with \( N \) (which is valid since \( N \) is quiescent and defined), sometime\( (P = 2^N \land M = 0) \), and so sometime\( (P = 2^N \land \text{halt}) \) (by Rule (3)). Hence sometime\( (P = 1 \land M = N) \rightarrow \text{sometime}(P = 2^N \land \text{halt}). \)

Since we have firstly\( (P = 1 \land M = N) \), by Axiom (1) we get firstly sometime\( (P = 2^N \land \text{halt}) \). By Rule (4), lastly\( (P = 2^N) \) and then by Rules (5) and (6),

\[ \text{result} = \text{last}(P) = 2^N. \]

We can discharge our initial assumption since it is quiescent:

\[ N \geq 0 \rightarrow \text{result} = 2^N. \]

and this can be moved out of the compute, becoming

\[ N \geq 0 \rightarrow Power = 2^N. \]

This completes the proof of the main theorem.

**Program 2.** The second program, which counts the tips of a binary tree, is as follows:

```plaintext
compute Leaves using Tr
    first(Count, T, Stack) = (0, Tr, \Lambda)
    next(Count, T, Stack) =
        if T eq nil
            then (Count + 1, right(hd(Stack)), tl(Stack))
        else (Count, left(T), T \circ Stack)
    Halt = T eq nil \land Stack eq \Lambda
    Result = last(Count) + 1
```
Variable \textit{Stack} holds lists, with \( \Lambda \) denoting the empty list, \( \text{hd}, \text{tl} \) and \( \circ \) having the usual property \( s = \text{hd}(s) \circ \text{tl}(s) \). Variable \( T \) holds binary trees, with “nil” denoting the tree consisting of a single leaf, and \( \text{left}(T) \) and \( \text{right}(T) \) being the left and right subtrees of \( T \). We will use the function \textit{tips} defined by

\[
tips(\tau) = \begin{cases} 
1 & \text{if } \tau = \text{nil} \\
\text{tips}((\text{left}(\tau)) + \text{tips}(\text{right}(\tau)) & \text{else}
\end{cases}
\]

We take this as our definition of what the number of tips of \( \tau \) is, provided \( \text{tree}(\tau) \), i.e., \( \tau \) is a tree.

**Theorem 2.** For this program,

\[ \text{tree}(T) \rightarrow \text{Leaves} = \text{tips}(T) \]

**Proof.** We first need the following lemma:

**Lemma 2.** If \( \text{tree}(T) \) then

\[
\text{sometime}(\text{Count} = c \land T = t \land \text{Stack} = s) \rightarrow \\
\text{sometime}(\text{Count} = c + \text{tips}(t) - 1 \land \text{Tree} = \text{nil} \land \text{Stack} = s)
\]

**Proof.** By structural induction on \( t \).

\( t = \text{nil} \). In this case \( \text{tips}(t) = 1 \) and the result is immediate.

\( t \neq \text{nil} \). (in fact \texttt{true} nil):

Assume \( \text{sometime}(\text{Count} = c \land T = t \land \text{Stack} = s) \). Now if \( \text{Count} = c \land T = t \land \text{Stack} = s \), it is easily seen that \( \neg \text{Halt} \) and

\[
\text{next}(\text{Count} = c \land T = \text{left}(t) \land \text{Stack} = t \circ s).
\]

Thus by Rule (2), \( \text{sometime}(\text{Count} = c \land T = \text{left}(t) \land \text{Stack} = t \circ s) \). Now, applying the induction hypothesis, since \( \text{left}(t) \) is a subtree of \( t \),

\[
\text{sometime}(\text{Count} = c + \text{tips}((\text{left}(t)) - 1 \land \text{Tree} = \text{nil} \land \text{Stack} = t \circ s).
\]

Again, if \( \text{Count} = c + \text{tips}((\text{left}(t)) - 1 \land \text{Tree} = \text{nil} \land \text{Stack} = t \circ s) \), it is easily seen that \( \neg \text{Halt} \) and

\[
\text{next}(\text{Count} = c + \text{tips}((\text{left}(t)) \land \text{Tree} = \text{right}(t) \land \text{Stack} = s).
\]

Applying Rule (2),

\[
\text{sometime}(\text{Count} = c + \text{tips}((\text{left}(t)) \land \text{Tree} = \text{right}(t) \land \text{Stack} = s).
\]

By the induction hypothesis,

\[
\text{sometime}(\text{Count} = c + \text{tips}((\text{left}(t)) + \text{tips}(\text{right}(t)) - 1 \land \text{Tree} = \text{nil} \land \text{Stack} = s),
\]

i.e., \( \text{sometime}(\text{Count} = c + \text{tips}(t) - 1 \land \text{Tree} = \text{nil} \land \text{Stack} = s) \). We can now discharge our original assumption (since Rule (2) was never applied to consequences of this assumption):

\[
\text{sometime}(\text{Count} = c \land \text{Tree} = t \land \text{Stack} = s) \rightarrow \\
\text{sometime}(\text{Count} = c + \text{tips}(t) - 1 \land \text{Tree} = \text{nil} \land \text{Stack} = s).
\]

This concludes the proof of the lemma. \( \square \)
We now use the lemma to prove the theorem as follows. Assume \( \text{tree}(Tr) \).

Since firstly(\( \text{Count} = 0 \land T = \text{Tr} \land \text{Stack} = \Lambda \)), by Axiom (1) and the lemma,

\[
\text{firstly sometime(Count} = \text{tips}(\text{Tr}) - 1 \land T = \text{nil} \land \text{Stack} = \Lambda),
\]

hence firstly sometime(\( \text{Count} = \text{tips}(\text{Tr}) - 1 \land T = \text{nil} \land \text{Halt} \)), by Rule (3). Now, by Rule (4),

\[
\text{lastly}(\text{Count} = \text{tips}(\text{Tr}) - 1).
\]

By Rules (5) and (6),

\[
\text{Result} = \text{last}(\text{Count}) + 1 = \text{tips}(\text{Tr}),
\]

and so \( \text{tree}(\text{Tr}) \rightarrow \text{Result} = \text{tips}(\text{Tr}) \).

Moving this out of the compute, we get

\[
\text{tree}(\text{Tr}) \rightarrow \text{Leaves} = \text{tips}(\text{Tr}).
\]

□

Program 3. The last example is a Lucid version of the simple operating system program in the Manna and Waldinger paper.

\[
\text{produce Output using Input}
\]

\[
\text{compute Result using Input}
\]

\[
\text{first Jobqueue = Input}
\]

\[
\text{first Printouts = nil}
\]

\[
\text{Job = hd(Jobqueue)}
\]

\[
\text{next Printouts = Printouts \circ process(Job)}
\]

\[
\text{next Jobqueue = tl(Jobqueue)}
\]

\[
\text{Halt = Jobqueue eq nil}
\]

\[
\text{Result = last(Printouts)}
\]

\[
\text{end}
\]

\[
\text{end}
\]

The produce clause takes a whole stream of Input values and produces a whole stream of Output values. Each Input value will be a list of jobs and the corresponding Output value obtained from the compute clause will be a list of the results of processing these jobs.

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

**Theorem 3.** Inside the compute clause of this program,

\[
\text{sometime}(j \in \text{Jobqueue}) \rightarrow \text{sometime}(\text{Job} = j).
\]

**Proof.** We will actually prove the following lemma:

**Lemma 3.**

\[
\text{sometime}(\text{Jobqueue} = \alpha \circ j \circ \beta) \rightarrow \text{sometime}(\text{Job} = j).
\]

**Proof.** By structural induction on \( \alpha \):

\( \alpha = \text{nil} \). Then \( \text{Jobqueue} = j \circ \beta \rightarrow \text{Job} = j \), since \( \text{Job} = \text{hd}(\text{Jobqueue}) \). Thus, by Rule (3),

\[
\text{sometime}(\text{Jobqueue} = \alpha \circ j \circ \beta) \rightarrow \text{sometime}(\text{Job} = j).
\]

\( \alpha \neq \text{nil} \). Now if \( \text{Jobqueue} = \alpha \circ j \circ \beta \) it is easily shown that \( \neg \text{Halt} \) and next(\( \text{Jobqueue} = \text{tl}(\alpha) \circ j \circ \beta \)). Therefore, by Rule (2),

\[
\text{sometime}(\text{Jobqueue} = \alpha \circ j \circ \beta) \rightarrow \text{sometime}(\text{Jobqueue} = \text{tl}(\alpha) \circ j \circ \beta).
\]

Then applying the induction hypothesis

\[
\text{sometime}(\text{Job} = j).
\]
This completes the proof of the Lemma.

To prove the theorem, we assume
\[ \text{sometime}(j \in \text{Jobqueue}) \]
and note that
\[ j \in \text{Jobqueue} \rightarrow \exists \alpha, \beta \quad \text{Jobqueue} = \alpha \circ j \circ \beta. \]
Thus by Rule (3), \( \text{sometime}(\exists \alpha, \beta \quad \text{Jobqueue} = \alpha \circ j \circ \beta) \) and hence \( \exists \alpha, \beta \quad \text{sometime}(\text{Jobqueue} = \alpha \circ j \circ \beta) \). (The Barcan formula (see [5]) is valid in Lucid.) By the lemma
\[ \text{sometime}(\text{Job} = j). \]
We can discharge the assumption, giving
\[ \text{sometime}(j \in \text{Jobqueue}) \rightarrow \text{sometime}(\text{Job} = j) \]
since Rule (3) was not applied to consequences of the assumption.

These three proofs have closely followed the steps in the original versions. Some slight clumsiness is introduced because of the restrictions on the use of the Deduction Theorem which Lucid imposes.

7 Discussion

It has been shown that proofs using Burstall’s intermittent assertion method can easily be accomplished using Lucid.

A rigorous treatment of the informal method would be based on the modal logic S4 (see [5]). It is interesting to note that in modal logic in general, the Deduction Theorem is not valid (just as it is not in Lucid). However if can be regained by introducing the idea of a strict subproof, which in the natural deduction formulation of the modal logic S4 corresponds to a subproof concentrating on a particular time in the future. Essentially, this is what happens in informal intermittent assertion proofs, and the use of the deduction theorem in such proofs is always justified. However, in Lucid it is not possible to reason exactly the same way because the whole of Lucid, not just that part dealing with \text{sometime} \cdots \text{before}, is based on S4, and if we want the deduction theorem to work in general we must change the whole Lucid formal system to correspond to the natural deduction formulation of S4. This introduces new complications in things like the Lucid Induction Rule, and does not appear to be worth it. Experience to date seems to indicate that it is better to put up with the few restrictions on the use of the deduction theorem.

Acknowledgement

Thanks are due to Tom Cargill for stimulating discussions in which some of the ideas presented here were worked out.

References


