# Is Sometime Ever Better Than Alway?

DAVID GRIES
Cornell University

The "intermittent assertion" method for proving programs correct is explained and compared with the conventional method. Simple conventional proofs of iterative algorithms that compute recursively defined functions, including Ackermann's function, are given.

Key Words and Phrases: correctness of programs, axiomatic method, intermittent assertion method, simulating recursion using iteration, Ackermann function
CR Categories: 5.24

## 1. INTRODUCTION

The "intermittent assertion" method of proving programs correct has begun to attract a good deal of attention. The purpose of this paper is to compare the method, as it is explained in [6], with the now conventional method proposed by Hoare [4].

We shall have to give the latter method a name. "Axiomatic method" will not do because most methods can be axiomatized. "Invariant assertion method," proposed in [6], is unacceptable because it is too long and because the term "invariant" has already been connected with loops. My solution in this paper is to provide short names for both methods. The intermittent assertion method will be called the *sometime method*, for reasons that will become apparent later. By analogy, the conventional method of [4], along with the concept of total correctness (see, for example, [2]), will be called the *alway method*. (*Alway* is poetic for *always*.) It is assumed that the reader is familiar with the *alway method*.

The sometime method has been used mainly to reason about iterative algorithms that compute recursively defined functions, and in this setting it has been thought to be more "natural" than the alway method. In fact, [6] contains a challenge to use the alway method on an iterative algorithm that computes Ackermann's function. We meet this challenge in Section 2. Section 3 outlines the sometime method and presents for comparison a second proof of correctness of the iterative Ackermann algorithm. Section 4 shows how to transform a particular recursive definition scheme into an equivalent iterative algorithm using


This research was supported by the National Science Foundation under Grant MCS76-22360.
Author's address: Computer Science Department, Upson Hall, Cornell University, Ithaca, NY 14853.

the alway method. The scheme was taken from [6]. Finally, the discussion in Section 5 leads to the conclusion that the alway method is to be preferred.

## 2. THE ACKERMANN ALGORITHM AND THE ALWAY METHOD

Ackermann's function $A(m, n)$ is defined for $m, n \geq 0$ by

$$A(m, n) = \begin{cases} m = 0 & \rightarrow n + 1 \\ m \neq 0, n = 0 & \rightarrow A(m - 1, 1) \\ m \neq 0, n \neq 0 & \rightarrow A(m - 1, A(m, n - 1)). \end{cases}$$

The following algorithm to compute $A(m, n)$ uses a "sequence" variable $s$. Each element $si$ of sequence $s = \langle sn, \ldots, s2, s1 \rangle$ satisfies $si \geq 0$, and $n = size(s) \geq 0$ is the length of the sequence. Using a sequence instead of a stack and numbering the elements in reverse order, as done here, simplifies later notation. Element $si$ of $s$ will be referenced within the algorithm by $s(i)$, while $s(..i)$ refers to the possibly empty sequence $\langle s(n), s(n-1), \ldots, s(i) \rangle$. Operation $s \mid x$ denotes the concatenation of value $x$ to sequence $s$. For example, if $size(s) \geq 2$, then $s = s(..3) \mid s(2) \mid s(1)$.

```
{m, n ≥ 0}
s := ⟨m, n⟩;
do size(s) ≠ 1 →
  if s(2) = 0                    → s := s(..3) | s(1) + 1
  □ s(2) ≠ 0 and s(1) = 0 → s := s(..3) | s(2) − 1 | 1
  □ s(2) ≠ 0 and s(1) ≠ 0 → s := s(..3) | s(2) − 1 | s(2) | s(1) − 1
  fi
od
{s = ⟨A(m, n)⟩}
```

This algorithm repeatedly manipulates sequence $s$ until its length is 1. Our problem is to prove that the loop halts and that, when it halts, $s$ contains the desired result. In order to provide means for solving the problem, it seems reasonable to abstract from the algorithm the manner in which sequences are manipulated and to examine this manipulation in a purely mathematical setting. Thus we analyze how a single iteration of the loop can transform any sequence $s'$ into a sequence $s''$ and define a relation $\succ$ between such $s'$ and $s''$.

*Definition* 2.1. The relation $\succ$ on sequences is defined by

(a) $s \mid 0 \mid b \succ s \mid b + 1$, for $b \geq 0$, any sequence $s$
(b) $s \mid a \mid 0 \succ s \mid a - 1 \mid 1$, for $a > 0$, any sequence $s$
(c) $s \mid a \mid b \succ s \mid a - 1 \mid a \mid b - 1$, for $a, b > 0$, any sequence $s$.

Note that for any sequence $s'$ with $size(s') > 1$ there exists exactly one sequence $s''$ such that $s' \succ s''$. For $s'$ with $size(s') \leq 1$ there is no such $s''$.

Given an initial sequence $s = \langle m, n \rangle$, a *number* of iterations of the loop is supposed to transform $s$ into $\langle A(m, n) \rangle$. Since *one* iteration transforms $s$ into $s'$ such that $s \succ s'$, we are led to consider relations $\succ^+$, the transitive closure of $\succ$; $\succ^*$, the reflexive transitive closure; and $\succ^t$ for fixed $t \geq 0$, which represents exactly $t$ applications of $\succ$. The necessary properties are proved in the following lemma.

LEMMA 2.2. *Given $a, b \geq 0$, for any sequence $s$ there exists $t > 0$ such that $s \mid a \mid b \succ^t s \mid A(a, b)$.*

PROOF. The proof is by induction on the lexicographic ordering of pairs of nonnegative integers. We assume the lemma true for $\tilde{a}$, $\tilde{b}$ satisfying $\langle a, b\rangle >_2 \langle \tilde{a}, \tilde{b}\rangle$ and prove it true for $a$, $b$, where $\langle a, b\rangle >_2 \langle \tilde{a}, \tilde{b}\rangle$ is defined by

$$\langle a, b\rangle >_2 \langle \tilde{a}, \tilde{b}\rangle \equiv a > \tilde{a} \textbf{ or } (a = \tilde{a} \textbf{ and } b > \tilde{b})$$

There are three cases to consider, based on the definition of $\succ$.

*Case* $a = 0$. $s \mid 0 \mid b \succ s \mid b + 1 = s \mid A(0, b)$, and $t = 1$. (This is the basis step.)

*Case* $a \neq 0$, $b = 0$. $s \mid a \mid 0 \succ s \mid a - 1 \mid 1$. Since $\langle a, 0\rangle >_2 \langle a - 1, 1\rangle$, by induction there exists $t1$ such that $s \mid a - 1 \mid 1 \succ^{t1} s \mid A(a - 1, 1) = s \mid A(a, 0)$. Thus $s \mid a \mid 0 \succ^{t} s \mid A(a, 0)$ with $t = t1 + 1$.

*Case* $a, b \neq 0$. $s \mid a \mid b \succ s \mid a - 1 \mid a \mid b - 1$. Since $\langle a, b\rangle >_2 \langle a, b - 1\rangle$, by induction there is a $t1$ such that $s \mid a - 1 \mid a \mid b - 1 \succ^{t1} s \mid a - 1 \mid A(a, b - 1)$. Since $\langle a, b\rangle >_2 \langle a - 1, A(a, b - 1)\rangle$, by induction there is a $t2$ such that $s \mid a - 1 \mid A(a, b - 1) \succ^{t2} s \mid A(a - 1, A(a, b - 1)) = s \mid A(a, b)$. Hence $s \mid a \mid b \succ^{t} s \mid A(a, b)$ with $t = 1 + t1 + t2$. This ends the proof.

Now consider $s = \langle m, n\rangle$. By the lemma, there is a $t > 0$ such that $s = \langle m, n\rangle \succ^{t} \langle A(m, n)\rangle$. Furthermore, $t$ is unique, since for any sequence $s'$ there is at most one $s''$ such that $s' \succ s''$ and there is *no* $s''$ such that $\langle A(m, n)\rangle \succ s''$. Hence for any sequence $s$ satisfying $\langle m, n\rangle \succ^{*} s$ there is a unique value $\tau(s)$, $\tau(s) \geq 0$, a function of $s$, such that $s \succ^{\tau(s)} \langle A(m, n)\rangle$. We therefore take as the necessary loop invariant

$$P\text{: } \langle m, n\rangle \succ^{*} s \succ^{\tau(s)} \langle A(m, n)\rangle.$$

$P$ is initially true with $s = \langle m, n\rangle$ and $\tau(s) = \tau(\langle m, n\rangle)$; upon termination ($P$ **and** $size(s) = 1$) implies the desired result. That $P$ remains true is almost trivial to show, since $\succ$ was expressly defined so that execution of the loop body with variable $s$ containing a value $s'$ would change $s$ to the unique value $s''$ satisfying $s' \succ s''$. For a termination function we use $\tau(s)$, which is decremented by 1 each time the loop body is executed.

*Remark* 1. The invariant $P$ was not as easy to derive as the above description indicates, although it should have been. □

*Remark* 2. Reference [6] says that the alway method requires two separate proofs to establish total correctness, one to show partial correctness and the other to show termination. While this is true, the example indicates that a proper choice of invariant can make the proof of termination almost trivial. □

*Remark* 3. The formalization of the method for proving termination has previously been done in two ways, which we summarize here.

(1) Strong Termination. Derive an integer function $t(\bar{x})$ of the program variables $\bar{x}$, show that $t \geq 0$ whenever the loop is still executing, and show that each execution of the loop body decreases $t$ by at least 1. For a loop **do** $B \rightarrow S$ **od** with invariant $P$ this means proving that

$$(P \textbf{ and } B) \Rightarrow t \geq 0 \quad \text{and} \quad \{P \textbf{ and } B\}\ T := t;\ S\ \{t \leq T - 1\}$$

where $T$ is a new variable.

(2) Weak Termination. Choose a "well-founded" set $(W, \succ)$, i.e., $\succ$ is a partial ordering with the property that for any $w$ in $W$ there is no infinite chain $w \succ w1 \succ w2 \succ \cdots$. Then choose a function $f(\bar{x})$ of the program variables $\bar{x}$ and prove

that

$$\{P \textbf{ and } B\}\ \tilde{w} := \tilde{x};\quad S\{f(\tilde{w}) \succ f(\tilde{x})\}$$

where $\tilde{w}$ is a new set of variables.

Under the reasonable assumption that nondeterminism is bounded [2], the two methods are equivalent. The first induces a function $f(\tilde{x}) = t(\tilde{x})$ and a well-founded ordering defined by $f(\tilde{x}) \succ f(\tilde{y})$ iff ($P$ **and** $B$) implies $t(x) > t(y) \geq 0$. Given bounded nondeterminism and a proof by the second method, one can show that the number of iterations of the loop for any initial state $\tilde{x}$ is bounded, and we choose $t(\tilde{x})$ as that bound.

In this situation, we prefer using strong termination. Having an explicit bound on the number of iterations of each loop is indeed useful if one wants to analyze the execution time of an algorithm. □

## 3. THE SOMETIME METHOD

The sometime method was invented as early as 1972 by R.M. Burstall and presented in a 1974 IFIP Congress lecture [1]. Burstall felt that it would have intuitive appeal, since "it is very much like checking out a program by doing a step-by-step hand simulation" [1]. His student, R.W. Topor, noticed after the fact that D.E. Knuth actually used a similar style of argument on a few problems [5], but Knuth had not explained it as a general method for reasoning about programs. Manna and Waldinger bestowed the term "intermittent assertion" on the method in their 1978 paper [6], which is responsible for the current wave of interest in the method. Topor [7] also uses it to prove correct a version of the Schorr-Waite algorithm for marking nodes of a directed graph; a proof by the alway method appears in [3].

The method involves associating an assertion with a point in the algorithm with the intention that *sometime* during execution control will pass through that point with the assertion true, but that it need not be true *every* time control passes that point. Based on the fact that sometime control will be at that point with the assertion true, one then argues that control will later reach another point (e.g., the end of the algorithm) with another assertion true (e.g., the output assertion).

To illustrate the method, let us first of all rewrite the iterative Ackermann algorithm to include labels, which are necessary in order to discuss it. This algorithm is actually a restatement of that given in [6], paraphased to make it and its proof as clear as we possibly could.

```
start: s := ⟨m, n⟩;
do test: size(s) ≠ 1 →
   if s(2) = 0                    → s := s(..3) | s(1) + 1
   □ s(2) ≠ 0 and s(1) = 0        → s := s(..3) | s(2) − 1 | 1
   □ s(2) ≠ 0 and s(1) ≠ 0        → s := s(..3) | s(2) − 1 | s(2) | s(1) − 1
   fi
od;
finish: skip
```

The sometime method allows one to use an assertion that is true at a point of

a program, but not necessarily always. A typical example is contained in the following lemma.

LEMMA 3.1. *If sometime size(s)* $\geq 2$ **and** $s = \tilde{s} \mid a \mid b$, $a, b \geq 0$, *at test, then sometime* $s = \tilde{s} \mid A(a, b)$ *at test.*

PROOF. Suppose $s = \tilde{s} \mid a \mid b$ at *test*. The lemma is proved by induction on the lexicographic ordering $>_2$ on pairs of nonnegative integers. Thus we assume the lemma holds for any sequence $\tilde{s}$ and pair $\langle \tilde{a}, \tilde{b} \rangle$ satisfying $\langle a, b \rangle >_2 \langle \tilde{a}, \tilde{b} \rangle$, and we show that it holds for any sequence $\tilde{s}$ and $\langle a, b \rangle$. The reasoning is based on an informal understanding of how programs are executed. There are three cases to consider, corresponding to the three guarded commands of the alternative statement of the loop body.

*Case* $a = 0$: $s = \tilde{s} \mid 0 \mid b$ *at test.* Since $size(s) \neq 1$ the loop body is executed, the first guarded command is executed, $s$ is changed to $s = \tilde{s} \mid b + 1$, and control returns to *test* with $s = \tilde{s} \mid b + 1 = \tilde{s} \mid A(0, b)$.

*Case* $a \neq 0$, $b = 0$: $s = \tilde{s} \mid a \mid 0$ *at test.* Note that $A(a, 0) = A(a - 1, 1)$. Execution of the second guarded command changes $s$ to $\tilde{s} \mid a - 1 \mid 1$ and control returns to *test*. Since $\langle a, 0 \rangle >_2 \langle a - 1, 1 \rangle$, by induction control will at some point reach *test* with $s = \tilde{s} \mid A(a - 1, 1) = \tilde{s} \mid A(a, 0)$. Thus the lemma is established in this case.

*Case* $a, b \neq 0$: $s = \tilde{s} \mid a \mid b$ *at test.* The third guarded command is executed, $s$ becomes $\tilde{s} \mid a - 1 \mid a \mid b - 1$, and control returns to *test*. Since $\langle a, b \rangle >_2 \langle a, b - 1 \rangle$, by induction control will return to *test* at some point with $s = \tilde{s} \mid a - 1 \mid A(a, b - 1)$. Since $\langle a, b \rangle >_2 \langle a - 1, A(a, b - 1) \rangle$, by induction further execution is guaranteed to cause control to reach *test* again, with $s = \tilde{s} \mid A(a - 1, A(a, b - 1)) = \tilde{s} \mid A(a, b)$. The lemma is established.

This is typical of the reasoning used with the sometime method. Notice how one is relying on informal "hand simulation" of the algorithm, but with an assertion that represents a set of possible initial states (e.g., $s = \tilde{s} \mid a \mid b$ **and** $a, b \geq 0$), rather than one particular set of initial values. This is an informal way of performing what has been called "symbolic evaluation."

Now suppose execution of the algorithm begins with $m, n \geq 0$. Control reaches *test* with $s = \langle m, n \rangle$. By the lemma, control will reach *test* again with $s = \langle A(m, n) \rangle$, the loop will terminate because $size(s) = 1$, and control will reach *finish* with $s(1) = A(m, n)$. Thus we have proved the following theorem.

THEOREM 3.2. *If sometime* $m, n \geq 0$ *at start, then sometime* $s(1) = A(m, n)$ *at finish.*

## 4. A TRANSFORMATION SCHEME

In [6] it is proved using the sometime method that a recursive definition (or algorithm) of the form

$$F(x) = \begin{cases} p(x) & \rightarrow f(x) \\ \textbf{not } p(x) & \rightarrow h(F(g1(x)), F(g2(x))) \end{cases}$$

under the assumptions

(1) $p, f, g1, g2$, and $h$ are total functions
(2) $h$ is associative: $h(u, h(v, w)) = h(h(u, v), w)$ for all $u, v, w$
(3) $e$ is the left identity of $h$: $h(e, u) = u$ for all $u$

is equivalent to the following iterative algorithm. The algorithm uses a sequence variable $s$ and a simple variable $z$:

$\{F(x)$ well defined$\}$
$s, z := \langle x \rangle, e;$
**do** $s \neq \langle\ \rangle \rightarrow$
  **if** $\quad p(s(1)) \rightarrow s, z := s(..2), h(z, f(s(1)))$
  $\square$ **not** $p(s(1)) \rightarrow s := s(..2) \mid g2(s(1)) \mid g1(s(1))$
  **fi**
**od**
$\{z = F(x)\}$

We want to prove the same thing using the alway method. It is tempting to apply the technique used to prove the Ackermann algorithm correct, and indeed it works like a charm.

We first note that there must be a well-founded ordering $\succ$ defined by ($F(x)$ well defined **and not** $p(x)$) implies ($x \succ g1(x)$ **and** $x \succ g2(x)$). This means that there is no infinite chain $x \succ x1 \succ \cdots$ such that **not** $p(x_i)$ if $F(x)$ is well defined, and that we can use the ordering $\succ$ to prove something by induction, the way $>_2$ was used in Section 2.

In attempting to define an ordering on sequences as in Section 3, we find that we must also take into account the value of simple variable $z$. So we define instead a relation $\succ$ on pairs $(s; z)$, where $s$ is a sequence and $z$ a value.

*Definition* 4.1. Relation $\succ$ is defined for any sequence $s$ and values $x$ and $z$ as follows:

(a) if $p(x)$, then $(s \mid x; z) \succ (s; h(z, f(x)))$

(b) if **not** $p(x)$, then $(s \mid x; z) \succ (s \mid g2(x) \mid g1(x); z)$.

LEMMA 4.2. *Given $x$ for which $F(x)$ is well defined, for any sequence $s$ and value $z$ there exists a $t \geq 0$ such that $(s \mid x; z) \succ^t (s; h(z, F(x)))$.*

PROOF. The proof is by induction on the ordering $\succ$ described above. There are two cases, corresponding to the cases in Definition 4.1.

*Case* $p(x)$: $(s \mid x; z) \succ (s; h(z, f(x))) = (s; h(z, F(x)))$, and $t = 1$.

*Case* **not** $p(x)$: We have

| | | |
|---|---|---|
| $(s \mid x; z)$ | | |
| $\succ$ | $(s \mid g2(x) \mid g1(x); z)$ | by definition |
| $\succ^{t1}$ | $(s \mid g2(x); h(z, F(g1(x))))$ | by induction, since $x \succ g1(x)$ |
| $\succ^{t2}$ | $(s; h(h(z, F(g1(x))), F(g2(x))))$ | by induction, since $x \succ g2(x)$ |
| $=$ | $(s; h(z, h(F(g1(x)), F(g2(x)))))$ | by associativity of $h$ |
| $=$ | $(s; h(z, F(x)))$ | by definition of $F$. |

Thus $(s \mid x; z) \succ^t (s; h(z, F(x)))$ with $t = 1 + t1 + t2$. This completes the proof of Lemma 4.2.

Now note that Lemma 4.2 implies the existence of a $t \geq 0$ such that

$$(\langle x \rangle; e) \succ^t (\langle\ \rangle; h(e, F(x))) = (\langle\ \rangle; F(x)).$$

We define a function $\tau$ as in Section 3 and use the loop invariant

$$P\text{: } (\langle x \rangle; e) \succ^* (s; z) \succ^{\tau((s;\ z))} (\langle\ \rangle; F(x)).$$

We leave the simple proof that $P$ is indeed the desired invariant to the reader; the necessary termination function is $\tau$ of the invariant $P$. To the reader we also leave the proof that if $F(x)$ is not well defined then the algorithm does not terminate.

## 5. DISCUSSION OF THE METHODS

Reference [6, p. 163] has said that all known proofs of the Ackermann algorithm using conventional methods are extremely complicated. The proof in Section 2 is offered to support our conjecture that alway method proofs need be no more complicated than sometime method proofs. The material in Section 4 offers hope that iterative algorithms that compute recursively defined functions—a major stronghold of the sometime method—will quietly succumb to the alway method. It is simply a matter of learning the necessary techniques. (In this case the technique is, quite simply, to define a relation $\succ$ such that $s' \succ s''$ if one iteration of the loop transforms the loop variables $s'$ into $s''$, and then to investigate this relation.) The authors of [6] quite rightly imply that a proof method should be "natural," but "naturalness" in any field of endeavor must be learned.

Let us compare the two methods, where our knowledge of the practical use of the sometime method is based solely on the examples given in [6]. We can begin by comparing the two proofs of the Ackermann algorithm. Here one notices a strong similarity. Lemmas 2.2 and 3.1 lie at the heart of the proofs, and both are proved by induction over the ordering $>_2$. Each proof breaks down into three similar cases. The main difference is that one proof requires a detailed analysis of an algorithm, while the other requires an analysis only of a simple relation that took four lines to define. And herein lies what we would call a major drawback to the sometime method, which we now try to explain.

Any algorithm is based on certain properties of the objects it manipulates and it seems desirable to keep a clear distinction between these properties and the algorithm that works on the objects. Thus in the alway method proof of Section 2, Definition 2.1 and Lemma 2.2 define, describe, and prove properties of sequences in a completely mathematical setting. Then the proof of the algorithm follows easily by considering the algorithm together with these properties. A change in the algorithm does not destroy the neat mathematical properties, but only perhaps their relevance. In addition, one can work with mathematical properties that have been proven by others, without having to understand their proof. The principle of *separation of concerns* is being adhered to clearly in the alway method.

The sometime method, on the other hand, as explained in current proofs, seems to encourage confusion of properties of the objects and the algorithm itself. Thus, in Section 3, all parts of the complete proof, including the use of induction based on execution of a program, were packaged together.

Through programming, we hope to learn to cope with complexity (and to teach others how to cope) using principles like *abstraction* and *separation of concerns.* The alway method encourages the use of and gives insight into these principles; the sometime method seems by its very nature to discourage their use, and thus seems to be a step backward.

It is true that an alway method proof may have more parts to it. For example,

once the mathematical properties were stated and proved in Section 2, it was necessary to relate them to the algorithm itself, using a loop invariant and termination function. We gladly accept this "extra" work, for in return we gain a better understanding and have a proof that is clearly structured into its component parts.

One referee offered the following way of looking at the two methods, which may help the reader. For the Ackermann algorithm, using the sometime method one

(1) makes a hypothesis that the "snapshots" of the program variables at various points of execution form a finite sequence
(2) proves the correctness of the hypothesis by induction on program execution.

In the alway method, as used here, one

(1) defines a sequence and proves that it exists and is of bounded length
(2) shows that the sequence matches exactly the sequence of snapshots.

ACKNOWLEDGMENTS

I wish to thank R. Constable and G. Levin for discussions that led to the invariant used in the Ackermann algorithm. I am indebted to J. Donahue, G. Levin, and J. Williams for critically reading drafts of this paper.

REFERENCES

1. BURSTALL, R.M. Program proving as hand simulation with a little induction. Proc. IFIP Congress 1974, Amsterdam, The Netherlands, pp. 308–312.
2. DIJKSTRA, E.W. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1976.
3. GRIES, D. The Schorr-Waite graph marking algorithm. To appear in *Acta Informatica.*
4. HOARE, C.A.R. An axiomatic basis for computer programming. *Comm. ACM 12,* 10 (Oct. 1969), 576–580, 583.
5. KNUTH, D.E. *The Art of Computer Programming, Vol. I.* Addison-Wesley, Reading, Mass., 1968.
6. MANNA, Z., AND WALDINGER, R. Is "sometime" sometimes better than "always"? *Comm. ACM* 21, 2 (Feb. 1978), 159–172.
7. TOPOR, R.W. A simple proof of the Schorr-Waite garbage collection algorithm. To appear in *Acta Informatica.*

Received June 1978; revised January and May 1979