IS SOMETIMES EVER BETTER THAN ALWAYS?

by

David Gries[†]

TR78-343

Department of Computer Science
Cornell University
Ithaca, New York 14853

[*]This research was supported by the National Science Foundation under Grant MCS76-22360.

IS SOMETIMES EVER BETTER THAN ALWAYS?

by

David Gries[†]

Cornell University

Abstract

The "intermittent assertion" method for proving programs correct is explained and compared to the conventional axiomatic method. Simple axiomatic proofs of iterative algorithms that compute recursively defined functions, including Ackermann's function, are given. A critical examination of the two methods leads to the opinion that the axiomatic method is preferable.

[†]This research was supported by the National Science Foundation under Grant MCS76-22360.

1. Introduction

The so-called "intermittent assertion" method for proving programs correct [1] has begun to attract a good deal of attention, so much that it can no longer be ignored. The purpose of this paper is to compare the method -- as it is explained in [1] -- with the more conventional axiomatic method. It is assumed that the reader is familiar with the axiomatic method [2], together with the concept of total correctness -- see e.g. [3].

The intermittent assertion method is used in [4] to argue informally about several algorithms. The method involves associating an assertion with a point in the algorithm with the intention that at some time 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 at some time 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).

Burstall discusses the idea in [5], while Manna and Waldinger [1] are responsible for the current wave of interest in the technique. Topor [6] also uses it to prove correct a version of the Schorr-Waite algorithm for marking nodes of a directed graph; an axiomatic proof appears in [7].

The intermittent assertion method has been mainly used to reason about iterative algorithms that compute recursively defined functions, and in this setting it has been thought to be more "natural" than the axiomatic method. In fact, [1] contains a challenge to use the axiomatic method on an iterative algorithm that computes Ackermann's function. Sections 2 and 3 contain proofs of this algorithm using the two methods, which the reader is invited to compare. Section 4 shows how to transform a particular recursive definition scheme into an equivalent iterative algorithm using the axiomatic method. The scheme was taken from [1]. Section 5 gives arguments that lead to the conclusion that the axiomatic method is to be preferred.

2. The Intermittent Assertion 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. Numbering the elements in reverse order, as I have done, 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 $\langle sn,s_{n-1},\ldots,si\rangle$. Operation s|x denotes the concatenation of element x to sequence s. For example, if $size(s)\geq 2$, then s = s(..3) | s(2) | s(1). The algorithm contains labels needed to discuss the "flow of control" in the intermittent assertion method.

```
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
```

Remark The above algorithm is a paraphrase of that given in [1], which was written in terms of conditional and *goto* statements and arrays. The use of guarded commands and sequences, together with the label *test* on the guard of the loop, leads to a clearer algorithmic description and proof. *end of remark*

The intermittent assertion method allows one to use an assertion that is true at a point of a program, but only sometimes. A typical example is contained in the following lemma.

Lemma 2.1. If sometime $size(s) \geq 2$ and $s = \hat{s}|a|b$ at *test*, then sometime $s = \hat{s}|A(a,b)$ at *test*.

Proof. Suppose $s = \hat{s}|a|b$ at *test*. The lemma is proved by induction on the lexicographic ordering $\succ$ on pairs of nonnegative integers, which is defined as follows:

$$\langle a,b\rangle \succ \langle \hat{a},\hat{b}\rangle \text{ if and only if } a>\hat{a} \text{ \underline{or} } (a=\hat{a} \text{ \underline{and} } b>\hat{b}).$$

Thus we assume the lemma holds for any sequence $\hat{s}$ and pair $\langle \hat{a},\hat{b}\rangle$ satisfying $\langle a,b\rangle \succ \langle \hat{a},\hat{b}\rangle$, and show that it holds for any sequence $\hat{s}$ 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: $s = \hat{s}|0|b$ at *test*. Since $size(s) \neq 1$ the loop body is executed, the first guarded command is executed, s is changed to $s = \hat{s} \mid b+1$, and control returns to *test* with $s = \hat{s}|b+1 = \hat{s}|A(0,b)$.

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

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

This is typical of the reasoning used with intermittent assertions.

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:

Theorem 2.2. If some time $m,n\geq 0$ at start, then some time $s(1) = A(m,n)$ at finish.

This proof is a paraphrase of that in [1]; I have tried to make the reasoning as concise and clear as possible.

3. The Axiomatic Method

We now give a proof of correctness of the algorithm using the axiomatic approach. We first define a relation $\succ$ on sequences. The reader will note that $p \succ q$ if and only if one execution of the loop body with s=p transforms s into q.

Definition 3.1. The relation $\succ$ on sequences is defined by

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

Note that for any sequence p with size(p)>1 there exists exactly one sequence q such that $p \succ q$. For p with size(p)$\leq$1 there is no such q. Most of the work in proving correctness is contained in the following

Lemma 3.2. Given $a,b \geq 0$, for any sequence s there exists $t \geq 0$ such that $s|a|b \overset{t}{\succ} s|A(a,b)$ (i.e. one gets from s|a|b to s|A(a,b) by t applications of $\succ$).

Proof. The proof is by induction on the lexicographic ordering of pairs of nonnegative integers. We assume the lemma true for $\hat{a},\hat{b}$ satisfying $\langle a,b\rangle \gtrdot \langle \hat{a},\hat{b}\rangle$ and prove it true for a,b . There are three cases to consider, based on the definition of $\succ$.

case a=0: $s|0|b \succ s|b+1 = s|A(0,b)$, and t=1.

case $a \neq 0, b=0$: $s|a|0 \succ s|a-1|1$. Since $\langle a,0\rangle \gtrdot \langle a-1,1\rangle$, by induction there exists t1 such that $s|a-1|1 \overset{t1}{\succeq} s|A(a-1,1) = s|A(a,0)$. Thus $s|a|0 \overset{t}{\succeq} s|A(a,0)$ with t = t+1.

case $a,b \neq 0$: $s|a|b \succ s|a-1|a|b-1$. Since $\langle a,b\rangle \gtrdot \langle a,b-1\rangle$, by induction there is a t1 such that $s|a-1|a|b-1 \overset{t1}{\succeq} s|a-1|A(a,b-1)$. Since $\langle a,b\rangle \gtrdot \langle a-1,A(a,b-1)\rangle$, by induction there is a t2 such that $s|a-1|A(a,b-1) \overset{t2}{\succeq} s|A(a-1,A(a,b-1)) = s|A(a,b)$. Hence $s|a|b \overset{t}{\succeq} s|A(a,b)$ with t=1+t1+t2. This ends the proof.

For convenience, we give the algorithm again, without the labels.

```
{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)>}
```

One way to derive a useful loop invariant is to weaken the result assertion (i.e. s=<A(m,n)>) to include the initial condition (i.e. s=<m,n>). To do this we make use of relation $\succ$. Note that there is a t≥0 such that <m,n> $\overset{t}{\succ}$ <A(m,n)>. Furthermore, t is unique, since for any sequence p there is at most one q such that s$\succ$q, and there is no q such that <A(m,n)>$\succ$q. Hence, for any sequence p such that <m,n> $\overset{*}{\succ}$ p there is a unique $\mathcal{T}$(p), $\mathcal{T}$(p)≥0, an integer function of p, such that <m,n>$\overset{*}{\succ}$ p $\overset{\mathcal{T}(p)}{\succ}$ <A(m,n)>. We therefore take as our loop invariant P:

P: <m,n>$\overset{*}{\succ}$ s $\overset{\mathcal{T}(s)}{\succ}$ <A(m,n)>.

P is initially true with s=<m,n> and $\mathcal{T}$(s)=$\mathcal{T}$(<m,n>); 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 p would change s to the unique value q satisfying p$\succ$q. For a termination function we take $\mathcal{T}$(s) that was just defined, 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. end of remark 1.

Remark 2. Reference [1] says that the axiomatic approach 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. end of remark 2.

Remark 3. The formalization of the method for proving termination has previously been done in two ways, which we summarize here. (1) Derive an integer function $t(\tilde{x})$ of the program variables $\tilde{x}$; 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 this means proving that

$$(P \text{ and } B) \Rightarrow t \geq 0 \qquad \text{and}$$
$$\{P \text{ and } B \text{ and } t=c\}\ S\ \{t \leq c-1\} \qquad \text{for all } c.$$

(2) 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 $w \succ w1 \succ w2 \succ \ldots$. Then choose a function $f(\tilde{x})$ of the program variables $\tilde{x}$ and prove that

$$\{P \text{ and } B \text{ and } f(\tilde{x})=w\}\ \ S\ \ \{w \succ f(\tilde{x})\} \qquad \text{for any } w \text{ in } W.$$

The two methods are equivalent. The first induces a function $f(\tilde{x}) = t(\tilde{x})$ and a well-founded ordering $\succ$ defined by $f(\tilde{x}) \succ f(\tilde{y})$ if (P and B) implies $t(x) > t(y) \geq 0$. Given a proof by the second method, under the reasonable assumption that nondeterminism is bounded (see [3]), choosing $t(\tilde{x})$ to be the length of the longest sequence $f(\tilde{x}) \succ w1 \succ \ldots$ yields a proof by the second method.

In this situation, I prefer the first method to the second; it is easier to state, just as easy to use, and makes more sense to the majority of programmers. end of remark 3

4. A Transformation Scheme

In [1] it is proved using intermittent assertions that a recursive definition (or algorithm) of the form

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

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:= <x>,e;
 do s≠<> →
     if     p(s(1)) → s,z:= s(..2), h(z,f(s(1)))
      ▯ not p(s(1)) → s:= s(..2) | g2(s(1)) | g1(s(1))
     fi
 od
 {z=F(x)}
```

We want to prove the same thing using the axiomatic 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 $\leq$ defined by

(F(x) well defined <u>and</u> <u>not</u> p(x))   implies
(x $\gtrdot$ g1(x) <u>and</u> x $\gtrdot$ g2(x)).

This means that there is no infinite chain x $\gtrdot$ x1 $\gtrdot$ ... if F(x) is well defined, and that we can use the ordering $\gtrdot$ to prove something by induction, the way $\gtrsim$ was used in Section 3.

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.

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

(a) if p(x), then (s|x; z) $\succ$ (s; h(z,f(x)))
(b) if <u>not</u> p(x), then (s|x; z) $\succ$ (s|g2(x)|g1(x); z)

<u>Lemma 4.2</u>. Given x for which F(x) is well defined, for any sequence s and value z there exists a t≥0 such that

$$(s|x;\ z)\ \overset{t}{\succ}\ (s;\ h(z,F(x))).$$

<u>Proof.</u> The proof is by induction on the ordering $\gtrdot$ described above. There are two cases, corresponding to the cases in definition 4.1:

<u>case p(x)</u>: (s|x; z) $\succ$ (s; h(z,f(x))) = (s; h(z,F(x))), and t=1.

<u>case not p(x):</u>   We have:

	(s|x; z)	
$\succ$	(s|g2(x)|g1(x); z)	by definition
$\overset{t1}{\succ}$	(s|g2(x); h(z,F(g1(x))))	by induction, since x $\gtrdot$ g1(x)
$\overset{t2}{\succ}$	(s; h(h(z,F(g1(x))),F(g2(x))))	by induction, since x $\gtrdot$ 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 $(s|x;\ z) \overset{t}{\succ} (s;\ h(z,F(x)))$ 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

$$(\langle x \rangle ; e) \overset{t}{\succ} (\langle\rangle ;\ h(e,F(x))) = (\langle\rangle ;\ F(x)),$$

we define a function $\mathcal{T}$ as in Section 3, and use the loop invariant

$$P:\ (\langle x \rangle ;\ e) \overset{*}{\succ} (s;z) \overset{\mathcal{T}((s;z))}{\succ} (\langle\rangle ;\ F(x)).$$

We leave the simple proof that P is indeed the desired invariant to the reader; the necessary termination function is $\mathcal{T}$ 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 [1] said that all known proofs of the Ackermann algorithm using conventional methods were extremely complicated. The proof in Section 3 is offered to support my conjecture that axiomatic proofs need be no more complicated than intermittent assertion proofs. The material in Section 4 offers hope that iterative algorithms that compute recursively defined functions -- a major stronghold of the intermittent assertion method -- will quietly succumb to the axiomatic method. It is simply a matter of learning the necessary techniques. The authors of [1] quite rightly imply that a proof method should be 'natural', but 'naturalness' in any field of endeavor must be learned.

The reader should note that the intermittent assertion method has not yet been formalized. The major reference on the subject, [1], explains the method by example only, and the examples are based only upon an informal understanding of how programs are executed. This is not a criticism; it takes time and thought to make progress in research. But it does mean that one should regard claims made about the method as only enthusiastic opinion. For example, in [1] it is proven that any axiomatic proof can be mechanically translated into an intermittent assertion proof, but it is claimed without proof that going the other way is impossible. It is also maintained that the intermittent assertion method is strictly more powerful than the axiomatic method. To argue against these statements is pointless until the intermittent assertion method has been properly defined.

Let us now compare the two methods, where our knowledge of the intermittent assertion method is based solely on the examples given in [1]. We can begin by cómparing the two proofs of the Ackermann algorithm. Here one notices a strong similarity. Lemmas 2.1 and 3.2 lie at the heart of the proofs, and both are proved by induction over the ordering $\geq_2$. Each proof breaks down into 3 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 4 lines to define. And herein lies what I would call a major drawback to the intermittent assertion method, which I will now try to explain.

Any algorithm is based on certain properties of the objects it manipulates, and it seems to me desirable to keep a clear distinction between these properties and the algorithm that works on the objects. Thus, in the axiomatic proof of Section 3, Definition 3.1 and Lemma 3.2 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 how they are used in the proof. 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 axiomatic approach.

The intermittent assertion 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 2 the fact that there is a nice ordering of sequences is hopelessly entangled in the proof of algorithmic correctness. It should be noted that the proof given in Section 2 is a paraphrase of that given in [1], and it is designed to clarify and not obscure the method. This proof seems to be typical of intermittent assertion proofs.

It is true that an axiomatic proof may have more parts to it. For example, once the mathematical properties were stated and proved in Section 3, it was necessary to relate them to the algorithm itself, using a loop invariant and termination function. I gladly accept this "extra" work, for in return I gain a better understanding and have a proof that is clearly structured into its component parts.

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 axiomatic method encourages the use of and gives insight into these principles; the intermittent assertion method seems by its very nature to discourage their use, and thus seems to be a step backward.

A symptom of this backward step is the reintroduction of time. The beauty and elegance of Hoare's axiomatic method was that it taught us to understand an algorithm as a mathematical entity instead of a program to be executed by a computer, and we can now bring to bear on the programming task all our mathematical training. The reintroduction of time confuses the issue and appears to be a step backward.

It has been asserted that time must be introduced in order to formally prove concurrent nonterminating programs correct. Reference [1] goes so far as to say that "the standard tools for proving correctness of terminating programs, input-output specifications, and invariant assertions, are not appropriate for continuously operating programs." Having myself participated in extending the axiomatic techniques to this class of programs (see e.g. [8]) I fail to see how the authors can make this claim.

Let us briefly discuss possible formalization of the intermittent assertion method. One way to do this would be to give "axiomatic" proof rules for the various constructs -- an attempt in this direction has already been made [9]. My opinion (not a claim) is that this will likely lead to complex, unmanageable proof rules. This opinion is based on the complexity of the argument used in the proof in Section 2. The argument had to include not only the normal kind of induction typical for loops, but also two successive induction steps based on the ordering $\succcurlyeq$. A proof rule to formalize the method as explained in this example is going to be more complex than in the axiomatic approach. Another way to formalize is to introduce "dynamic logic;" I fear this will be too complex for the gain it achieves.

Again, the beauty of the axiomatic approach is partly in the simplicity of the proof rules, although much mathematical manipulation may be necessary in order to simplify assertions, etc. For example, the proof rule for the simple loop $\underline{\text{do}}\ B \rightarrow S\ \underline{\text{od}}$ is

$$\frac{\begin{array}{l}\{P\ \underline{\text{and}}\ B\}\ S\ \{P\}\\ (P\ \underline{\text{and}}\ B) \Rightarrow t \geq 0\\ \{P\ \underline{\text{and}}\ B\}\ T := t;S\ \{t \leq T-1\}\end{array}}{\{P\}\ \underline{\text{do}}\ B \rightarrow S\ \underline{\text{od}}\ \{P\ \underline{\text{and}}\ \underline{\text{not}}\ B\}}$$

P:	loop invariant
t:	termination function
T:	extra variable

Finally, let me comment on the difficult of deriving useful loop invariants. It is true that deriving the invariant for the Ackermann algorithm was not as easy as might be inferred from the discussion, and I am grateful to Manna and Waldinger for challenging me to find it. However, finding loop invariants is becoming easier and easier to those who practice the method. More and more complicated algorithms are succumbing to the approach. It is simply a matter of experience, as illustrated by the steady progress being made.

For me, the loop invariant is a crisp, clear way of understanding a loop, and for me finding an invariant is the prime way to develop or understand a loop. I believe that all good programmers use loop invariants, in that they look at the "general picture" or state of affairs before each loop iteration. All we are requiring when asking for an invariant is a precise definition of what the programmer has up till now been doing in a vague, imprecise way.

To conclude, all the arguments seem to me to be on the side of that axiomatic method, and it remains the method that I will teach and practice, until other arguments convince me otherwise.

Acknowledgements. I wish to thank Bob Constable and Gary Levin for discussions that led to the invariant used in the Ackermann algorithm. I am indebted to Jim Donahue, Gary Levin and John Williams for critically reading drafts of this paper.

References

[1] Manna, Z. and R. Waldinger. Is "sometime" sometimes better than "always"? CACM 21 (Feb 78), 159-171.

[2] Hoare, C.A.R. An axiomatic basis for computer programming. CACM 12 (Oct 69), 576-580, 583.

[3] Dijkstra, E.W. A Discipline of Programming, Prentice Hall, 1976.

[4] Knuth, D.E. The Art of Computer Programming, Vol. I, Addison-Wesley, Reading, MASS. 1968.

[5] Burstall, R.M. Program proving as hand simulation with a little induction. Proc. IFIP Congress 1974, Amsterdam. (308-312).

[6] Topor, R.W. A simple proof of the Schorr-Waite garbage collection algorithm, to appear in Acta Informatica?

[7] Gries, D. The Schorr-Waite graph marking algorithm. Computer Science, Cornell University, 1977. Submitted to Acta Informatica.

[8] Gries, D. An exercise in proving parallel programs correct. CACM 20 (Dec 77), 921-930.

[9] Soundararajan, N. Axiomatic proofs of total correctness of programs. NCSDCT, Tata Inst. of Fundamental Research, Bombay, India, 1978.