A LOGIC FOR EXPRESSIONS WITH SIDE-EFFECTS

Hans-J. Boehm

TR 81-478
November 1981

Department of Computer Science
Cornell University
Ithaca, New York 14853

A Logic for Expressions With Side-Effects

Hans-J. Boehm
Cornell University

This paper presents a simple programming logic LES, which is particularly well suited for reasoning about so-called expression languages, i.e. languages that incorporate imperative features into expressions rather than distinguishing between expressions and statements. An axiomatization of a simple programming language is presented using this formalism. It is shown that this axiomatization is relatively complete, roughly in the sense of [Coo 76].

Introduction

Most existing formalisms for reasoning about programs insist on a clear distinction between "statements" and "expressions" - exceptions known to the author are [Kow 77], [Pri 77], and [Schw 78]. It is assumed that statements are executed only for effect and therefore produce no value, whereas expressions have no effect and only produce a value. At the same time, few programming languages enforce this distinction, and many so-called expression languages (e.g. Algol 68, Russell) do not distinguish between the two at all.

The purpose of this paper is to present a programming logic LES which does not impose any restrictions on this aspect of the underlying programming language, though the logic can be justified even for pure statement languages. It is constructed by adding a single primitive to a logic for the underlying data domain. Thus we can argue that it is really no more complex to axiomatize an expression language than it is

---

This research was supported in part by NSF grant no. MCS 79-01048

to axiomatize a pure statement language such as the one Hoare uses [Hoa 69].

To illustrate various points it is useful to define a very simple programming language which exhibits the desired characteristics.

The Programming Language

The basic language used for illustration will be a pure expression language in that every construct will yield a value. It may or may not change the state of the computation. The constructs to be considered are:

1. The simple expression

   a op b

   where op is +, * etc.

   A and b are evaluated in that order. Op is applied to their respective values to yield the result. The total effect on the state is equivalent to evaluating a; b as described below.

2. The simple assignment

   x := a

   Evaluation consists of evaluating a and storing the result in x. The value produced is that stored in x.

3. The sequence

   a; b

   Evaluation changes the state by evaluating a and b in succession. It yields the value of b as its value.

4. The conditional

   if c **then** e1 **else** e2 **fi**

   First c is evaluated. If it yields the value **true** e1 is evaluated. Otherwise e2 is evaluated. The value produced is that of either e1 or e2.

5. The loop

while c **do** e **od**

If c evaluates to **true**, e is evaluated and the process is repeated until c becomes **false**. The loop always yields the value **false**, the last value of c.

Procedures will be discussed in a later section.

Notation and Other Conventions

It is clear that a Hoare style formalism does not suffice for reasoning about languages such as this. While several problems arise, we shall directly address only the following one.

It is no longer clear what an assertion in which arbitrary programming language expressions appear means. For example is

(1) (x := 3; **true**) **and** x = 3

always true?

This problem has been solved in the past by restricting assertions to contain only side-effect free formulas. We take a different approach. The primitive underlying our formalism can be viewed as clarifying the meaning of formulas like the one above by making the range of state changes explicit. The resulting logic is sufficient to define a construct similar to Dijkstra's weakest preconditions [Dij 76]. Thus nothing else is needed. In this sense the logic is similar to that of [Con 77].

We assume that we have a base logic BL which is adequate for reasoning about data domains over which program variables may range.

A term in LES has the form

< programming_language_expression >

Intuitively, such a term represents the value of the expression within the brackets when executed starting in the initial state. An LES expression is built by combining terms and BL constants using the connectives in BL, e.g. +, *, **not**, and $\forall$. If several terms appear in an expression, side-effects produced by one of them never affect the value of the others. Thus, the side-effects produced by an expression in an assertion are confined to within that pair of brackets. As an illustration, if we rewrite (1) as

<x := 3; **true**> **and** <x> = 3

then it is equivalent to <x> = 3, and not to **true**.

We refer to an LES expression as a formula if it yields a boolean value.

Any quantifiers appearing in an expression are interpreted as quantification over the initial value of the variable in question. For example, the formula

$\exists$x (<x := 3> + <x>) = 7

should be read as: There exists an initial value of the variable x such that the value produced by x := 3 (i.e. 3) plus that initial value of x is 7. There is no distinction between logical and program variables.

We will make the usual assumption that free variables are implicitly universally quantified.

In contrast to the usual convention, we do not assume that the operations provided by the logic are the same as those provided by the programming language. This requires a slightly larger set of axioms. On the other hand, if the two sets of operations are actually the same, the additional axioms are all trivial. Furthermore, it may make it possible to give proof rulês for a complicated expression language by simply defining any strange programming language operators in terms of standard mathematical operations.

The following definitions will help to establish the relationship between LES and more conventional programming logics.

[<>E def]:

We let <S> E be the expression E with each terms <t> appearing in E replaced by <S; t>. Thus

<x := 3> (<x> - <z>)

is equivalent to

<x := 3; x> - <x := 3; z>

Informally we think of <S> E as the value of E after executing S. Observe that if E is a boolean expression <S> E in this notation is very similar to the same construct in dynamic logic (see e.g. [Har 79]), S; E in [Con 77], or wp(S, E) in Dijkstra's notation [Dij 76]. The rest of this paper will rely critically on this definition.

[{} def]:

{P} S {Q} will be used occasionally to represent the formula P ==> <S> Q. Again this corresponds closely to the standard use of the notation.

[<- def]:

The notation E[x <- e] denotes the expression E with free occurrences of identifier x replaced by expression e. (This may produce a syntactically incorrect expression if x appears on the left side of an assignment. The actual proof rules presented below carefully avoid this.)

[1 var def]

The term <x> where x is a variable will be frequently written as just x. Thus if we write x = 3 as an LES formula we mean <x> = 3.

We assume that we are given some standard axiomatization of the the underlying logic BL. This axiomatization must deal with '=' in a sufficiently general way to allow substituting equal expressions for equal expressions. In particular this requires the addition of the inference rule

[<>E equ rule]:

e1 = e2
⊢
<e> e1 = <e> e2

(Note that from e1 = e2 we may not conclude <e1> e = <e2> e.) Some further assumptions about BL will be made in conjunction with the inference rule for loops.

Consequence and Composition Theorems

Two rules of inference usually present in a programming logic follow immediately from the predicate calculus axioms and [<>E equ rule]. These are Hoare's rules of consequence and composition.

[consequ thm]:

{P} S {Q}, Q ==> R
⊢
{P} S {R}

We can rewrite the assumptions as

P ==> <S> Q, and
true = (Q ==> R)

respectively. Since by [<>E def]

true = (<S> **true**)

we get by [<>E equ rule]

true = (<S> (Q ==> R)).

But this is equivalent to

<S> (Q ==> R)

which is, again by [<>E def], the same as

(<S> Q) ==> (<S> R)

Combining the above with the first hypothesis yields

P ==> (<S> R)

or

{P} S {R}

which was the desired conclusion.

This leaves us in a position from which we can easily derive a version of Hoare's statement composition rule as well:

[comp thm]:

P ==> <S1> Q, Q ==> <S2> T
⊢
P ==> <S1; S2> T

This follows immediately if we let R be <S2> T in the previous theorem.

The following sections will present axioms and inference rules for various programming language constructs. These will all be generalizations of Hoare's rules (modulo the treatment of termination).

Simple Expressions

We start with an axiom defining the effect on the state of evaluating an identifier:

[id ef ax]:

$$\langle x \rangle\ E \;=\; E$$

where x is a variable or constant and E is any LES expression. This axiom expresses the fact that evaluation of an identifier does not modify the state of the computation, and thus the value of a subsequent expression is not affected by it.

It follows from [<>E def] that the following formulation of the axiom is valid as well:

$$\langle x;\ e \rangle \;=\; \langle e \rangle$$

It is actually equivalent. The equality E = <x> E can be derived by first deriving equalities of the corresponding terms in E and <x> E using the above axiom, and then substituting equals for equals. The first notation is more convenient to use, and thus will be used in stating the remaining axioms.

In general axioms for LES will be presented in pairs. The first one, like the preceding one, will describe the effect that a construct has on the state. The second one will describe the value yielded by the construct.

The preceding effect axiom has a corresponding value axiom only when x is a constant. If there is an equivalent constant in the logic this is simply:

[const val ax]:

$$\langle c \rangle = c$$

We also need proof rules to describe both side effects of, and the value yielded by various operators in the language. In general the rules involved will depend on the semantics of the operator. For the sake of illustration we will consider the case of a side-effect free binary operation op. We will also assume that there is a corresponding operator of the same name in the logic.

Recall that our language uses left to right argument evaluation. The cumulative side-effects of evaluating e1 op e2 are thus identical to those of evaluating e1; e2. We describe this formally by the effect axiom:

[op ef ax]:

```
<e1 op e2> E  =  <e1; e2> E
```

Again, we are defining the side-effects of an expression by stating how the value of subsequent expressions is affected. Note that it now follows from the above definitions that for formulas P and Q:

```
{P} e1 op e2 {Q}    =    {P} e1; e2 {Q}
```

All we need now is a rule describing the value computed by e1 op e2. We might want an axiom of the form:

```
                ?
<e1 op e2>  =  <e1> op <e2>
```

The above however does not hold if the value of e2 is changed by previously executing e1. Thus we rewrite the value axiom as:

[op val ax]:

```
<e1 op e2>  =  <e1> op <e1; e2>
```

As an illustration consider the expression

```
<x * y + z>
```

Using the two preceding axioms we have

```
<x * y + z>  =  <x * y> + <x * y; z>
             =  <x * y> + <x; y; z>
             =  <x> * <x; y> + <x; y; z>
```

Applying [id ef ax] a few times gives us

```
<x * y + z>  =  <x> * <y> + <z>
             =  x * y + z      (by [1 var def])
```

In general if all operators in a programming language expression e behave like op we can (and will) neglect to distinguish between <e> and e.

Rules for Assignment and Conditional

We can express Hoare's assignment axiom as:

```
            ?
P[x <- e] = <x := e> P
```

This unfortunately is no longer correct under our assumptions because evaluation of e may have side-effects.

Let y have the value of e before execution of x := e. Then we want P[x <- y] to be true **after** e is evaluated but before the assignment takes place. If we now generalize from the formula P to an arbitrary LES expression E we get

```
[:= ef ax]:
        y = <e>    ==>    (<x := e> E) = (<e> E[x <- y])
```

Here y is a fresh variable, i.e. y does not occur in e or E. We add the trivial value axiom

```
[:= val ax]:
                      <x := e>    =    <e>
```

As a simple application of the effect axiom we again consider the formula

```
<x := 3> (x - z)
```

Applying the above axiom we get

```
y = <3>    ==>    <x := 3> (x - z) = <3> (y - z)
```

or, using predicate calculus rules,

```
<x := 3> (x - z)   =   <3> (<3> - z)
                   =   3 - z
```

A more easily usable version of [:= ef ax] and a more interesting application are given in Appendix (a).

The proof rules for the conditional are obtained essentially by translating the well-known one into this formalism. In particular if we abbreviate

if b **then** e1 **else** e2 **fi**

as IF, we have

[if ef ax]:

<b> ==> (<IF> E = <b; e1> E), and

not <b> ==> (<IF> E = <b; e2> E)

[if val ax]:

<b> ==> <IF> = <b; e1>, and

not <b> ==> <IF> = <b; e2>

The Loop Rule

We could derive a **while** rule based on Hoare's. The result however would be more complex than necessary. The problem is that the standard statement of the rule involves a hypothesis of the form

P ==> <S> P

which states that the invariant is preserved when the body of the loop is executed. However, one of the things one might wish to prove in our present logic is that the value of some, say integer, expression E is unaffected by the execution of the loop. That is,

<**while ... od**> E = E

This requires some notion of a non-boolean invariant. The obvious solution is to generalize the above hypothesis to arbitrary, not necessarily boolean, P. This however leaves us with the question of what to do with the '==>' operator in the hypothesis, whose arguments must be boolean values.

There are several possible solutions. The one we chose involves generalizing the implication to an arbitrary partial order $\subset$. This makes our 'invariants' sufficiently general that we no longer need a separate notion of a variant function to show termination. Instead we just insist that $\subset$ is well founded and that what used to be our 'invariant' is in fact strictly decreasing with each loop iteration. Thus we obtain the following, at first glance strange, formulation. If c has no side effects and we use WH to denote

while c **do** e **od**

then the following rule holds:

$\subset$ is a well-founded partial order,
<c> **and** v $\subset$ k ==> (<e> v) $\subset$ v
$\vdash$
v $\subset$ k ==> (<WH> v) $\subseteq$ v **and**
<WH> (**not** <c>)

Here k is a constant and v an expression in the logic. Note that we have assumed a fairly powerful base logic. In particular it is essential that we be able to express the first hypothesis formally, and to prove it for some interesting relations $\subset$.

The rule can be read as: If the initial value of v is less than k, and execution of the loop body strictly decreases v, then the final value of v is no larger than its original value, and after execution the condition c evaluates to false.

We could have arrived at this rule using the opposite approach as well. We could have started with a conventional formulation of the rule using both an invariant, and a variant function whose range was some well founded partial order. We would then have transformed this into the above rule by observing, as we will below, that the part of the rule dealing with invariants is redundant. Thus we will refer to v as the variant expression.

The first derivation has the advantage that its ideas will turn out to be of some use later.

At this point the reader may still have trouble believing that the above rule is sufficiently general. Actually it is at least as general as the invariant formulation, again provided the base logic is sufficiently expressive.

To show this, assume we have a Dijkstra style proof of the while loop. This means we have shown that e preserves some invariant assertion I, and decreases the value of the integer-valued variant expression f. This can be transformed into a proof using the above rule as follows: Let v be defined to be f whenever I holds, and ∞ otherwise. Let $\subset$ be the $<$ relation on the natural numbers extended in the obvious way to ∞. Let k be ∞. The first hypothesis is thus trivially satisfied.

Whenever $v \subset k$ the invariant I holds by the construction of v. Thus in this case $v = f$, and therefore v is decreased by e. So the second hypothesis holds as well. We conclude that after the while loop c is false, and the final value of v is less than the initial one, and therefore finite. The latter is equivalent to saying that the invariant still holds.

As another example of an application of the above rule consider the following. Assume again that we have a variant expression f. Also assume that e (and c) leaves expression E invariant. We want to show that:

$$\langle \textbf{while } c \textbf{ do } e \textbf{ od} \rangle E \quad = \quad E$$

In this case we let v be the pair (f, E). We define $(a, b) \subset (c, d)$ iff $a < c$ and $b = d$. Thus values of v with different E components are incomparable.

We can now apply the **while**-rule in a straightforward manner. (K can be (k', E) for any k' larger than the initial value of f. If necessary we can again introduce a constant ∞.) Our desired conclusion is equivalent to the

$$\langle \textbf{while}\ c\ \textbf{do}\ e\ \textbf{od}\rangle\ v \quad \subseteq \quad v$$

conclusion of the proof rule.

Side-effects in the condition add some complexity. The easiest way to deal with them is to consider a new programming language construct

$$\textbf{while'}\ c\ \textbf{do}\ e\ \textbf{od} \qquad \text{(abbreviated as WH')}$$

It is executed like a **while**-loop, except that after the execution is completed we back up to the state immediately preceding the last execution of c. This is somewhat simpler to analyze than the original **while** construct since its execution is equivalent to an integral number of executions of c; e. Thus it can be axiomatized as before simply by requiring that c; e, rather than just e, decreases the variant v:

[WH' ef rule]:

$$\begin{array}{l} \subset \text{ is a well-founded partial order} \\ \langle c\rangle\ \textbf{and}\ v \subset k \implies (\langle c;\ e\rangle\ v) \subset v, \\ \vdash \\ v \subset k \implies (\langle WH'\rangle\ v) \subseteq v\ \textbf{and} \\ \qquad\qquad \langle WH'\rangle\ (\textbf{not}\ \langle c\rangle) \end{array}$$

We can now define the effect of the real **while**-loop in terms of the **while'** construct by adding the following conclusion to the above rule:

[WH ef rule]:

$$\langle WH\rangle\ E \quad = \quad \langle WH';\ c\rangle\ E$$

(It can't be added as an independent axiom since, as we will see in the next section, the loop must terminate for it to hold.)

As usual, we conclude by giving a value axiom:

[WH val ax]:

$$\langle \textbf{while}\ c\ \textbf{do}\ e\ \textbf{od}\rangle = \textbf{false}$$

The Question of Termination

Up to this point we have carefully avoided saying anything specific about the treatment of diverging computations in the logic. In the preceding section we chose a total correctness formulation of the **while**-rule. Thus the reader may have assumed that we were dealing with total rather than partial correctness.

We in fact use something between the two. We can view our treatment of termination as an extension of that normally used for applicative languages. There a nonterminating expression is treated as yielding a special undefined value $\perp$. Here we also have to worry about the affect on the state of such an expression. Thus we assume that every component of the state is assigned such an undefined value as well. Note that this is distinctly different from either a partial or a total correctness logic. In particular, for a nonterminating loop S we do not have

{P} S {**false**}

but we do have

{P} S {**true**}.

Furthermore

{**true**} S; x := 2 {x = 2}

is a theorem.

Rather than introducing a new undefined element $\perp$ into all data domains described by the logic, we treat such a result as being simply some existing value in the domain. Our logic however will not allow us to conclude anything specific about its identity.

This has two technical advantages over the conventional treatment of $\perp$. First, it does not require us to modify the axiomatizations of the underlying data domains by the introduction of this extra element. Second, we can sidestep some sticky issues related to the treatment of $\perp$ in the logic itself. (E.g. what is $\perp$ **and true**?)

We make one further simplifying assumption. The only construct in our language that can possibly fail to terminate is the **while** loop. Since it doesn't yield an interesting value anyway, we assume that it in fact yields the value **false** whether it terminates or not. That is, we assume that when we write

while c **do** e **od**

we in fact mean

while c **do** e **od; false**

This simplifies primarily the relative completeness proof in Appendix (b).

This whole situation perhaps deserves a short explanation. Consider the task of transforming this system into a total correctness logic. Let S be a non-terminating loop as before. This means that {**true**} S {**true**} can no longer be a valid theorem. On the other hand this is by definition equivalent to **true** ==> <S> **true** or just **true** ==> **true.** Thus we would be led to a strange logic indeed.

The attempt to build a partial correctness logic fares no better. An argument almost identical to the one above shows that **true** ==> **false** now has to be a theorem in our logic!

So where does this leave us? Actually things are not bad at all. After all the final goal in all this is to produce a totally correct program that satisfies some given specification, usually of the form:

{P} S {Q}

We accomplish this as follows: First we write an appropriate program and prove it correct using the above logic. We then see whether our proof includes termination proofs for all **while**-loops. If so, our program is totally correct and we're done. If not we simply replace the other **while**-loops with assignments of $\perp$ to all state variables followed by the constant **false.** Here $\perp$ can denote any element of the appropriate data domain. We then have a totally correct program, since the proof in effect showed that the state after execution of such a loop didn't matter.

We have even gained some amusing properties in the process. We have noted that our <S> Q corresponds very closely to Dijkstra's wp(S, Q) or wlp(S, Q). It however enjoys the additional property that

<S> (**not** P) <==> **not** (<S> P)

(The correponding property for **or** also holds. This however should be no great surprise to anyone. It normally does not hold due to nondeterminism in the programming language. Allowing nondeterminism in this context would only add unnecessary complications. In particular the reflexive law of equality would no longer hold.)

Procedures

Up to this point all side-effects in expressions were introduced by explicit assignments. The reader is however hopefully convinced that our framework suffices for axiomatizing other operations with side-effects such as the += (or +:=) operator of C or Algol 68.

In most of these cases it is still not too difficult to transform a program in one of these languages to one which uses only side-effect free expressions, and then to reason about the result. Thus dealing directly with side-effects is perhaps not completely essential until we allow user-defined procedures or functions. The purpose of this section is to illustrate that these can be incorporated into the logic without a great deal of difficulty. We will not, however, do this by presenting one definitive rule, which would lead us too far astray from the topic at hand.

Instead we will present only a very simple Hoare-style rule and then show how to adapt it to our formalism.

We consider the procedure declaration:

$$\begin{array}{c}\textbf{proc } f(\bar{x});\\ B;\\ \textbf{end proc}\end{array}$$

Here B is the body of the procedure and $\bar{x}$ is a sequence of call-by-reference parameters.

We start by making some simplifying assumptions. First, we assume that we are not interested in proving the correctness of recursive procedures. (Our rule will turn out to be of little help in this case.) Secondly we restrict procedure calls as follows:

1. Arguments are simple variables passed by reference.
2. No aliasing occurs between two arguments or between an argument and a global variable.

Consider two predicates $P(\bar{z})$ and $Q(\bar{z})$, where $\bar{z}$ denotes some sequence of variables, and neither P nor Q has any of the parameters or arguments free in it. We could write down the following proof rule for f:

$$\begin{array}{l}\{P(\bar{x})\}\ B\ \{Q(\bar{x})\}\\ \vdash\\ \{P(\bar{a})\}\ f(\bar{a})\ \{Q(\bar{a})\}\end{array}$$

We could just introduce this rule into our system. This may however be more restrictive than desired. We would have no hope of proving something of the form

$$E1 \quad = \quad <f(\bar{a})> E2$$

This problem can be solved as follows. First, translate the above rule into our original notation:

$$P(\bar{x}) \quad ==> \quad <B> Q(\bar{x})$$
$$\vdash$$
$$P(\bar{a}) \quad ==> \quad <f(\bar{a})> Q(\bar{a})$$

We now want to generalize so that P and Q can be logical expressions of arbitrary type. However we have the same difficulty as with the **while**-rule. Clearly, we have to replace the implications by something else.

Our solution in the case of loops was to use arbitrary well founded orderings. This happened to be convenient since it subsumed the termination condition as well.

Presently however we have no additional constraints. Thus we can replace the ==> here by an arbitrary binary predicate R of the base logic rather than a restricted type of ordering. We can now write our modified rule as

$$R(\ e1(\bar{x}),\ <B>\ e2(\bar{x})\ )$$
$$\vdash$$
$$R(\ e1(\bar{a}),\ <f(\bar{a})>\ e2(\bar{a})\ )$$

Up to this point we have neglected the fact that our procedure yields a value. We use a value rule that looks very similar to the effect rule:

$$R(\ e1(\bar{x}),\ <B>\ )$$
$$\vdash$$
$$R(\ e1(\bar{a}),\ <f(\bar{a})>\ )$$

The ambitious reader should now be in a position to carry over some more general procedure rules, such as those given in [Gri 80] and [Car 78].

Conclusions

The logic LES is a simple formalism for reasoning about programs written in so-called expression languages. Although it is somewhat inconvenient if proofs are carried out in full detail, it seems to interact nicely with informal reasoning about such programs. In particular we have defined the meaning of Hoare style assertions in terms of LES and all our proof rules are generalizations of Hoare's. Thus if a section of the program is amenable to proof using Hoare's system we can easily translate such a proof into an LES proof. It is still possible to present proof outlines in the form of a program annotated with assertions, although this becomes less and less informative as one's programming style becomes more and more applicative.

It may be useful to develop some notational convention for inserting pre- and postconditions for arbitrary sub-expressions (rather than just at a ';'). We may also want to establish a syntactic shorthand for referring to the value yielded by the last expression. It would then be possible to derive rules similar to those in [Kow 77].

Our approach can be viewed as a combination of the following two ideas. We distinguish strictly between programming language and logic operations. This may be useful in general, and is virtually unavoidable in this context since it is not clear what the meaning of side-effects in logical formulae is.

Programming logics such as Dynamic Logic or Constable's Programming Logic allow one to talk about the truth or falsity of a predicate after execution of some sequence of statements. The present logic generalizes this to the value of an arbitrary expression after a number of state-changing operations.

This generalization is also useful even if expressions do not have side-effects. For example consider the statement

x := 2 * x

To express the fact that the final value of x is twice its original value in Hoare's notation would require an auxiliary variable. In the present system it can be expressed as

<x := 2 * x; x> = 2 * x

Acknowledgements

I would like to thank David Gries, Chris Buckley, and especially Alan Demers for their suggestions on earlier drafts of this paper.

References

[Car 78] Cartwright, Robert and Derek Oppen, "The Logic of Aliasing", Department of Computer Science, Cornell University, Technical Report TR78-358, 1978.

[Con 77] Constable, Robert L., "On the Theory of Programming Logics", Proceedings of the Ninth Annual ACM Symposium on Theory of Computing, ACM, May 1977.

[Coo 76] Cook, Stephen A., "Soundness and Completeness of an Axiom System for Program Verification", Department of Computer Science, University of Toronto, Technical Report No. 95, June 1976.

[Dij 76] Dijkstra, Edsger W., A Discipline of Programming, Prentice-Hall, 1976.

[Gri 80] Gries, David and Gary Levin, "Assignment and Procedure Call Proof Rules", TOPLAS 2, Oct 1980.

[Har 79] Harel, D., First-Order Dynamic Logic, Springer Verlag, 1979.

[Hoa 69] Hoare, C.A.R, "An Axiomatic Basis for Computer Programming", CACM 12, Oct. 1969, pp.576-580.

[Kow 77] Kowaltowski, Tomasz, "Axiomatic Approach to Side Effects and General Jumps", Acta Informatica 7, pp. 357-360, 1977.

[Pri 77] Pritchard, Paul, "Program Proving - Expression Languages", Information Processing 77, North Holland, 1977, pp. 727-731.

[Schw 78] Schwartz, Richard L., "An Axiomatic Treatment of Asynchromous Processes in ALGOL 68", preliminary draft. More detail can be found in:

Schwartz, Richard L. "An Axiomatic Semantic Definition of ALGOL 68", Computer Science Department, UCLA - 34P214-75, University of California, Los Angeles, July 1978.

Appendix A. An Example

We give a proof of a small program which includes an expression with side-effects. It illustrates the interaction of LES with Hoare style reasoning.

Consider the program P:

```
a := b;
r := ( x := (a := a + 1) + a;
       2 * x
     )
```

Assume we wish to prove

$$\{\mathbf{true}\}\ P\ \{r = 4 * b + 4\}.$$

We begin by deriving a more easily applicable, though somewhat restricted form of [:= ef ax]. Recall that it was originally stated as:

```
<y> = <e>  ==>  (<x := e> E) = (<e> E[x<-y])
```

We consider the case in which E contains no bracketed programming language expressions other than simple variables. If we expand (<e> E[x<-y]) according to [<>E def], we find that y only occurs in terms corresponding to the term <x> in E. These have the form

```
<e; y>
```

Since y is a fresh variable which does not occur in e, each such term is equivalent to <y>, which is equivalent to <e>. Thus (<e> E[x<-y]) can be rewritten as E', where E' is obtained from E by substituting substituting <e> for <x>, and substituting <e; z> for each other term <z>. We can then rewrite the axiom as

```
[simple := ef ax]:
        <y> = <e>  ==>  (<x := e> E) = E'
```

or, since y no longer appears on the right side of the implication:

```
<x := e> E   =   E'
```

Let E represent the expression (a := a + 1) + a, and M the expression (x := E; 2 * x), i.e. the right hand side of the second main assignment in P.

As usual we first decompose the problem into two pieces by inserting the assertion '{ a = b }' between the two "statements". We thus have to prove the following two theorems:

```
{true} a := b {a = b}
```

and

```
{a = b} r := M {r = 4 * b + 4}
```

The first theorem is trivial. From [simple := ef ax] we get:

```
<a := b> (a = b)   =   (b = b)   =   true
```

The left side of this equality is the theorem we wanted to prove. We could just as well have used Hoare's assignment axiom since it is certainly valid in this context.

It is almost as straightforward to prove the main theorem. Here we do however have to use LES. We first rewrite it as

```
a = b   ==>   <r := M> (r = 4 * b + 4)
```

The basic idea is to rewrite the right side R of this implication by applying [simple := ef ax] once for each assignment. We start with the assignment to r:

```
R  =  (<M> = 4 * b + 4)
   =  (<x := E; 2 * x> = 4 * b + 4)
```

We observed that b does not appear in M and we could thus replace <M; b> by just b. We also made use of the fact that in general

```
<e; c> = c
```

for any expression e and constant c. If c is axiomatized as above this is easy to see as follows:

```
<e; c>  =  <e> <c>    (by [<>E def])
        =  <e> c      (by [<>E equ rule]
        =  c          (by [<>E def])
```

By applying the assignment axiom to x := E we obtain

```
R  =  (<2 * E> = 4 * b + 4)
   =  (<2 * ((a := a + 1) + a)> = 4 * b + 4)
   =  (2 * <(a := a + 1) + a>  =  4 * b + 4)
```

Using [op val ax] and [:= val ax], we rewrite this as

```
R  =  (2 * (a+1 + <a := a + 1; a>) = 4 * b + 4)
```

We then apply [simple := ef ax] one last time to get

$$R = (2 * (a + 1 + a + 1) = 4 * b + 4)$$
$$= (2 * a + 4 = 4 * b + 4)$$

Now

$$a = b \implies R$$

follows trivially and we are done.

Appendix B. A Relative Completeness Proof

What follows is a proof that the preceding axiomatization of LES without procedures is relatively complete, roughly in the spirit of [Coo 76] or [Har 79].

We begin by making the following assumptions:

1. All primitive operations and constants in the programming language, other than assignment, **if**, and **while**, are axiomatized using axioms of the form given in the "simple expressions" section. It should become obvious that this is necessary for explanatory purposes only.

2. We augment the axiomatization of LES by the following rule, which allows us to prove nontermination for **while** loops:

```
[∞ WH ef rule]:
        P ==> <c; e> P
        P ==> <c>
        ⊢
        P  ==>  <while c do e od; x> = ⊥
```

   This rule is generally useless since it serves only to prove programs incorrect. On the other hand, it makes it much easier to state the relative completeness theorem.

3. We assume we have a fairly powerful base logic BL. ZF set theory is a reasonable choice. In particular we need:

   a) BL includes a conventional axiomatization of the predicate calculus with equality.

   b) BL is expressive for LES. More precisely, for any formula P in LES there is an equivalent formula Q in BL. It would be sufficient to assume that for any programming language expression e there is a BL expression e' such that <e> = e'.

c) The state transition function associated with any programming language expression e is definable in the base logic BL. That is, for any programming language expression e and any sequence of n variables $\bar{x}$ there is a formula $P_{e,\bar{x}}$ in BL such that P(R) is true iff R is the relation defined by: $\bar{a}R\bar{b}$ iff whenever the initial values of the $\bar{x}$ are $\bar{b}$ then their final values after executing e are $\bar{a}$.

d) N-tuples are definable if their components are.

e) A relation is definable whenever we can write down the corresponding binary predicate and its domain is definable.

f) The reflexive-transitive closure R^* of a relation R is definable.

g) The restriction of a relation to some subset of its domain is definable. Such a subset will be specified by a unary predicate.

h) Well-foundedness of a relation is expressible.

4. For purposes of simplicity we assume that any formula in LES can be written in prenex normal form. This should be the case anyway unless BL is a constructive logic.

5. We assume that the axiomatization of LES is extended with a complete axiomatization of BL, e. g. by adding all true statements of BL as axioms. The resulting axiomatization is unlikely to be recursively enumerable, but as in [Coo 76] that will not concern us here. Here we identify a formula in LES which has angle brackets only around simple variables and constants with the corresponding formula in BL which has all brackets deleted.

Of course we have no hope of proving LES complete without assumption (5), since under reasonable assumptions BL will never be complete. Thus introducing this assumption essentially reduces our statement to the weaker claim that the incompleteness of LES can, in some some sense, be traced back exclusively to the incompleteness of BL.

We now claim that under these assumptions any true LES formula is provable using the above axiomatization.

If we had proceeded completely formally we would have given a definition of a true LES formula before stating the theorem. We skipped this process by assuming that the reader has a sufficient informal understanding of our toy language without a formal execution model. It should however be repeated that our model of a nonterminating loop is that it assigns some element $\perp$ to each component of the state. A true LES formula is one that holds for all possible values of $\perp$.

BL does not have to deal with $\perp$ explicitly. We can still think of BL as a restriction of LES as we did in assumption (5). If the LES formula mentions $\perp$ we simply precede the corresponding BL formula by

$$\forall\perp \ldots$$

The proof of our theorem relies on purely syntactic principles. We will first define the syntactic complexity of a programming language expression to be some natural number. We then show that proving any LES formula can be reduced to proving a number of LES formulae in which only simpler programming language expressions occur.

Assume without loss of generality that we are only dealing with formulas in pure LES, that is formulas in which constructs of the form <e> E have been eliminated using [<>E def].

We will show that given a true formula P in the above form which has n occurrences of simple terms <e> such that e has complexity m and no simple terms of complexity greater then m, we can give a proof for P, assuming that this hypothesis holds for formulas with at most n-1 occurrences of simple terms of complexity m. In most cases this will be done by showing that P is provably equivalent to a predicate P' in which one of the simple terms of highest complexity has been replaced by some number of terms of lower complexity.

We define the complexity of a programming language expression to be 2n + m where n is the number of operators appearing in the expression and m is the number of semicolons. Here the **if** and **while'** constructs are counted as operators. The complexity of

while c **do** e **od**

is defined to be that of

while' c **do** e **od;** c

At this point we consider the most complex simple term <e> (or one such term if there are several) appearing in the formula P which we wish to prove. In general e will have the form

$$e_1; \ldots ; e_n$$

We will eliminate <e> by applying the inference rule or axiom associated with the outermost operator in e_1. If n = 1 we apply the rule giving the value of the operator in question. Otherwise we use the one defining its side-effects.

The n = 1 case is almost trivial. We will therefore dispense with it first.

If e consists of just a variable or constant then P is in BL and therefore provable. Thus we assume e contains an operator. We will consider various possible outermost operators separately.

Simple operation -

<e> has the form

<e' op e''>

[op val ax] allows us to show that this is equal to

<e'> op <e'; e''>

Thus we can prove that P is equal to P' which is obtained by substituting the above for <e>. Note that our definition of complexity was cleverly contrived so that <e'; e''> is less complex than <e>. Thus our induction hypothesis allows us to assume that P' is provable. Thus P is provable.

Assignment -

<e> has the form

<x := e'>

Here we just substitute <e'> for <e>. [:= val ax] allows us to prove that the result is equivalent to the original formula.

Conditional -

<e> has the form

<**if** b **then** e' **else** e'' **fi**>

We assume that P has no explicit quantifiers (but possibly some number of free variables). If this should not be the case, by assumption 4 we can write P in prenex normal form and apply the following reduction to the matrix of P instead.

[if val ax] lets us prove that <e> is equivalent to <b; e1> if <b> holds, and <b; e2> otherwise. Thus if we let P' be P with the conditional replaced by <b; e'> and P'' be P with the conditional replaced by <b; e''> then P is provably equivalent to

(<b> ==> P') **and** (**not** <b> ==> P'')

Again the rewritten formula no longer has occurences of <e> and the newly introduced simple terms have complexity less than e.

Loop -

By [WH val ax] these can always be rewritten as **false.**

Now we can return to the case in which e is a sequence of semicolon separated expressions. Again we consider several different cases, depending on the outermost operator of the first expression e_1.

None -

e_1 may just be a constant or variable. In this case [id ef ax] allows us to replace

$$<e_1; \ldots ; e_n>$$

by

$$<e_2; \ldots ; e_n>$$

Simple operation -

e_1 may have the form

$$e1 \text{ op } e2$$

We can rewrite $<e>$ as

$$<e1; e2; e_2; e_3; \ldots ; e_n>$$

using [op ef ax].

Assignment -

Assume e_1 has the form

$$x := e'$$

We again assume wolog that P has no explicit quantifiers. Let P' be P with each occurrence of $<e>$ replaced by

$$<e'; e_2[x \text{ <- } y]; \ldots ; e_n[x \text{ <- } y]>$$

where y is a new variable. We can then replace P by the statement Q:

$$y = <e'> \implies P'$$

Q is a true statement which is less complex than P and is thus provable. From [:= ef ax] we immediately have

$$\begin{aligned} &y = <e'> \implies \\ &\quad <e> = <e'; e_2[x \text{ <- } y]; \ldots ; e_n[x \text{ <- } y]> \end{aligned}$$

Thus we get

$$y = <e'> \implies (P' = P)$$

From Q it then follows that

$$y = <e'> \implies P$$

If we now substitute $<e'>$ for y we get a proof of P.

Conditional -

This is almost identical to the case of the conditional by itself.

Loop -

$<e_1>$ may be of the form

while c **do** e **od**

This is actually the only interesting case in the whole argument. Again assuming wolog that P has no explicit quantifiers, we rewrite P as

Q **or** Q'

Q is true whenever P is true and the loop e_1 does not terminate. Q' is true whenever P is true and the loop terminates.

We will show that both Q ==> P and Q' ==> P are provable. Furthermore both Q and Q' have the instance of the **while** loop replaced by simple terms of lesser complexity. Since their disjunct is thus less complex than P it will be provable by the induction hypothesis. Thus P will be provable.

We first rewrite P as P' where each term beginning with the loop e_1 has been replaced by

$$<e_1> E_i$$

where each E_i is in BL. (Such E always exist by assumption (3b).) P' is provably equivalent to P by our induction hypothesis since the formulas

$$E_i = < \ldots >$$

are less complex than P. Expanding P' using [<>E def] we get a formula P'' in which e_1 occurs only in terms of the form

$$(*) \qquad <e_1; x_i>$$

where x_i is a simple variable.

We turn our attention to the formula Q. We first need to express the fact that the loop does not terminate. The BL formula N equivalent to the following LES formula will do:

<**while** c **do** e **od**> (y = ⊥)

Here y is a fresh variable.

Since we know that in the case under consideration each term of the form (*) is equivalent to ⊥, we can now write Q as

$$N \text{ and } P_\perp$$

where $P_\perp$ is P'' with each instance of a formula (*) replaced by $\perp$.

We can prove that Q ==> P as follows. Certainly N is an invariant of the nonterminating loop. Since both

$$N \Longrightarrow \langle c;\ e\rangle N, \text{ and}$$
$$N \Longrightarrow \langle c\rangle$$

are less complex than P they are provable by induction hypothesis. Thus by [∞ WH ef rule] we can show that each term (*) is equivalent to $\perp$ (provided N holds initially). Therefore

$$N \Longrightarrow (P_\perp = P)$$

is provable. The conclusion follows immediately.

We now consider Q'. We again want to chose it in such a way that from Q', [WH' ef rule], and [WH ef rule] we can deduce P.

We start by rewriting <e> as

$$\langle \textbf{while}'\ c\ \textbf{do}\ e\ \textbf{od};\ c;\ e_2;\ \ldots\ ;\ e_n\rangle$$

Thus from now on we will deal only with **while'** loops. We then define P'' as before. Thus P'' has no occurrences of e_1 and the only occurrences of the corresponding **while'** loop WH' are in terms of the form

$$\langle \textbf{while}'\ c\ \textbf{do}\ e\ \textbf{od};\ x_i\rangle$$

In order to apply [WH' ef rule] we need to pick an ordering $\subset$, a constant k, and a variant function v. We chose v to be the whole state of the computation. That is, we let v be

$$(x_1,\ \ldots\ ,\ x_j)$$

The idea then is to let $\subset$ be something like the transitive closure of the transition function associated with one iteration of the loop.

Unfortunately this relation is not in general well-founded. We want to restrict it to those states s such that when the loop is started in state s it terminates. We express this condition by the following predicate T(s):

$$(v = s) \Longrightarrow \textbf{not}\ N$$

Thuse we let $\subset$ be the reflexive transitive closure of the following relation R on

$$\{ (a_1, \ldots, a_j) \} \cup \{ \infty \}$$

Let aRb hold iff T(a) holds and either b = ∞ or

$$(v = b) \text{ ==> } \mathbf{not} <c> \mathbf{and} (<c; e> v \;=\; a)$$

As expected we let k = ∞. The statement s $\subseteq$ k is equivalent to saying that s leads to termination. a $\subseteq$ b where both a and b are j-tuples defining a state is equivalent to the statement that both a and b lead to termination and that if the **while'**-loop is started in state b then after some number of iterations of the loop state a will be reached. Thus $\subset$ is clearly well-founded. Furthermore this statement is expressible in BL and therefore provable. By construction the other hypothesis of the [WH' ef rule] is also satisfied. Since it is less complex than P it is provable by the induction hypothesis.

We define a formula R to be P'' with each instance of a term

$$<WH'; x_i>$$

replaced by $<x_i'>$. Thus intuitively we can think of x_i' as being the value of x_i after execution of the loop.

We let Q' be

$$v \subset k \;\mathbf{and}$$
$$\forall x_1' \ldots x_j' \; ((v' \subseteq v \;\mathbf{and\ not}\; c') \text{ ==> } R)$$

Here v' and c' represent v and c respectively with each x_i replaced by x_i'. In the case of c we first rewrite it in BL.

It should be observed that

$$v' \subseteq v \;\mathbf{and\ not}\; <c'>$$

Is just a formal way of saying that v' is the final state produced by the **while'**-loop when started in state v. (There is after all just one state which is reachable from v by some number of loop iterations in which c is false.) Thus Q' is true whenever the loop terminates and R holds when all the x_i' are the final values of x_i. It follows from the definition of R that Q' is exactly the formula we promised.

Q' ==> P is easily provable as before. We know that the two hypotheses of [WH' ef rule] hold. Since we are given v $\subset$ k as well, we conclude that

$$(<WH'> v) \subseteq v \;\mathbf{and\ not}\; <WH'> c$$

We again assume that c has been rewritten in BL. We obtain the desired result by substituting <WH'> x_i for each x_i' in Q' and then observing that this transforms R back into P''. Thus P'' holds. It follows from [WH ef rule] that WH'; c can be written as e_1 and thus P and P'' are equivalent.