Side Effects and Aliasing Can Have Simple Axiomatic Descriptions

HANS-JUERGEN BOEHM
University of Washington

We present a different style of axiomatic definition for programming languages. It is oriented toward imperative languages, such as Algol 68, that do not distinguish between statements and expressions. Rather than basing the logic on a notion of pre- or postcondition, we use the value of a programming language expression as the underlying primitive.

A number of language constructs are examined in this framework. We argue that this style of definition gives us a significantly different view of the notion of "easy axiomatizability." Side effects in expressions as well as aliasing between variables are shown to be "easily axiomatizable" in our system.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics*; F.3.1 [**Logics and Meaning of Programs**]: Specifying and Verifying and Reasoning about Programs—*logics of programs, mechanical verification*; F.3.3 [**Logics and Meaning of Programs**]: Studies of Program Constructs

General Terms: Languages, Theory, Verification

Additional Key Words and Phrases: Algol 68, aliasing, completeness, expression language, LISP, side effect

1. INTRODUCTION

Much work has been done in the area of formal definition of programming languages through axioms and inference rules. Such definition not only provides a precise way of specifying semantics, it is also directly usable for formal reasoning about programs written in that language. It can thus serve as a basis for a machine-assisted verification system for the language (cf. [4] and [9]). It may also be used as a foundation for less formal programming methodologies, such as the one presented in [13] and [15].

Unfortunately, it has been difficult to give axiomatic semantics for most nontrivial programming languages. Few "real" programming languages have been so specified, and even these specifications are sufficiently complex or

Most of the material presented here also appears in the author's Ph.D. dissertation [2]. A much earlier version of parts of the material was presented at the 9th ACM Symposium on Principles of Programming Languages [1]. Much of the work leading up to this paper was done at Cornell University, where it was supported in part by NSF grant MCS 01048 and amendment MCS 8104010. Author's current address: Dept. of Computer Science, Rice University, P. O. Box 1892, Houston, TX 77251-1892.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1985 ACM 0164-0925/85/1000-0637 $00.75

ill-understood as to cast doubt on their correctness. (See for example Gries' discussion of the Euclid procedure call rule in [14] and O'Donnell's comments on defined functions and goto statements [23].)

It has been suggested that the difficulty in obtaining axiomatic semantics for conventional languages is an indication of the deficiencies of these languages. A "good" language design should make it easy to both describe the semantics of the language to the programmer and for the programmer to reason about programs written in that language. This simplicity should be reflected in a simple formal definition of the language.

The axiomatizability of a programming language and of programming language features has thus been proposed as a criterion for the language designer. It has been used in support of Dijkstra's language [13], of Euclid [25], and sometimes of Pascal [19]. Pascal and Euclid have usually been characterized with a style of axiomatic system directly derived from Hoare's original work [18].[1] Dijkstra uses a different notation, but his axioms are very similar to Hoare's.

We argue that an evaluation of programming language features based on their axiomatizability needs to be viewed with caution.

There are few formal results on this subject. It is easy to be convinced that any programming language could be formally axiomatized if the aesthetics of axiomatization were not a consideration. It would then be possible to just model an interpreter for the language. Thus, the problem is interesting only because we want an axiomatization that is compact and easy to understand and work with. Needless to say, these criteria are hard to formalize.

The only major negative result in the axiomatizability of programming languages is [7]. It is of real interest only when dealing with formal systems very similar to Hoare's. Conclusions about the "easy axiomatizability" of certain constructs are thus almost exclusively empirical results based on actual attempts to do so. These attempts have almost always been based on Hoare's work.

We outline the development of a programming logic that leads to a significantly different evaluation of a number of different programming language constructs. We hope to convince the reader that this development is no less intuitive than Hoare's. In fact we are able to overcome many of the problems that have traditionally plagued Hoare-style logics. We rely crucially on the fact that the underlying language is an expression language in the sense of Algol 68 (cf. [28]). Expressions are allowed to have arbitrary side effects, and the notions of statement and expression coincide. Aliasing between variables does not introduce any problems. User-defined functions can be accommodated in a straightforward manner.

In many respects the logic we present closely parallels a denotational semantic definition of the language. The difficulties encountered in defining a construct are similar. On the other hand, the logic can be directly applied to reasoning about programs in much the same style as Dijkstra's weakest precondition calculus. We do not explicitly mention stores or environments in the logic. It is not necessary to introduce a separate object to represent nontermination.[2]

---

[1] We have concentrated on imperative languages. Different techniques, in some sense similar to the ones we propose here, are normally used in conjunction with applicative languages (cf. [21]).

[2] We refer the reader to [2] for a thorough discussion of the role of nontermination in the logic. It can be argued fairly easily that neither the conventional total view nor the partial correctness one

We claim that "easy axiomatizability" is largely a matter of a rather arbitrary choice of formalism. We do not question the utility of an axiomatic language definition, but we do want to cast doubt on claims of the form: "Programming language X is bad because it has feature Y, which is hard to axiomatize."

2. SOME PROGRAMMING LANGUAGE FRAGMENTS

We assume that we are dealing with a programming language in which the only syntactic entity is an expression. Expressions always yield a value. They may or may not modify the state of the computation.[3] This is the approach taken in languages such as Algol 68, Lisp, and Russell (see [3]). As stated, this is in complete contrast to the languages commonly used in conjunction with Hoare logic.

Identifiers in our language may be bound to simple values or to variables (locations).[4] Similarly, expressions may yield any of these as results. Thus, we allow expressions like

$$\textbf{if } a < 1 \textbf{ then } x \textbf{ else } y \textbf{ fi} := 3$$

where a is bound to an integer value, and x and y are bound to integer variables.

We explicitly distinguish between variables and their values. This allows us to conveniently talk about aliasing of variables. For present purposes, we assume that programmers will explicitly write $.x$ when they mean the value stored at location x. The name x by itself would denote the location.[5]

We introduce only enough of a programming language so that we can introduce the logic and look at aliasing between variables. The language is taken to be typed, but the details of the type system are not important to this discussion.

It is easy to give an axiomatization of loops within our framework. We neglect to do so here in order to avoid a lengthy discussion of nontermination. We also omit any discussion of functions and recursive declarations. (Appropriate axiomatizations of all three constructs are given in [2]; loops are discussed in a slightly different setting in [1].)

Since all language constructs are expressions, we proceed to informally specify the syntax and semantics of those constructs that we are interested in. We

---

can be accommodated directly by this logic. This is neither surprising nor an indictment of the logic. As O'Donnell [23] points out, it is not even clear how to adhere to either of these views in a conventional programming logic if we wish to accommodate nonterminating function calls. The discussion in [1] and [2] interprets a nonterminating expression as returning some result about which nothing is known. Similar approaches are discussed in [9], [23], and [17]. None of these treatments is very complicated. The system of [2] can serve the same purpose as a total correctness logic by insisting that termination be proved separately (but within the same logical framework).

As an alternative, it would be possible to preserve the explicit notion of nontermination. This would necessitate a definition of "formula" different from the one given here, and would add somewhat to the complexity of the system.

[3] This is a necessary assumption for our development. There are arguments both for and against designing a language in this way. We hope to convince the reader that "axiomatizability" is not such an argument.

[4] In the full language of [2], identifiers may be bound to functions as well.

[5] In practice, the programmer can usually be allowed to omit the ".", since the compiler can typically infer it from type information. In reasoning about a program, we do need to be aware of its existence. This distinction also makes possible a very natural treatment of arrays and other structured objects.

Our notation is borrowed from the Bliss programming language (cf. [29]).

consider the following kinds of expressions:

(1) A simple identifier. It yields the simple value, or the location, bound to the identifier. It does not modify the state.

(2) The expression

$$a + b$$

where a and b are expressions producing integers. (This is used to illustrate the treatment of such side-effect-free operations.) It yields the sum of the values produced by the two subexpressions. Its effect on the state is that of evaluating first a and then b. (Note that left-to-right evaluation is assumed.)

(3) The expression

$$.a$$

where a is an expression producing a variable (location). It yields the simple value stored at that location. Its effect on the state is the same as that of evaluating the subexpression a.

(4) The conditional

$$\textbf{if } c \textbf{ then } a \textbf{ else } b \textbf{ fi}$$

where c yields a Boolean value. If c produces true, then a is evaluated next, and its value is used as the value of the conditional. If c evaluates to false, b is evaluated instead of a.

(5) The assignment

$$a := b$$

where a produces a location and b produces a simple value. This stores the value produced by b at the location produced by a. The assignment as a whole yields the value produced by b. Thus,

$$x := (y := 1)$$

assigns 1 to both x and y.

(6) The sequence

$$a; b$$

where a and b are any subexpressions. The effect of the sequence is that of first evaluating a and then b. The sequence yields the value (or variable) produced by b.

(7) The block

$$\textbf{let } x == a \textbf{ in } b \textbf{ ni}$$

Its effect on the state is that of evaluating a and then evaluating b, with x bound to the value (or variable) produced by a. The block yields the resulting value of b. Note that x is not known inside a.

(8) The expression

New( )

It yields a previously unallocated location. It affects the state by marking this location as allocated.[6]

We can obtain the equivalent of a Pascal variable declaration by explicitly binding an identifier to a new location obtained from the New function. We might write

let x == New( ) **in** . . . **ni**

Formal definitions of these constructs, together with soundness and relative completeness proofs for a variant of the logic given below, can be found in [2].

3. THE LOGIC

There are a number of Hoare-like axiomatizations of expression languages or expressions with side effects (cf. [11, 20, 26, and 27]). They all modify Hoare's original logic by explicitly introducing one or more symbols to denote expression values. (They also generally prevent us from using programmer-defined functions directly in assertions.) There are also several augmented versions of Hoare's logic which allow aliasing between variables (cf. [5, 6]). In adding additional primitives to Hoare logic, these developments all support the impression that side-effect-free expressions, "value-free" statements, and absence of aliasing inherently lead to the simplest axiomatization.

We base our approach on a primitive that allows us to make explicit statements about the value of programming language expressions. Because of the nature of the programming language, we can then define a construct similar to Dijkstra's weakest preconditions ([13] or [15]). Thus our one primitive suffices.

A logical term denotes a simple value, a location, or a function. It is defined inductively as follows:

(1) If x is an identifier, then x is a term. The constants **true** and **false** are terms, as are any other constants we may choose to provide. These constants may be functions that operate on simple values or on other functions. Thus we view any primitive function symbols (e.g., + for integer addition) as a special case of such constants. Primitive predicates are simply primitive functions which produce a Boolean value. Thus they are also included here.

(2) Given terms $t_1, \ldots, t_n$ and a term t_0 which denotes a function, the application $t_0(t_1, \ldots, t_n)$ is a term. We use standard mathematical notation where appropriate (e.g., $+(a, b)$ is written as $a + b$).

(3) If t_1 and t_2 are terms, then $t_1 = t_2$ is a (Boolean) term. Note that we write t_1 iff t_2 as $t_1 = t_2$. We assume that equality is the only operation that may be applied directly to locations.

[6] We assume that it does not initialize the value at that location. This assumption is convenient for present purposes, but by no means necessary.

(4) If t_1 and t_2 are Boolean terms, then so are

$$t_1 \wedge t_2 \qquad t_1 \vee t_2 \qquad \sim t_1$$

$$t_1 \Rightarrow t_2 \qquad \exists x \cdot t_1 \qquad \forall x \cdot t_1$$

We may quantify over locations, as well as over simple values.

(5) If a is any programming language expression, then

$$\langle a \rangle$$

is a term denoting the value of the expression.

For the purpose of the logic, we augment the notion of a programming language expression from the previous section with the expression,

$$\text{IsAllocated}(a)$$

It is a Boolean expression that is true whenever the location produced by a is already allocated in the current state; that is, whenever we can be assured that location a will not be returned by a subsequent call to New.[7]

We take a formula to be simply a Boolean term, that is, a term with Boolean type.[8] We interpret free identifiers in formulas to be universally quantified.

The first four kinds of terms have their usual meaning.

We refer to the last kind of term as a programming language term. Such a term represents the value of the expression within the brackets when executed starting in the current state. Identifiers inside these terms can be bound either by declaration in a **let** block or by a quantifier preceding the term. (Recall that identifiers anywhere refer to the value they are bound to, and never to the value stored in a location the identifier might be bound to.)

If several programming language terms appear in an expression, side effects produced by one of them never affect the others. Thus, the side effects produced by an expression in an assertion are confined to within that pair of brackets. (This convention will be extended later.) As an illustration, if we write

$$\langle x := 3; \textbf{true} \rangle \textbf{ and } \langle .x \rangle = 3$$

then this is equivalent to $\langle .x \rangle = 3$ (and not to **true**).

We say that a formula is valid if and only if it is true in all possible states, that is, for all possible values stored at all locations and for any combination of allocated and unallocated locations.

Unlike Hoare logic, we can maintain a strict separation between logical and programming language operations. We never require that it be possible to substitute programming language expressions into logical ones. In this way we gain some increased generality at the expense of explicitly axiomatizing all operations (even side-effect-free ones) in the programming language. Our axioms

[7] In interpreting the logic, we assume that New never allocates the same location twice; that is, locations are never recycled. In practice, a garbage collection implementation would be acceptable, since its behavior would be indistinguishable from this model.

[8] We use a slightly nonstandard approach for the sake of brevity and simplicity. It is also possible to separate the notion of term and formula in a more conventional manner. This would be necessary if we were to introduce an explicit value representing nontermination.

could, for example, reflect the fact that "+" in a programming language expression does not exactly correspond to the same symbol in a mathematical term. (The programming language version might overflow, produce a rounding error, etc.)

For present purposes, there is no need to take advantage of this. Thus we do assume that the operations and constants provided by the logic are a superset of those provided by the programming language. (For the purposes of this paper, integer addition is the only such operation provided by the programming language.)

The following relates our logic to other programming logics.[9]

We define the extended term $\langle a \rangle t$ to be the term t with each programming language term $\langle b \rangle$ appearing in t syntactically replaced by $\langle a; b \rangle$. (We assume that variables bound in t are first renamed to avoid clashes with a.)

For example, the extended term

$$\langle x := 3 \rangle (\langle .x \rangle - \langle .z \rangle + a)$$

is, by the preceding definition, equivalent to

$$\langle x := 3; .x \rangle - \langle x := 3; .z \rangle + \langle x := 3; a \rangle$$

Informally, we think of $\langle a \rangle t$ as the value of t after executing a. Observe that if t is Boolean, then $\langle a \rangle t$ in this notation is very similar[10] to the same construct in dynamic logic (see, e.g., [16]), to the construct $S; t$ in [8], or to $wp(S, t)$ in Dijkstra's notation [13].

We use this notation frequently. We occasionally use Hoare's notation $\{P\}S\{Q\}$ to represent the formula $P \Rightarrow \langle S \rangle Q$.

Again this corresponds closely to the standard use of the notation. In Hoare's framework, the statement

$$\{\mathbf{true}\}\ x := 1\ \{x = 1\}$$

means that, in the state after the assignment, the assertion "$x = 1$" holds, or equivalently, that the final value of x is 1. In our framework, we know, from the preceding definition, that

$$\{\mathbf{true}\}\ x := 1\ \{\langle .x \rangle = 1\}$$

is equivalent to

$$\mathbf{true} \Rightarrow \langle x := 1 \rangle (\langle .x \rangle = 1)$$

This simplifies to

$$\langle x := 1; .x \rangle = 1$$

which again states that the final value of x is 1.

---

[9] We approximate the primitives fundamental to other logics in ours. It is also possible to express something similar to our primitive in some other logics. [22] uses a logic which introduces a primitive similar to ours, but in a different context. In most cases our notion of a programming language term is expressible in Dynamic Logic. The purpose of this paper is to point out that we get a different perspective by using it as the sole basis of the logic.

[10] The only differences occur in conjunction with nontermination or nondeterminism, neither of which is discussed here.

4. THE AXIOMATIZATION

We do not concern ourselves with axioms dealing purely with the predicate calculus, equality,[11] or with simple values. A simple adaptation of a standard mathematical theory should do for these. We are interested primarily in axiomatizing programming language terms.

An immediate consequence of our definition of the validity of a formula is

[U validity rule]:

$$\begin{array}{l} P \\ \vdash \\ \langle a \rangle P \end{array}$$

This states that if P is valid (i.e., holds for all states), then evaluating the expression a results in a state in which P holds.

Two rules of inference usually present in a programming logic follow immediately from the predicate calculus axioms and [U validity rule]. They are Hoare's rules of consequence and composition.

THEOREM [consequ thm]:

$$\begin{array}{l} \{P\}a\{Q\},\ Q \Rightarrow R \\ \vdash \\ \{P\}a\{R\} \end{array}$$

PROOF. We can rewrite the assumptions as

$$P \Rightarrow \langle a \rangle Q, \quad \text{and} \quad Q \Rightarrow R$$

By [U validity rule], the second assumption gives

$$\langle a \rangle (Q \Rightarrow R)$$

From the definition of the extended term notation, this represents the same nonextended term as, and is therefore equivalent to,

$$(\langle a \rangle Q) \Rightarrow (\langle a \rangle R)$$

Combining the above with the first hypothesis yields

$$P \Rightarrow (\langle a \rangle R) \quad \text{or} \quad \{P\}a\{R\}$$

which was the desired conclusion. □

We can now easily derive a version of Hoare's statement composition rule as well.

THEOREM [comp thm]:

$$\begin{array}{l} P \Rightarrow \langle a \rangle Q, \quad Q \Rightarrow \langle b \rangle S \\ \vdash \\ P \Rightarrow \langle a;\ b \rangle S \end{array}$$

PROOF. This follows immediately if we let R be $\langle b \rangle S$ in the previous theorem. □

[11] We assume that we can substitute equals for equals. Note however that we have to be careful about the treatment of extended terms. It is not acceptable to conclude $\langle a \rangle t = \langle b \rangle t$ from $\langle a \rangle = \langle b \rangle$. In particular, $\langle x := 1;\ 13 \rangle = \langle x := 2;\ 13 \rangle$ holds, but $\langle x := 1;\ 13 \rangle \langle .x \rangle = \langle x := 2;\ 23 \rangle \langle .x \rangle$ does not.

The following sections present axioms and inference rules for specific programming language constructs. These can be interpreted as generalizations of Hoare's rules.

4.1 Traditional Expressions

The meaning of most expressions is given by a pair of axioms (technically axiom schemas). The first is a "value axiom" which defines the value returned by the expression. The second is an "effect axiom" which specifies how the value of an arbitrary term is affected by execution of the expression, and thus how the state of the computation is modified.

We start by considering a simple identifier or constant.

The value axiom states that the meaning of an identifier or constant in the programming language is the same as in the logic. Formally we write

[id val ax]:

$$\langle x \rangle = x$$

From [U validity rule] it follows that $\langle a; x \rangle = \langle a \rangle \langle x \rangle = \langle a \rangle x = x$. Recall that an identifier denotes its binding, and this cannot be changed by executing a.

The effect axiom specifies that the state is unchanged after execution of x. Thus the value of a term is not affected. We have

[id ef ax]:

$$\langle x \rangle t = t$$

The only difficulty with addition and conditionals is that we need to be careful not to assume that any of the subexpressions are side-effect-free. We give the following axiom pairs without further discussion:

[+ val ax]:

$$\langle a + b \rangle = \langle a \rangle + \langle a; b \rangle$$

[+ ef ax]:

$$\langle a + b \rangle t = \langle a; b \rangle t$$

[if val ax]:

$$\begin{aligned} \langle c \rangle &\Rightarrow \langle \textbf{if } c \textbf{ then } a \textbf{ else } b \textbf{ fi} \rangle = \langle c; a \rangle \\ \sim\langle c \rangle &\Rightarrow \langle \textbf{if } c \textbf{ then } a \textbf{ else } b \textbf{ fi} \rangle = \langle c; b \rangle \end{aligned}$$

[if val ax]:

$$\begin{aligned} \langle c \rangle &\Rightarrow \langle \textbf{if } c \textbf{ then } a \textbf{ else } b \textbf{ fi} \rangle t = \langle c; a \rangle t \\ \sim\langle c \rangle &\Rightarrow \langle \textbf{if } c \textbf{ then } a \textbf{ else } b \textbf{ fi} \rangle t = \langle c; b \rangle t \end{aligned}$$

4.2 Assignment and the Value of a Variable

It is easy enough to specify the value of an assignment:

[:= val ax]:

$$\langle a := b \rangle = \langle a; b \rangle$$

This is hardly surprising; the purpose of an assignment is after all to change the state, and not to produce an interesting value.

The effect of an assignment on the state is almost as easy to describe. We simply define the value returned by the "." or "value of" operation after an assignment. We have to distinguish the two cases in which the argument of the "value of" operation does, or does not, coincide with the left side of the assignment. Thus we get the following axioms:

[:= ef ax]:

$$\langle a \rangle = x \Rightarrow \langle a := b;\ .x \rangle = \langle a;\ b \rangle$$
$$\langle a \rangle \neq x \Rightarrow \langle a := b;\ .x \rangle = \langle a;\ b;\ .x \rangle$$

where x is an identifier.

Although Hoare's basic assignment axiom is not very complicated, it is perhaps not quite as intuitive as one might like; it takes an argument to convince the first-time reader of its correctness. The difficulty arises because we have to describe the effect of the assignment on the whole postcondition; we cannot decompose the problem and discuss the value of a subterm "after the assignment."

In our present system we have no such constraints. It suffices to specify the value of a simple variable after the assignment. If we wish to simplify a term $\langle x := a \rangle t$, for some general term t, we first simplify t so that all programming language terms have the form $\langle .x \rangle$. We then apply the above axioms to each such term in turn.

If we wish to prove

$$\langle x := 3 \rangle (\langle 13 + .x \rangle = 16)$$

we first use [+ val ax] to simplify this to

$$\langle x := 3 \rangle (\langle 13 \rangle + \langle 13;\ .x \rangle = 16)$$

With the aid of [id val ax] and [id ef ax], this can be further simplified to

$$\langle x := 3 \rangle (13 + \langle .x \rangle = 16)$$

By the definition of an extended term, this is equivalent to

$$13 + \langle x := 3;\ .x \rangle = 16$$

By [:= ef ax] (and [id val ax]), we finally simplify this to

$$13 + \langle x;\ 3 \rangle = 16$$

or just

$$13 + 3 = 16$$

Note that this amounts to applying Hoare's substitution axiom in those cases in which it applies, and, under the appropriate conditions (simple left side, no aliasing), we may continue to do so.

Avoiding substitution in the statement of the axioms gains us more than a simpler correctness argument for the axioms. Hoare's axiom relies on the fact that only an identifier can appear on the left side of an assignment, and that no two distinct identifiers are bound to the same location. This makes "location equivalence" a simple syntactic property, and substituting for identifiers becomes equivalent to "substituting for a location." This makes it difficult to extend

Hoare's axiom to real programming languages, in which these properties typically do not hold. The present formalism requires neither of these properties.

The preceding axioms assume that there is no "partial aliasing." In other words, locations are either the same or are independent; they may not overlap. This assumption can be relaxed simply by replacing "$\langle a \rangle \neq x$" by the condition that $\langle a \rangle$ and x do not overlap. We can then axiomatize arrays and records by defining how component locations relate to (i.e., whether they overlap, or are the same as) other locations. There is no need to change the assignment axioms. This captures the notion that the assignment operation has a uniform function, independent of what appears on the left side.

Since the result of the "value of" or "." operation depends directly on the state of the computation, we cannot make any general statement about it. Instead we put the burden of specifying how an operation interacts with an assignment on the assignment axioms.

The preceding axioms specified the result of the "value of" operation for an identifier argument only. The following axiom generalizes this to arbitrary expressions. We choose to label it as the "value" axiom for this operation:[12]

[. val ax]:

$$x = \langle a \rangle \Rightarrow \langle a; .x \rangle = \langle .a \rangle$$

We conclude with the obvious:

[. ef ax]:

$$\langle .a \rangle t = \langle a \rangle t$$

4.3 Declarations

We deal with variable allocation separately in the next section; here we are concerned only with the **let** block itself.

The purpose of a declaration is to bind a new name to some value. It is important to note that quantification already accomplishes this purpose in the predicate calculus. The programming language expression

$$\textbf{let } x == 3 + 5 \textbf{ in } x < 9 \textbf{ ni}$$

is different syntax for the predicate calculus formula

$$\forall x .(x = 3 + 5 \Rightarrow x < 9)$$

The following two axioms express this equivalence formally:

[**let** val ax]:

$$\forall x .(x = \langle b \rangle \Rightarrow \langle \textbf{let } x == b \textbf{ in } a \textbf{ ni} \rangle = \langle b; a \rangle)$$

[**let** ef ax]:

$$\forall x .(x = \langle b \rangle \Rightarrow \langle \textbf{let } x == b \textbf{ in } a \textbf{ ni} \rangle t = \langle b; a \rangle t)$$

Informally these axioms are easy to justify. If x is already bound to the value of b in the current state, then binding it again, as the **let** construct does, will not

[12] The axiom in fact deals more with argument parameter binding than anything else. In the full logic given in [2] it is a special case of an axiom which applies to arbitrary function applications.

influence anything. Thus the **let** construct is equivalent to just evaluating the two expressions in sequence.

Unfortunately, the preceding pair of axioms does not suffice. In particular, they allow the following interpretation of "**let** $x == b$ **in** a **ni**": "Check if x is bound to the value given by b; if not, abort; if so, evaluate a."

The **let** axioms must usually be applied in conjunction with the following axioms:

[rename ax]:

$$\langle \textbf{let } x == b \textbf{ in } a \textbf{ ni} \rangle = \langle \textbf{let } y == b \textbf{ in } a\{x \leftarrow y\} \textbf{ ni} \rangle$$
$$\langle \textbf{let } x == b \textbf{ in } a \textbf{ ni} \rangle t = \langle \textbf{let } y == b \textbf{ in } a\{x \leftarrow y\} \textbf{ ni} \rangle t$$

where x and y are identifiers, y does not occur in a, and $a\{x \leftarrow y\}$ denotes a with each occurrence of x replaced by y.

These axioms state that the name of a bound identifier does not matter; it can be replaced by any other identifier.

We can now give a procedure to "simplify" an arbitrary predicate which includes a **let** construct. Let LET be a term of the form:

$$\langle \alpha; \textbf{let } x == b \textbf{ in } a \textbf{ ni} \rangle$$

and let P(LET) be a formula containing the term.

We can use [rename ax] to insure that x does not occur elsewhere in P(LET).

Under these conditions, proving P(LET) reduces to proving

$$x = \langle \alpha; b \rangle \Rightarrow P(\langle \alpha; b; a \rangle)^{13}$$

Once this statement has been proven, we use [**let** val ax] to get

$$x = \langle \alpha; b \rangle \Rightarrow P(\text{LET})$$

We then use [rename ax] to remove any occurrences of x from P(LET). Predicate calculus rules now suffice to prove P(LET) itself.

We treat the case in which the **let** block appears somewhere before the end of the programming language term identically, except that [**let** ef ax] is used rather than [**let** val ax].

4.4 Variable Allocation

This section gives an axiomatic definition of the New function. The IsAllocated expression is used to aid us in this endeavor, so we will discuss it first.

The only interesting property of the IsAllocated function itself is that locations are not (visibly) deallocated. This is formally stated as

[IsAllocated ax]:

$$\langle \text{IsAllocated}(x) \rangle \Rightarrow \langle a; \text{IsAllocated}(x) \rangle$$

We cannot define the exact location produced by the New function, and neither do we want to. The only properties we are interested in are

(1) New never returns a previously allocated location.

[13] P(LET) should not explicitly quantify over variables in α or b. This can always be arranged by rewriting it in prenex normal form and then applying this transformation to the formula without any (explicit) quantifiers.

(2) The location returned by New will subsequently be allocated. This can be summarized as

[New val ax]:

$$\langle \text{IsAllocated}(x) \rangle \Rightarrow \langle \text{New}(\ ) \rangle \neq x$$
$$x = \langle \text{New}(\ ) \rangle \Rightarrow \langle \text{New}(\ );\ \text{IsAllocated}(x) \rangle$$

Allocating a new location does not affect the value stored at any location.[14] This is expressed by the effect axiom:

[New ef ax]:

$$\langle \text{New}(\ );\ .x \rangle = \langle .x \rangle$$

Note that the location returned by a subsequent call to New is affected.

4.5 The Complexity of Proofs

We have tried to keep the preceding proof system as simple as possible, but have not chosen it to minimize the length of a typical proof. We can justify this choice in several ways.

First, it is easier to develop confidence in a proof system that is based on a few "obvious" axioms. We can then transform the system into a more practical one by deriving further inference rules from the given set. These may be less "obvious," but they should have simple proofs based on the original system.

Second, our programming language definition can be less dependent on a programming methodology. We may later consider several sets of derived rules. One might consist of translations of Dijkstra's rules into our formalism to support his methodology. Another might support a more applicative programming style.

Third, we can give an algorithm for reducing theorems about programs (without loops or recursion) to the mathematical theorems involved in the correctness of the algorithm. Disregarding variable allocation for the moment, we can outline this procedure as follows. It is illustrated by an example in the next section.

The idea is to take an arbitrary formula, including programming language terms, and to replace it with an equivalent one which mentions only "simpler" programming language terms, until we finally arrive at a formula in which all programming language terms have the form $\langle .x \rangle$.

We start with a formula that includes programming language terms of the form

$$\langle \alpha; a \rangle \quad \text{or} \quad \langle \alpha; b; .x \rangle$$

where a does not have the form ".x", and where α is any, possibly empty, sequence of expressions. (Recall that extended terms $\langle a \rangle t$ are simply abbreviations.) We repeatedly decrease the size of such terms by applying the value axiom for the outermost operator of a to $\langle a \rangle$ or by applying the effect axiom for the outermost

[14] If New initializes the newly allocated location, then we have to treat the x = New( ) and $x \neq$ New( ) cases separately.

operator in b to $\langle b; .x\rangle$.[15] If $\langle a'\rangle$ is the simplified version of $\langle a\rangle$,[16] we get, by [U validity rule],

$$\langle\alpha\rangle(\langle a\rangle = \langle a'\rangle)$$

or

$$\langle\alpha; a\rangle = \langle\alpha; a'\rangle$$

The second step follows from the definition of the extended term syntax.

In the case of a **let** block we use the procedure outlined when we presented the axioms.

Thus we can simplify the formula to the point where all programming language terms have the form $\langle .x\rangle$. We then prove that this formula holds for all possible values of $\langle .x\rangle$ for each location x.[17]

The presence of the New function complicates matters slightly. We illustrate an approach, which will work if we are trying to prove a property of a program that uses New( ) only as the right side of a declaration.[18]

In this case we always deal with formulas P, which have the form

$$Q \Rightarrow P'$$

where there are no expressions containing IsAllocated in P', and Q is a (possibly empty) conjunction of terms of the form

$$\langle\alpha; \text{IsAllocated}(x)\rangle$$

We keep track of all known information about aliasing (equality of locations) in P'. We use Q only to infer additional information of this kind for newly allocated variables.

We simplify P' by using the above procedure, combined with the one following, for variable allocation. We can disregard Q once P' is simplified sufficiently so that no variables are allocated inside it. We then simply prove P' as before.

Assume that P has the form

$$Q \Rightarrow P'(\langle\beta; \textbf{let } x == \text{New( )} \textbf{ in } a \textbf{ ni}\rangle)$$

Using the procedure given above for declarations, we insure that x occurs nowhere else and simplify P to

$$Q \Rightarrow (x = \langle\beta; \text{New( )}\rangle \Rightarrow P'(\langle\beta; \text{New( )}; a\rangle) \qquad (*)$$

[15] For the induction to go through, the "size" or "simplicity" ordering can be any well-founded partial order on programming language expressions. In a few places it is convenient to use a somewhat contrived ordering. In particular, $\langle a; b\rangle$ is smaller than $\langle a + b\rangle$. We also need to insure that $.x$ is much smaller than $.a$, where x is an identifier, and the expression a is more complicated. We can then "simplify" a formula mentioning $\langle .a\rangle$ to one mentioning $\langle a; .x\rangle$.

[16] In some cases, like the conditional, it will be simplified to a term involving two or more simpler programming language expressions. The procedure in these cases is essentially the same.

[17] More formally, we replace each $\langle .x\rangle$ by $f(x)$ and then show that the resulting formula holds for all functions f. If we have only universally quantified location variables, this amounts to showing that for each conceivable combination of aliases, and for each assignment of values to each group of aliased variables, the formula holds. Thus, in the interesting cases, the quantification over the function f is not really necessary.

[18] It is possible to generalize this. As we point out in the next section, though, the axioms dealing with New are intentionally incomplete. Thus we will never be able to deal with certain kinds of formulas. [2] gives axioms which are (relatively) complete, roughly in the sense of [10].

The clause "$x = \langle \beta;$ New($)\rangle$" is of interest for two reasons. First, it tells us that x will not be aliased by subsequently allocated variables. Second, it tells us that x itself does not alias any previously allocated variables. We can make this explicit by rewriting (*) as

$$(Q \wedge \langle \beta; \text{New}(\ ); \text{IsAllocated}(x)\rangle) \Rightarrow (\wedge_i\, x \neq y_i) \wedge P'(\langle \beta; \text{New}(\ ); \alpha\rangle) \quad (**)$$

where the y_i are all the variables such that

$$\langle \alpha_i; \text{IsAllocated}(y_i)\rangle$$

appears in Q and α_i is a prefix of β.[19] It follows from [IsAllocated ax] and [New val ax] that (**) implies (*). The occurrence of "New( )" in the middle of a term can eventually be removed by applying [New ef ax]. Thus we have succeeded in removing the declaration from P'.

We deal with the case in which the declaration does not appear at the end of the term in a similar manner.

The existence of a simple algorithm to reduce proofs of loop- and recursion-less programs to mathematical proofs is hardly new. We can easily obtain an equivalent procedure for systems such as Dijkstra's weakest precondition calculus.

It would be possible, but useless, to develop a similar procedure for loops. The procedure itself would be much more complex and the resulting mathematical formula would probably be useless in the construction of a proof.

This argues that the only real complexity in a program proof lies in the underlying mathematical proof and in the treatment of loops and recursion. The latter can be treated similarly in our logic and in more conventional ones. The number of steps involved in proving a "straight-line" program is nearly irrelevant, provided we have even minimal machine assistance.

4.6 An Example

We illustrate the procedure outlined in the previous section with a sample proof.

Assume we want to prove that the following section of code interchanges the values of x and y:

```
let
  t == New( )
in
  t := .x;
  x := .y;
  y := .t
ni
```

[19] We are interested in identifiers y_i declared in an enclosing scope. In this case checking for a simple prefix suffices. Consider simplifying

$$\langle \gamma; \textbf{let } y == \text{New}(\ ) \textbf{ in } \alpha; \textbf{let } x == \text{New}(\ ) \textbf{ in } a \textbf{ ni ni}\rangle$$

In simplifying the outer block we add the clause

$$\langle \gamma; \text{New}(\ ); \text{IsAllocated}(y)\rangle$$

to Q, and simplify P' so it only mentions the term

$$\langle \gamma; \text{New}(\ ); \alpha; \textbf{let } x == \text{New}(\ ) \textbf{ in } a \textbf{ ni}\rangle$$

Note that "γ; New ( )" is a prefix of "γ; New( ); α; New( )".

Let us abbreviate the above program segment as α and the body of the **let** block as β. We wish to prove

$$\text{IsAllocated}(x) \wedge \text{IsAllocated}(y) \Rightarrow \langle \alpha; .x \rangle = \langle .y \rangle \wedge \langle \alpha; .y \rangle = \langle .x \rangle$$

Note that this holds whether or not x and y are aliases. Using the above strategy we reduce this problem to proving

$$\langle \text{IsAllocated}(x) \rangle \wedge \langle \text{IsAllocated}(y) \rangle \wedge \langle \text{New}(\ ); \text{IsAllocated}(t) \rangle \Rightarrow ((x \neq t \wedge y \neq t) \Rightarrow \langle \text{New}(\ ); \beta; .x \rangle = \langle .y \rangle \wedge \langle \text{New}(\ ); \beta; .y \rangle = \langle .x \rangle)$$

Since there are no further calls to New, we no longer have use for the information about allocated variables. We prove the conclusion of the main implication.

To shorten the proof somewhat, we immediately, and without further mention, make use of [U validity rule] and the identity

$$\langle \gamma; x \rangle = x$$

where x is an identifier. The latter is justified by [id val ax].

We also make use of propositional logic to keep formulas structured in the form

$$\wedge_i (P_i \Rightarrow Q_i)$$

where the P_i are distinct and contain hypotheses about aliasing of variables. The Q_i represent the rest of the formula. (Note that both of these steps could easily be incorporated into our algorithm. Thus our proof could easily be obtained automatically.)

We apply our procedure to simplify

$$x \neq t \wedge y \neq t \Rightarrow \langle \text{New}(\ ); \beta; .x \rangle = \langle .y \rangle$$

The other half of the proof is virtually identical.

We get

$$x \neq t \wedge y \neq t \Rightarrow \langle \text{New}(\ ); t := .x; x := .y; y := .t; .x \rangle = \langle .y \rangle$$
$$x \neq t \wedge y \neq t \wedge y \neq x \Rightarrow \langle \text{New}(\ ); t := .x; x := .y; y; .t; .x \rangle = \langle .y \rangle$$
$$\wedge\ x \neq t \wedge y \neq t \wedge y = x \Rightarrow \langle \text{New}(\ ); t := .x; x := .y; y; .t; \rangle = \langle .y \rangle$$
[:= ef ax]

$$x \neq t \wedge y \neq t \wedge y \neq x \Rightarrow \langle \text{New}(\ ); t := .x; x := .y; y; t; .x \rangle = \langle .y \rangle$$
$$\wedge\ x \neq t \wedge y \neq t \wedge y = x \Rightarrow \langle \text{New}(\ ); t := .x; x := .y; y; .t; \rangle = \langle .y \rangle$$
[. ef ax]

$$x \neq t \wedge y \neq t \wedge y \neq x \Rightarrow \langle \text{New}(\ ); t := .x; x := .y; y; .x \rangle = \langle .y \rangle$$
$$\wedge\ x \neq t \wedge y \neq t \wedge y = x \Rightarrow \langle \text{New}(\ ); t := .x; x := .y; y; .t; \rangle = \langle .y \rangle$$
[id ef ax]

$$x \neq t \wedge y \neq t \wedge y \neq x \Rightarrow \langle \text{New}(\ ); t := .x; x := .y; .x \rangle = \langle .y \rangle$$
$$\wedge\ x \neq t \wedge y \neq t \wedge y = x \Rightarrow \langle \text{New}(\ ); t := .x; x := .y; .t; \rangle = \langle .y \rangle$$
[id ef ax]

$$x \neq t \wedge y \neq t \wedge y \neq x \Rightarrow \langle \text{New}(\ ); t := .x; .y \rangle = \langle .y \rangle$$
$$\wedge\ x \neq t \wedge y \neq t \wedge y = x \Rightarrow \langle \text{New}(\ ); t := .x; .t \rangle = \langle .y \rangle$$
[:= ef ax], [. ef ax], [id ef ax]

$$x \neq t \wedge y \neq t \wedge y \neq x \Rightarrow \langle \text{New}(\ );\ .y \rangle = \langle .y \rangle$$
$$\wedge\ x \neq t \wedge y \neq t \wedge y = x \Rightarrow \langle \text{New}(\ );\ .x \rangle = \langle .y \rangle$$

[:= ef ax], [. ef ax], [id ef ax]

$$x \neq t \wedge y \neq t \wedge y \neq x \Rightarrow \langle .y \rangle = \langle .y \rangle$$
$$\wedge\ x \neq t \wedge y \neq t \wedge y = x \Rightarrow \langle .x \rangle = \langle .y \rangle$$

[New ef ax]

We have to show that the preceding holds for all possible contents of all locations. We consider all possible combinations of aliases between x, y, and t and all possible value assignments to each group. We can immediately discard the cases in which t is aliased by either x or y.[20]

In the $y = x$ case, the first half of the formula simplifies to **true**. Thus we have, for all possible values X of x and y,

$$x \neq t \wedge y \neq t \wedge y = x \Rightarrow X = X$$

In the $y \neq x$ case, we get for all values X and Y of x and y, respectively:

$$x \neq t \wedge y \neq t \wedge y \neq x \Rightarrow Y = Y$$

It is easily shown, using [. val ax], that these together imply the preceding formula.

As expected, both of the resulting formulas are equivalent to **true**, and the proof is thus complete.

5. EVALUATION AND COMPARISON WITH HOARE LOGIC

We have outlined a programming logic that can be used to reason about a number of programming language constructs which have traditionally been considered difficult to work with. We have done so by choosing a primitive that takes advantage of the underlying "expression" character of the language.

This choice of primitive not only makes the above axiomatization feasible, but it also gives us, at least in an informal sense, somewhat more expressive power than Hoare logic. For example, if we want to state in Hoare logic that "$x := x * x$" squares the value of x, we need to introduce an auxiliary variable so that we can talk about the initial value of x in the postcondition. In our formalism we simply write

$$\langle x := .x * .x;\ .x \rangle = \langle .x \rangle^2$$

This increase in expressive power also introduces some complications. We can make statements of the form

$$\langle \alpha;\ \text{New}(\ ) \rangle = \langle \beta;\ \text{New}(\ ) \rangle$$

where α and β are arbitrarily complicated and completely unrelated expressions. Clearly, we should not care whether or not such an equality holds. (In practice this would be highly sensitive to the storage management scheme used.) Our axioms in general do not suffice, and probably should not suffice, for proving such statements. This makes it rather difficult to even state a reasonable

[20] Simple propositional reasoning suffices to discover this. In fact, given that we keep formulas structured as above, we only allow those combinations of aliases admitted by the P_i.

completeness theorem for the logic. (A different axiomatization of New( ) is used to accomplish this in [2]. Brooks [5] circumvents this problem in his treatment of storage allocation by avoiding the introduction of explicit locations. But this requires a different setting.)

The problem is that there are certain properties of the language we want to leave undefined. Sometimes, as here, by far the easiest way to do so is to specify a weak set of axioms and to state that nothing else is to be known about the construct. Unfortunately, this makes it hard to use an approach similar to that of [10] to be convinced that nothing else was accidentally omitted.

Another weakness of this logic lies in the fact that *the* value of an expression in a certain state has to be a well-defined concept. Thus, nondeterminism, and as a result, concurrency are difficult to accommodate. This characteristic also forces the nonstandard treatment of nontermination in [2]. Hoare logic shares this problem only if we allow user-defined functions to be mentioned in assertions.

It is thus difficult to argue that our approach is uniformly superior to all others. Rather, we argue that one might benefit from matching the programming logic to the programming language, and that it is not always reasonable to use a fixed logical formalism in evaluating characteristics of all programming languages.

ACKNOWLEDGMENTS

I would like to thank Alan Demers for his many comments on the material presented here. Many other people made helpful comments at earlier presentations of this material. Observations by Bob Constable, Mike O'Donnell, Ravi Sethi, Leslie Lamport, David Harel, and by the referees resulted in direct improvements to the paper.

REFERENCES

1. BOEHM, H.-J. A logic for expressions with side effects. In *Proceedings of the 9th Annual ACM Symposium on Principles of Programming Languages* (Jan. 1982), ACM, New York, 268–280.
2. BOEHM, H.-J. A logic for the Russell programming language. Ph.D. thesis, Cornell Univ., Jan. 1984. (Also available as Computer Science Tech. Rep. TR84-593.)
3. BOEHM, H.-J., DEMERS, A., AND DONAHUE, J. A programmer's introduction to Russell. Tech. Rep. 85-16, Dept. of Computer Science, Rice Univ., 1985. See also [12].
4. BOYER, R. S., AND STROTHER MOORE, J. *A Computational Logic.* Academic Press, New York, 1979.
5. BROOKES, S. D. A fully abstract semantics and a proof system for an Algol-like language with sharing. Tech. Rep. CMU-CS-84-118A, Dept. of Computer Science, Carnegie-Mellon Univ., Feb. 1985.
6. CARTWRIGHT, R., AND OPPEN, D. The logic of aliasing. *Acta Inf. 15* (1981), 365–384.
7. CLARKE, E. M., JR. Programming language constructs for which it is impossible to obtain good Hoare axioms. *J. ACM 26*, 1 (Jan. 1979), 129–147.
8. CONSTABLE, R. L. On the theory of programming logics. In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing* (May 1977), ACM, New York.
9. CONSTABLE, R. L., AND O'DONNELL, M. J. *A Programming Logic.* Winthrop, Cambridge, 1978.
10. COOK, S. Soundness and completeness of an axiom system for program verification. Tech. Rep. 95, Dept. of Computer Science, Univ. of Toronto, June 1976.
11. CUNNINGHAM, R. J., AND GILFORD, M. E. J. A note on the semantic definition of side effects. *Inf. Process. Lett. 4*, 5 (Feb. 1976), 118–120.
12. DEMERS, A. J., AND DONAHUE, J. E. Data types are values. Tech. Rep. 79-393, Dept. of Computer Science, Cornell Univ., 1979.

13. DIJKSTRA, E. W. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1976.
14. GRIES, D., AND LEVIN, G. Assignment and procedure call proof rules. *ACM Trans. Program. Lang. Syst. 2*, 4 (Oct. 1980), 564–579. See also [15].
15. GRIES, D. *The Science of Programming.* Springer Verlag, New York, 1981.
16. HAREL, D. *First-Order Dynamic Logic.* Springer Verlag, New York, 1979.
17. HEHNER, E. C. R. Predicative programming: Part I. *Commun ACM 27*, 2 (Feb. 1984).
18. HOARE, C. A. R. An axiomatic basis for computer programming. *Commun. ACM 12*, 10 (Oct. 1969), 576–580.
19. JENSEN, K., AND WIRTH, N. *Pascal User Manual and Report.* Springer Verlag, New York, 1974.
20. KOWALTOWSKI, T. Axiomatic approach to side effects and general jumps. *Acta Inf. 7*, 4 (1977), 357–360.
21. MCCARTHY, J. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg, Eds., North-Holland, Amsterdam, 1963. See also [24].
22. MIRKOWSKA, G. On formalized systems of algorithmic logic. *Bull. de L'Academie Polonaise des Sciences, Serie des Sciences Math., Astr. et Phys. 19*, 6 (1971), 421–428.
23. O'DONNELL, M. J. A critique of the foundations of Hoare-style programming logics. *Commun. ACM 25*, 12 (Dec. 1982).
24. PARK, D. Fixpoint induction and proofs of program properties. *Mach. Intell. 5*, American Elsevier, New York, 1970, 59–78.
25. POPEK, G. J., HORNING, J. J., LAMPSON, B. W., MITCHELL, J. G., AND LONDON, R. L. Notes on the design of Euclid. In *Proceedings of the ACM Conference on Language Design for Reliable Software. SIGPLAN Not. 12*, 3 (Mar. 1977), 11–18.
26. PRITCHARD, P. Program proving—expression languages. In *Information Processing 77*, North-Holland, Amsterdam, 1977, 727–731. For more details see: An axiomatic semantics for expression languages. Thesis, Australian National Univ., Nov. 1979. Available as a joint technical report from the Computer Science Depts. at the Australian National Univ. (TR-CS-80-11) and the Univ. of Queensland (TR-20).
27. SCHWARTZ, R. L. An axiomatic treatment of asynchronous processes in Algol 68. Preliminary draft. More details can be found in: An axiomatic semantic definition of Algol 68, Computer Science Dept., UCLA-34P214-75, Univ. of California, Los Angeles, July 1978.
28. VAN WIJNGAARDEN, A., MAILLOUX, B. J., PECK, J. E. L., KOSTER, C. H. A., SINTZOFF, M., LINDSEY, C. H., MEERTENS, L. G. L. T., AND FISKER, R. G. Revised report on the algorithmic language Algol 68. *Acta Inf. 5*, 1-3 (1975), 1–236.
29. WULF, W. A., ET AL. *BLISS-11 Programmer's Manual.* Digital Equipment Corp., Maynard, Mass., 1972.

Received June 1984; revised July 1985; accepted July 1985