1980

A Critique of the Foundations of Hoare-Style Programming Logics

Michael O'Donnell

Report Number:
80-338

A Critique Of The Foundations Of Hoare-Style Programming Logics

Michael O'Donnell

CSD-TR 338

Dept. of Computer Science

Purdue University

W. Lafayette, IN 47907

A Critique of the Foundations of Hoare-Style Programming Logics

Michael J. O'Donnell

Purdue University

ABSTRACT

Much recent discussion in computing journals has been devoted to arguments about the feasibility and usefulness of formal verification methods for increasing confidence in computer programs. Too little attention has been given to precise criticism of specific proposed systems for reasoning about programs. Whether such systems are to be used for formal verification, by hand or automatically, or as a rigorous foundation for informal reasoning, it is essential that they be logically sound. Several popular rules in the Hoare language are in fact not sound. These rules have been accepted because they have not been subjected to sufficiently strong standards of correctness. This paper attempts to clarify the different technical definitions of correctness of a logic, to show that only the strongest of these definitions is acceptable for Hoare logic, and to

April 9, 1980

correct some of the unsound rules which have appeared in the literature. The corrected rules are given merely to show that it is possible to do so. Convenient and elegant rules for reasoning about certain programming constructs will probably require a more flexible notation than Hoare's.

Key words and phrases: verification, soundness, partial correctness, defined functions, Goto, logic.

CR categories: 5.21, 5.24, 4.29.

1. Introduction

Logic is the study of the relation between a symbolic language and its meaning, with special emphasis on the legitimate ways of reasoning in the language. A primary accomplishment of Mathematical Logic in the earlier part of this century was the formalization of the First Order Predicate Calculus, a logical language which is generally regarded as sufficient in principle for nearly all mathematical discourse. Formal rules for reasoning in the First Order Predicate Calculus have been shown to be correct and powerful enough to derive all true theorems of this language. In the last decade, new languages and formal rules for reasoning about programs have been proposed, and attempts have been made to justify the correctness of these rules.

A particularly popular language for reasoning about

programs is the language of Hoare triples [13]. The Hoare language includes the formulae of the First Order Predicate Calculus, plus triples of the form A{P}B, with A and B Predicate Calculus formulae and P a program or part of a program. Such a triple is intended to mean that, if the initial state of a machine satisfies the assertion A, then after running the program P, B must be true of the final state. Unfortunately, several different definitions of the correctness of a system of reasoning, which are equivalent for the Predicate Calculus, are not equivalent for the Hoare language. So we must be very careful when studying rules for reasoning in the Hoare language to use a criterion for correctness which corresponds to our intuitive idea of legitimate reasoning. Several articles on Hoare logic in the past few years [6,16,19] have attempted to justify rules of reasoning by criteria which are insufficient to give intuitive confidence in the derivations which are carried out by such rules.

There are three main reasons for using a formal presentation of logic instead of relying solely on intuition when reading and writing technical arguments:

[1] A formal presentation provides a uniform standard which may be used as a final authority in disagreements.

[2] Formal presentation makes a system of reasoning into a mathematical object which may be studied objectively to discover its properties.

[3] A formally presented system may be processed automatically by computers.

To be useful for any of these three purposes, a formal system must be intuitively correct. A common enterprise in logic is to formalize the notion of correctness and to prove that a formal system is correct. Along with such a proof, a careful intuitive inspection of the formal definition of correctness is essential, since everything hinges on this definition. Such careful scrutiny has generally been omitted in published work on Hoare logics. The purpose of this paper is to begin such a scrutiny. I will show that several proposed rules for reasoning about programs have been judged by faulty standards of correctness, and are in fact incorrect by the proper standards.

Section 2 describes four different technical definitions of correctness and argues that only the strongest of these definitions is intuitively sufficient. Section 3 introduces the Hoare language and its meaning. Section 4 shows the well-known correct rules for reasoning about programs with assignments, conditionals and while loops. Section 4 extends the rules to handle programs with function definitions. The first two published attempts to give rules for function definitions [6,16,19] were incorrect. Section 5 discusses the problems of reasoning about programs with Goto commands. The best-known rule for reasoning about Gotos [6] is also incorrect, although it satisfies a weaker

condition which is sometimes mistaken for correctness.

2. Criteria for correctness of a logical system

Two primary requirements are known for the correctness of a system of reasoning, each with several variations in its technical definitions. Consistency refers to the inability of a system to derive an explicit contradiction, while the stronger notion of soundness says that everything derived in a system is in some sense true. There are two natural definitions of consistency.

Definitions

Assume that a relation contradictory(Φ) has been defined on finite sets Φ of formulae in a language so that contradictory(Φ) captures the intuitive notion that the formulae in Φ are explicitly contradictory.

A system of reasoning is strongly consistent if it is not possible to prove all of the formulae in a set Φ such that contradictory(Φ).

A system of reasoning is weakly consistent if it is not possible to prove a single formula F such that contradictory($\{F\}$).

Strong consistency certainly implies weak consistency.

In the First Order Predicate Calculus, contradictory(Φ) holds whenever Φ contains two formulae of the forms F and $\neg F$

or a single formula of the form (F&¬F), or the formula False. Other sets of formulae may be taken as contradictory as long as it is obviously impossible for all formulae in the set to be true. Since (F&¬F) (equivalently, False) is provable if and only if F is provable and ¬F is provable, weak and strong consistency are equivalent for the First Order Predicate Calculus with the definition of contradictory above, or with any reasonable more liberal definition. But in Hoare logics, two formulae A{P}B and C{Q}D cannot be combined with a symbol like &. So weak and strong consistency might not be equivalent for systems of reasoning in Hoare languages. I show in Section 5 that a system proposed by Musser [16,19] for reasoning about function definitions in Euclid is weakly consistent but not strongly consistent. The proposed system violates the principle that (F&¬F) is provable if and only if F and ¬F are each provable.

Strong consistency, for some reasonable definition of contradictory, is intuitively a necessary condition for the correctness of a logical system, but it is not in general a sufficient condition, since a system might prove a formula which is false but does not contradict any other provable formula.

Definitions

A set of formulae Φ implies a formula F if F is true in every world in which all the formulae in Φ are true.

A logical system is theorem sound if every provable formula is true.

A logical system is inferentially sound if, for every set of formulae Φ and every formula F, if F can be proved from assumptions in Φ, then Φ implies F.

In any system where contradictory formulae cannot all be true, theorem soundness implies strong consistency. If, in addition, there exists a trivially true formula F which cannot possibly be useful as an assumption (for example, F might already be an axiom), then inferential soundness implies theorem soundness.

In the First Order Predicate calculus, F is provable from assumptions in Φ if and only if there is some finite subset $\{F_1, \ldots, F_n\}$ of Φ such that $((F_1 \& \ldots \& F_n) \Rightarrow F)$ is provable with no assumptions. Since the meaning of the implication symbol is just that the left side implies the right side, theorem and inferential soundness are equivalent for the First Order Predicate Calculus. In Hoare logics, it is not always possible to join two formulae with an implication sign, so theorem soundness may be weaker than inferential soundness.

Although theorem soundness seems at first glance to be enough for an intuitive claim of correctness, this weaker form of soundness only justifies the theorems of a system, not the methods of reasoning. If a formal system is to pro-

vide a satisfactory foundation for actual reasoning, the methods of proof should be intuitively correct, not just symbol manipulation tricks which fortuitously produce true theorems at the end. One might argue that certain rules for program verification are intended only for automatic theorem proving, not for human consumption, so that the steps of reasoning are not important as long as the answer is right. Even from such a restricted point of view, theorem soundness is at best not a very robust notion.

Suppose that a certain logical system is incomplete, so that some particular true formula F cannot be proved. Such a system might be theorem sound, even though assuming F would lead to a proof of some false or even contradictory formula G. Any attempt to extend this system by adding true formulae as axioms or by providing additional correct rules of inference would be very dangerous, since once the true formula F became provable, so would the false formula G. In Section 6 I show that the rules for reasoning about Goto commands proposed by Clint and Hoare [6] create a system of reasoning with this dangerous property: because of the lack of inferential soundness, addition of true axioms yields an inconsistency. Arbib and Alagić [1,3] also noticed a problem with the Clint and Hoare Goto rule. In inferentially sound systems every step of reasoning is correct, so soundness is preserved when additional true axioms or additional sound rules are added.

3. Meanings of formulae in Hoare logics

Recall that a Hoare formula is either a formula of the First Order Predicate Calculus or a triple A{P}B with A and B formulae of the Predicate Calculus and P a program or program segment (some people prefer to write {A}P{B}). Predicate Calculus formulae are built from function, constant and variable symbols, relational symbols, the equality sign, and the usual logical symbols & (and), V (or), ¬ (not), => (implies), ∀x (for all x) and ∃x (there exists x). For example,

$$\forall x \exists y\ (y>x \ \&\ \mathrm{Prime}(y))$$

is a Predicate Calculus formula expressing the fact that there exist arbitrarily large primes. Such formulae have the standard meanings, which correspond exactly to the intuition; see [18] for a formal treatment.

Great effort has gone into formalizing the meanings of programs [22,11], but for this discussion I will use only programs whose meanings are intuitively obvious. There are two popular ways to define the meaning of a Hoare triple A{P}B, which differ in their treatments of cases where P fails to halt.

Definitions

A Hoare triple A{P}B is a true partial correctness formula if, whenever the program segment P begins execu-

tion with its first command, in a state for which A is true, and P terminates normally by executing its last command, then B is true of the resulting final state.

A{P}B is a true total correctness formula if, whenever P begins execution with its first command, in a state for which A is true, then P terminates normally by executing its last command, and B is true of the resulting final state.

For example,

A{While True do x:=x end}B

is always a true partial correctness formula, independently of A and B. Partial correctness formulae make no distinction between failure to terminate and abnormal or unsuccessful termination due to an error such as division by zero. The formula above is a false total correctness formula as long as there exists a state for which A is true. False{P}B is a true formula for both partial and total correctness. If P always halts when started in a state for which A is true, then the partial and total correctness meanings for A{P}B are the same. For example,

$x>0\&y>0$\{z:=1; i:=0; While $i<y$ do $z:=z*x$; $i:=i+1$ end\} $z=x^y$

is a true formula for both partial and total correctness, roughly expressing the fact that the program inside the braces computes x to the y power. To achieve machine

independence, programs in Hoare formulae are assumed to be executed on an ideal machine with an arbitrarily large memory capacity, so that there are no overflows.

The partial correctness meaning for Hoare triples is more popular than the total correctness meaning because it is thought to be easier to deal with in formal proofs. Of course a partial correctness proof for a program is only valuable if we convince ourselves by some other means that the program halts. In the rest of this discussion, Hoare triples will always be interpreted as partial correctness formulae unless otherwise stated.

For the Hoare language contradictory(Φ) should hold whenever some Predicate Calculus subset of Φ is contradictory. Also, if Y is a contradictory set of Predicate Calculus formulae, and P is a well-formed program which obviously halts (e.g., a program with no loops), and if Φ contains all the formulae True{P}A for A in Y, then Φ is contradictory. Any additional intuitively contradictory sets of formulae may be added to the definition of contradictory(Φ) without affecting the following discussion.

4. Proof rules for programs with conditional and while

Consider a programming language with simple assignments

x := E

for expressions E, a command

Null

which does nothing, a command

Fail

which never terminates normally, two-branched conditionals of the form

If A then P else Q end,

and loops of the form

While A do P end.

Commands may be sequenced in the Pascal style with semicolon separators. Of course, Null and Fail are not needed, but they are convenient for discussion.

Assume that we have taken some sufficiently powerful proof rules from Mathematical Logic for all of our Predicate Calculus reasoning. In order to prove theorems in the form A{P}B we need additional rules for reasoning about programs. Such rules are commonly written in the form

$$\frac{F_1, \ldots, F_n}{G}$$

where $F_1, \ldots, F_n$ and G are schematic descriptions of formulae. The meaning of such a rule is that, whenever the hypotheses $F_1, \ldots, F_n$ have already been proved, we may

prove the conclusion G in one more step. Sometimes restrictions are also given which limit the allowed applications of the rule. A rule with no hypotheses is often called an axiom or postulate.

The following well-known set of proof rules [13] is inferentially sound [8] for partial correctness Hoare logic with the conditional-while programming language described above:

Null: $\dfrac{}{A\{\underline{\text{Null}}\}A}$

Fail: $\dfrac{}{A\{\underline{\text{Fail}}\}B}$

In the next rule note that A(E/x) means A with the expression E replacing all free occurrences of x. A variable occurrence x is free as long as it is not in a subformula beginning with $\forall x$ or $\exists x$. In the process of replacing x by E, quantified variables in $\forall y$ and $\exists y$ within A must be renamed so that all variables in E remain free after substitution.

Assignment: $\dfrac{}{A(E/x)\{x:=E\}A}$

$$\text{Composition-1:}\quad \frac{A\{P\}B,\ B\{Q\}C}{A\{P;Q\}C}$$

$$\text{Conditional:}\quad \frac{A\&B\{P\}C,\ A\&\neg B\{Q\}C}{A\{\underline{\text{If}}\ B\ \underline{\text{then}}\ P\ \underline{\text{else}}\ Q\ \underline{\text{end}}\}C}$$

$$\text{While:}\quad \frac{A\&B\{P\}A}{A\{\underline{\text{While}}\ B\ \underline{\text{do}}\ P\ \underline{\text{end}}\}A\&\neg B}$$

$$\text{Consequence:}\quad \frac{A\Rightarrow B,\ B\{P\}C,\ C\Rightarrow D}{A\{P\}D}$$

To see that these rules are inferentially sound, we merely check each rule individually to see that whenever the hypotheses are true, the conclusion must also be true. Since combinations of inferentially sound systems are inferentially sound, we need not consider the possible interactions between rules. Cook [8] has shown that these rules are sufficiently powerful to prove all true statements in the Hoare language of conditional-while programs.

5. Defined functions

Let us add to the conditional-while programming language the ability to define functions by means of subprograms. For simplicity, consider only recursion-free (i.e.,

noncircular) definitions of unary functions, with no nesting of definitions, no side-effects and no global variables. Such a simple version of function definitions already provides interesting pitfalls for Hoare logic. Function definitions will be written in the form

f: Function(x); local $z_1,\ldots,z_n$; P; return(y) end

$x,y,z_1,\ldots,z_n$ must be distinct and must contain all variables in P. n may be 0, in which case there are no local variables, and the phrase local $z_1,\ldots,z_n$; is omitted. The form return(y) must occur exactly once, at the end, and should be thought of as a punctuation like Function(x) rather than a command. The value of x must not be changed in P. Any changes to the values of $y,z_1,\ldots,z_n$ within P have no effect on the values of these variables outside of the function definition.

Clint and Hoare [6,14] proposed the following rule:

Function-1:

$$\frac{A\{P\}B}{\forall x(A \Rightarrow B(f(x)/y))}$$

where f has been defined as

f:Function(x); local $z_1,\ldots,z_n$;
P; return(y) end

and A and B do not contain $z_1,\ldots,z_n$ free.

Ashcroft [4] noticed that adding the rule Function-1 to those of Section 4 yields an inconsistency. Let f be defined as

(*) f: Function(x); Fail; return(y) end.

Consider the following derivation:

1)	True{Fail}False	Fail
2)	∀x True=>False	Function-1, 1)
3)	False	Predicate Calculus

So, the system containing Function-1 is not even weakly consistent.

It may appear that Function-1 only derives contradictions from pathological function definitions which never halt. A similar contradiction arises whenever a defined function fails to halt for some possible argument, even if the value of the function is never computed for that argument. For example, it is very natural to define the factorial function by a program which works correctly for positive arguments, but computes forever on negative arguments. The presence of such a definition leads to a contradiction even if factorial is only computed for positive arguments.

Alagić and Arbib [1] present the rule Function-1 with an informal warning that the function body must halt when A is true initially. For a logical rule to be useful, we must be able to decide when the rule has been applied correctly. Alagić and Arbib's restriction, taken literally, cannot be formalized in an acceptable fashion, since the halting of P is undecidable. One reasonable way to fix the rule

Function-1 with such a restriction is to provide means for proving termination, that is, to use a total correctness logic instead of partial correctness. Alternatively, the rule could be restricted to some decidable proper subset of the set of all function bodies which halt.

The inconsistency in Function-1 is essentially Russell's paradox [21] in disguise. Russell's paradox arises from the definition of a set R as the set of all sets which do not contain themselves. Does R contain itself? A set may be represented by a function, called the characteristic function, which returns 1 for inputs in the set and 0 for inputs not in the set. Russell's set R is represented by the defined function

```
r: Function(g); y:=1-g(g); return(y); end
```

Now, the following derivation mimics Russell's paradox:

1)	$1-g(g)=1-g(g)\{y:=1-g(g)\}y=1-g(g)$	Assignment
2)	$True \Rightarrow 1-g(g)=1-g(g)$	Predicate Calculus
3)	$y=1-g(g) \Rightarrow y\neq g(g)$	Arithmetic
4)	$True\{y:=1-g(g)\}y\neq g(g)$	Consequence, 1),2),3)
5)	$\forall g(True \Rightarrow r(g)\neq g(g))$	Function-1, 4)
6)	$r(r)\neq r(r)$	Predicate Calculus, 5)

Musser [16,19] proposed a modified function rule in Euclid notation. Musser's basic idea is that the paradox of Function-1 arises when formulae A and B are chosen in such a

way that there does not exist a function f satisfying $\forall x(A \Rightarrow B(f(x)/y))$. The existence of such a function may easily be expressed in the First Order Predicate Calculus as $\forall x(A \Rightarrow \exists yB)$. To avoid the extra step of substituting various values for x, Musser includes the substitution in his rule. Musser's rule covers recursion, a form of data abstraction, and more complicated uses of parameters, but, for my restricted function definitions, the rule is essentially

Function-2:

$$\frac{\exists y(A(E/x) \Rightarrow B(E/x)),\ A\{P\}B}{(A(E/x) \Rightarrow B(E/x,f(E)/y))}$$

where f has been defined as

f:Function(x); local $z_1,\ldots,z_n$;
P; return(y) end

and A and B do not contain $z_1,\ldots,z_n$ free.

This rule may be applied with only one choice of A and B for each function definition.

The additional hypothesis $\exists y(A(E/x) \Rightarrow B(E/x,f(E)/y))$ prevents the simple contradiction which arose from Function-1. Now we need two proofs to derive a contradiction. Let f again be defined by a body which never halts (*).

1)	True{Fail}False	Fail
2)	False => y=0	Predicate Calculus
3)	True => True	Predicate Calculus
4)	True{Fail}y=0	Consequence, 1),2),3)
5)	∃y(True => y=0)	Predicate Calculus
6)	True => f(0)=0	Function-2, 4),5)
7)	f(0)=0	Predicate Calculus, 6)

Similarly,

1)	True{Fail}False	Fail
2)	False => y≠0	Predicate Calculus
3)	True => True	Predicate Calculus
4)	True{Fail}y≠0	Consequence, 1),2),3)
5)	∃y(True => y≠0)	Arithmetic
6)	True => f(0)≠0	Function-2, 4),5)
7)	f(0)≠0	Predicate Calculus, 6)

So, the system containing the rules of Section 4 as well as Function-2 is not strongly consistent. It is weakly consistent only because of the peculiar restriction that Function-2 may be applied to each function for only one choice of A and B. (Musser's rule does not express the restriction so explicitly. In Euclid, the Predicate Calculus formulae A and B in A(E/x) => B(E/x,F(E)/y) must be included in the function definition, so the single allowed application of Function-2 to f is determined by the definition of f.)

A strongly consistent system may be achieved through the following rule. The trick is to allow assertions about expressions f(E) only after f(E) has been computed within an expression G[f(E)]. So, if f(E) is undefined, any attempt to compute G[f(E)] fails, and all partial correctness formulae about z:=G[f(E)] are true. If the expression E does not contain the variable z, the following rule may be used for reasoning about defined functions:

$$\frac{A\{P\}B}{A(E/x)[z:=G[f(E)]]B(E/x,f(E)/y)}$$

where f has been defined as

f:Function(x); local $z_1, \ldots, z_n$;
P; return(y) end

and A and B do not contain $z_1,\ldots,z_n$ free,

and z does not occur in E.

If the variable z appears in the expression E in z:=G[f(E)], then the rule above does not work, because the assertion B(E/x,f(E)/y) has a different meaning after the assignment than before the assignment. The following more complicated rule uses the substitution technique from the Assignment rule to keep the assertion B(E/x,f(E)/y) before the assignment:

Function-assignment:

$$\frac{A\{P\}B}{A(E/x)\&(B(E/x,f(E)/y)\Rightarrow C(G[f(E)]/z))\{z:=G[f(E)]\}C}$$

where f has been defined as

f:Function(x); local $z_1,\ldots,z_n$;
P; return(y) end

and A and B do not contain $z_1,\ldots,z_n$ free.

If defined functions are used in the conditions of conditionals and loops, two more rules are required:

Function-conditional:

$$\frac{A\{P\}B,\ C\&G[f(E)]\&B(f(E)/y)\{Q\}D,\ C\&\neg G[f(E)]\&B(f(E)/y)\{R\}D}{A(E/x)\&C\{\underline{If}\ G[f(E)]\ \underline{then}\ Q\ \underline{else}\ R\}D}$$

where f has been defined as

f:Function(x); local $z_1,\ldots,z_n$;
P; return(y) end

and A and B do not contain $z_1,\ldots,z_n$ free.

Function-while:

$$\frac{A\{P\}B,\ C\&G[f(E)]\{Q\}C}{A(E/x)\&C\{\underline{While}\ G[f(E)]\ \underline{do}\ Q\ \underline{end}\}C\&\neg G[f(E)]}$$

where f has been defined as

f:Function(x); local $z_1,\ldots,z_n$;
P; return(y) end

and A and B do not contain $z_1,\ldots,z_n$ free.

These three rules may be extended in a natural way to handle

more than one defined function.

The soundness of rules for function definitions is a slippery issue when function bodies fail, since the normal interpretation of the Predicate Calculus does not allow for partial functions. So, we consider a Predicate Calculus formula containing a program-defined function f to be true when it is true for all total functions f consistent with the values computed by the definition of f [7]. If the definition fails to halt, then every total function is consistent with all the computed values (there are none), so only assertions which hold for all functions, such as $\forall x\ f(x)=f(x)$, are true for f. The assertion $f(0)=0$ is only true when the definition of f actually computes the output value 0 on input 0. Under such an interpretation, Function-assignment, Function-conditional and Function-while are inferentially sound.

Since the systems containing Function-1 or Function-2 are not even strongly consistent, they cannot be sound. Notice that Function-1 is an inferentially sound rule under the total correctness interpretation. For total correctness the rules Fail and While are not sound, so alternate rules must be used for reasoning about these constructs in a total correctness logic [7,10].

The logical system containing the rules of Section 4 plus Function-assignment, Function-conditional and Function-while cannot be relatively complete according to

Cook's [8] definition, because there is no way to prove properties of f(x) unless f(x) is actually computed in the program. This system is sufficient to prove all partial correctness properties of programs which only mention values of defined functions when those values have actually been computed.

6. The Goto problem

Since the Hoare language is tailored to the description of exactly two states associated with a program execution -- the normal entry and exit states -- it is not surprising that trouble arises in considering program segments with more than one mode of entry and/or exit. Such multiple entry and exit segments occur when the Goto command is used. It is not obvious how to interpret A{P}B when P may terminate by executing Goto 1, with the label 1 occurring outside of P. The usual solution, proposed by Donahue [11], is to regard such termination as abnormal. So True{Goto 1}False is a true partial correctness formula, and, by itself, Goto 1 is indistinguishable from Fail.

Under this interpretation, the Composition-1 rule is unsound. For example, True{Goto 1}False and False{1: Null}False are true hypotheses for Composition-1, but the associated conclusion True{Goto 1; 1: Null}False is false, since Goto 1; 1: Null is equivalent to Null. No system containing Composition-1 may be inferentially sound for reasoning about programs with Gotos. In [11] Donahue

places such strong restrictions on the use of Gotos that it is syntactically impossible to have a program segment P;Q with a jump between P and Q. Composition-1 is sound for Donahue's restricted language.

Clint and Hoare [6] proposed a rule for reasoning about Gotos which may be combined with Composition-1 in a theorem sound system. To understand this rule, consider a programming language with assignment, conditional, while loops, sequencing and Gotos which may branch out of but not into the scopes of conditionals and loops. Without loss of generality, let all labels be attached to Null commands. The Null rule must be expanded to allow labelled Null commands:

Null-label: $\dfrac{}{A\{1:\ \underline{\text{Null}}\}A}$

The Clint-Hoare Goto rule is:

Goto-1: $\dfrac{B\{\underline{\text{Goto}}\ 1\}\text{False} \vdash A\{P\}B,\ B\{\underline{\text{Goto}}\ 1\}\text{False} \vdash B\{Q\}C}{A\{P;\ 1:\ \underline{\text{Null}};\ Q\}C}$

The following critique also applies to Kowaltowski's variation on the Clint-Hoare Goto rule [15]. The hypothesis

$$B\{\underline{\text{Goto}}\ 1\}\text{False} \vdash A\{P\}B$$

is intended to mean that $A\{P\}B$ has been proved using $B\{\underline{\text{Goto}}\ 1\}\text{False}$ as an assumption (similarly for $B\{\underline{\text{Goto}}\ 1\}\text{False} \vdash B\{Q\}C$).

The system of reasoning using the rules of Section 4 plus Goto-1 is theorem sound. Notice that True{Goto 1}False, although true, cannot be proved with these rules, so Composition-1 cannot be used to produce True{Goto 1; 1: Null}False. Any extension of this system in which True{Goto 1}False is provable is theorem unsound, and even inconsistent.

What about the inferential soundness of the Goto-1 rule itself? That depends on how we interpret the truth or falsehood of

$$B\{\text{Goto } 1\}\text{False} \vdash A\{P\}B.$$

If we interpret this hypothesis as true only when there is a proof of A{P}B from B{Goto 1}False in the particular system we are using, then the meaning of this rule depends on the whole system. For example, the rule would be sound within the Clint-Hoare system, but not in a system which proves True{Goto 1}False. Clarke [5] uses this weak interpretation of $\vdash$ in expressing the soundness of a rule for recursive procedures. A more robust interpretation is that

$$B\{\text{Goto } 1\}\text{False} \vdash A\{P\}B$$

is true whenever there exists an inferentially sound system in which A{P}B may be proved assuming B{Goto 1}False -- equivalently, whenever B{Goto 1}False implies A{P}B. Donahue [11] uses this stronger interpretation of in his treatment of recursive procedures. Since B{Goto 1}False is

true, the implication reduces to simply A{P}B [3]. Contrary to Donahue's Theorem 5.15 [11], the Goto-1 rule is certainly not sound in the stronger interpretation, since

False{Goto 1}False $\vdash$ True{Goto 1}False,

False{Goto 1}False $\vdash$ False{Null}False

are true hypotheses, yet the associated conclusion

True{Goto 1; 1: Null}False

is false. Arbib and Alagić noticed this difficulty independently [3].

Perhaps the insistence on inferential soundness and the most liberal possible interpretation of $\vdash$ seems too picky. After all, it seems that we only need to be careful about Gotos, which are well-known to be dangerous beasts, and avoid introducing axioms like True{Goto 1}False. Unfortunately, the rule Goto-1 may yield false conclusions in the presence of added rules or axioms which do not appear to have anything to do with Gotos. For example, consider the sound and intuitively attractive rule:

$$\text{Zero:}\quad \frac{}{\text{True}\{P;\ x:=0\}x=0}$$

In the presence of the rule Zero, Goto-1 derives incorrect formulae. For example:

1)	1)	x=0{Goto 1}False	Assumption
	2)	True{x:=1; Goto 1; x:=0}x=0	Zero
2)	1)	x=0{Goto 1}False	Assumption
	2)	x=0{1: Null}x=0	Null-label
3)		True{x:=1; Goto 1; x:=0; 1: Null}x=0	Goto-1, 1),2)

The correct theorem True{x:=1; Goto 1; x:=0; 1: Null}x≠0 is also provable, so the system containing Goto-1 and Zero is not strongly consistent.

How may we reason correctly about Gotos? One way is to return to the Floyd [12] style of proof, in which a proof follows the control flow of a program. Constable and O'Donnell [7] have explored this idea. Manna and Waldinger's intermittent assertions [17] also handle Gotos easily. Even if we insist on using the Hoare language, we may still have a sound system for reasoning about Gotos. First, Composition-1 must be replaced by:

$$\text{Composition-2:}\quad \frac{A\{P\}B,\ B\{Q\}C}{A\{P;Q\}C}$$

where there are no Goto branches from P to Q or Q to P.

To understand the rest of the rules, notice that A{P; Fail; 1: Null}B says that if A is true initially, and P terminates by executing Goto 1, then B is true of the final state. Alagić and Arbib [1,3] express the same idea in the more convenient special notation {A}P{1: B}.

Goto-2:

$$\frac{}{A\{\underline{\text{Goto}}\ 1\}B}$$

Goto-label-same:

$$\frac{}{A\{\underline{\text{Goto}}\ 1;\ P;\ 1:\ \underline{\text{Null}}\}A}$$

Goto-label-other:

$$\frac{A\{P;\ \underline{\text{Fail}};\ 1:\ \underline{\text{Null}}\}B}{A\{P;\ m:\ \underline{\text{Null}};\ \underline{\text{Fail}};\ 1:\ \underline{\text{Null}}\}B}$$

where 1 and m are different labels.

Goto-composition:

$$\frac{A\{P;\ \underline{\text{Fail}};\ 1:\ \underline{\text{Null}}\}C,\ A\{P\}B,\ B\{Q;\ \underline{\text{Fail}};\ 1:\ \underline{\text{Null}}\}C}{A\{P;\ Q;\ \underline{\text{Fail}};\ 1:\ \underline{\text{Null}}\}C}$$

where there are no
Goto branches from
P to Q or Q to P.

Goto-conditional:

$$\frac{A\&B\{P;\ \underline{\text{Fail}};\ 1:\ \underline{\text{Null}}\}C,\ A\&\neg B\{Q;\ \underline{\text{Fail}};\ 1:\ \underline{\text{Null}}\}C}{A\{\underline{\text{If}}\ B\ \underline{\text{then}}\ P\ \underline{\text{else}}\ Q\ \underline{\text{end}};\ \underline{\text{Fail}};\ 1:\ \underline{\text{Null}}\}C}$$

Goto-while:

$$\frac{A\&B\{P\}A,\ A\&B\{P;\underline{\text{Fail}};\ 1:\ \underline{\text{Null}}\}C}{A\{\underline{\text{While}}\ B\ \underline{\text{Do}}\ P\ \underline{\text{end}};\ \underline{\text{Fail}};\ 1:\ \underline{\text{Null}}\}C}$$

Combination:

$$\frac{A\{P\}B,\ A\{P;\ \underline{\text{Fail}};\ 1:\ \underline{\text{Null}}\}B}{A\{P;\ 1:\ \underline{\text{Null}}\}B}$$

Alagić and Arbib [1] present the Goto-2, Goto-label and Goto-while rules in a somewhat more powerful notation. They also give the Goto-composition and Composition-1 rules combined into one rule, neglecting to state the restriction that there are no jumps between P and Q. Without such a restriction, the rule becomes unsound. (In private correspondence, Arbib indicates that the rule was only intended to apply to a restricted form of statement, called an L-statement. Arbib and Alagic's rule is sound for L-statements. The restriction is not given explicitly in the statement of the rule.) Combination is strengthened to include one application of Composition-1. Goto-conditional is omitted in [1].

The system consisting of the rules Null, Fail, Assignment, Conditional and While from Section 4, along with Null-label Composition-2, Goto-2, Goto-label, Goto-composition, Goto-conditional, Goto-while and Combination above, is inferentially sound. Cook's techniques for proving relative completeness [8] may be used to show that this system is sufficiently powerful to derive all true partial correctness formulae for our simple programming language with Gotos.

7. Summary and Conclusions

I have argued that a logical system is only correct when it is inferentially sound, so that every intermediate step in a proof, as well as the final result, is true

according to some intuitively meaningful notion of truth. Weaker correctness criteria, such as theorem soundness, which guarantees the truth of final results, but not intermediate steps, are unacceptable because they allow intuitively false reasoning which leads by formal tricks to true results. A logical system which is theorem sound but not inferentially sound is very dangerous because the addition of true axioms may introduce an inconsistency.

Rules proposed for reasoning about defined functions and Gotos in the Hoare style have not always met the standard of inferential soundness. Inferentially sound rules are not hard to find, but they are unsatisfyingly inelegant. The problem seems to be that partial correctness reasoning in the Hoare language is very natural for programs with only conditionals and loops for control structures, but not for programs with defined functions and/or Gotos. Defined functions tangle partial correctness and termination together to such an extent that it is no longer convenient to separate them. Since it is essential to prove termination anyway, we should use total correctness logics for reasoning about function definitions. Goto commands destroy the Hoare-style analysis of programs by structural induction, since the semicolon does not really indicate composition in the presence of Gotos, as it does in their absence. Goto commands are handled very naturally in the Floyd style of reasoning.

Bibliography

1. Alagić, S. and Arbib, M.A. The Design of Well-Structured and Correct Programs. Springer-Verlag, New York, (1978).

2. Apt, K.R. A sound and complete Hoare-like system for a fragment of Pascal. Report IW/78, Mathematisch Centrum, Afdeling Informatica, Amsterdam, (1978).

3. Arbib, M.A. and Alagić, S. Proof rules for gotos. Acta Informatica 11.2, (1979), 139-148.

4. Ashcroft, E.A., Clint M. and Hoare, C.A.R. Remarks on program proving: jumps and functions, Acta Informatica 6:3 (1976), 317.

5. Clarke, E.M. Programming language constructs for which it is impossible to obtain good Hoare-like axiom systems, JACM 26:1, (1979), 129-147.

6. Clint, M. and Hoare, C.A.R. Program proving: jumps and functions Acta Informatica 1:3 (1972), 214-224.

7. Constable, R. and O'Donnell, M. A Programming Logic. Winthrop, Cambridge Massachusetts, (1978).

8. Cook, S.A. Soundness and completeness of an axiom system for program verification. SIAM Journal on Computing 7:1 (1978), 70-90.

9. de Bruin, A. Goto statements: semantics and deductions systems. Preprint, Mathematisch Centrum, Amsterdam, (1978).

10. Dijkstra, E.W. Guarded commands, nondeterminacy and formal derivation of programs. CACM 18:8, (1975), 453-457.

11. Donahue, J.E. Complementary Definitions of Programming Language Semantics. Lecture notes in Computer Science 42, Springer-Verlag, New York, (1976).

12. Floyd, R.W. Assigning meanings to programs. Proceedings of symposia in applied mathematics, 19, American Mathematical Society, Providence, (1967).

13. Hoare, C.A.R. An axiomatic basis for computer programming. CACM 12:10, (1969), 576-580.

14. Hoare, C.A.R. and Wirth, N. An axiomatic definition of the programming language PASCAL. Acta Informatica 2:4, (1973), 335-355.

15. Kowaltowski, T. Axiomatic approach to side effects and general jumps. Acta Informatica 7:4, (1977), 357-360.

16. London, R.L., Guttag,J.V., Horning, J.J., Lampson, B.W., Mitchell, J.G., and Popek, G.J. Proof rules for the programming language Euclid. Acta Informatica 10:1, (1978), 1-26.

17. Manna, Z. and Waldinger, R. Is "sometime" sometimes better than "always"? Second international conference on Software Engineering, (1976).

18. Mendelson, E. Introduction to Mathematical Logic. 2nd edition, Van Nostrand, N.Y., (1976).

19. Musser, D. A proof rule for functions. USC information sciences institute technical report ISI/RR-77-62, (1977).

20. Olderog, E. Sound and complete Hoare-like calculi based on copy rules. Technical report 7905, Christian-Albrechts Universität, Kiel, (1979).

21. Russell, B. Letter to G. Frege, June 16, 1902. From Frege to Godel: A Source Book in Mathematical Logic, 1879-1931. J. van Heijenoort (Ed.), Harvard University Press, Cambridge, (1967), 124-125.

22. Scott, D. and Strachey, C. Towards a mathematical semantics for computer languages. Computers and Automata. J. Fox (Ed.), Wiley, New York, (1972), 19-46.