stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 65/76 NOVEMBER

L. AMMERAAL

ON FORWARD AND BACKWARD PROOF RULES FOR PROGRAM VERIFICATION

Prepublication

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O), by the Municipality of Amsterdam, by the University of Amsterdam, by the Free University at Amsterdam, and by industries.

AMS(MOS) subject classification scheme (1970): 68A05

ACM-Computing Reviews-category: 5.24

On forward and backward proof rules for program verification*)

by

L. Ammeraal

ABSTRACT

The notions of "strongest verifiable consequent" and "weakest precondition", introduced by Floyd and Dijkstra, respectively, suggest a partition of proof rules into forward and backward rules. New notations for such rules are proposed and motivated. The paper advocates the "total correctness" point of view. Forward and backward rules are specified for assignment statements, conditional statements and while statements. Proof rules may be related to one another; some of such relationships are presented with reference to set theory.

KEY WORDS & PHRASES: *program verification, program correctness.*

*) This report will be submitted for publication elsewhere.

1. INTRODUCTION

To verify the correctness of a program it is helpful to insert valid assertions as comments in the program text. Naur [7] describes how these comments can be regarded as "general snapshots" and suggests to use them in correctness proofs. Floyd [1] introduces the notion of the "strongest verifiable consequent" for a given antecedent and a given program statement. Indeed, we are not interested in just some assertion that holds upon the completion of a statement, but only in the strongest one. In contrast to the ("forward") derivation of a postcondition from a given precondition, there are "backward" rules, introduced by Hoare [2], to derive the weakest precondition which ensures that a given postcondition is satisfied upon completion of a statement. Manna [4] deals with another classification of proof rules. He distinguishes between total and partial correctness (also termed strong vs. weak verification). Proof rules are strong if preconditions for statements ensure termination of the statements. If termination is not required, proof rules are called weak. Like Hoare, Dijkstra [3] discusses backward rules and calls them "predicate transformers". In this context he briefly refers to the concept of a "state space" which has also been used in papers on semantics by Scott, Strachey, De Bakker and others. How state-space transformations are a basis for strong verification rules has been described in [10].
There is no reason to restrict our considerations to backward proof rules. Their dominance over forward rules in most papers on program verification is probably due to two circumstances. First, the actual application of Hoare's (backward) axiom for the assignment statement involves less labour than Floyd's (forward) axiom. Secondly, as we shall see later on, the former behaves better than the latter with respect to the conjunction of conditions. As far as backward rules are concerned, our subject matter is related to a paper of Basu and Yeh [9].

2. FORWARD AND BACKWARD PROOF RULES

We say that a (syntactically correct) program statement *terminates* if it can be executed in a finite amount of time and yields a well-defined result. We denote assertions about program variables by P and Q, and statements by S. Instead of "immediately before the execution of S" we simply say "before S";

a similar meaning has "after S". In this terminology Hoare's notation

$$P\{S\}Q$$

means:

"If P is true before S and if S terminates,
then Q is true after S".

Proof rules of this type are called *weak*. They allow, for example, cases like

$$|x| < 1 \quad \{x := 1/x\} \quad |x| > 1. \tag{1}$$

It is, however, a good convention in mathematics that formulas involving division are complemented by conditions which ensure that every denominator is non-zero. We therefore insist that $|x| < 1$ in (1) be replaced by $0 < |x| < 1$. Considerations like this lead to *strong* proof rules which we write as

$$\{P\}S\{Q\}.$$

Their meaning is:

"If P is true before S, then S terminates and Q
is true after S".

Unfortunately, even strong rules are too tolerant for our purposes. We do not want the rule

$$\{0 < |x| < 1\} \quad x := 1/x \quad \{|x| > \tfrac{1}{2}\}.$$

Here we feel the need to require that the postcondition be as strong as possible. There is another unsatisfactory aspect. In

$$\{x = 3\} \quad x := x * x \quad \{x = 9\} \tag{2}$$

the postcondition $x = 9$ is as strong as possible and we are completely content with (2) as long as we regard it as the solution of

$$\{x = 3\} \quad x := x * x \quad \{\ ?\ \}. \tag{3}$$

However, (2) is dubious if it stems from

$$\{ ? \} \quad x := x * x \quad \{x = 9\}, \tag{4}$$

since in (4) we would replace the question mark by $|x| = 3$ rather than $x = 3$.

Cases like (3) and (4) need different kinds of proof rules. To solve (3) we need a forward rule, which yields the strongest postcondition. For (4) we need a *backward* rule yielding the weakest precondition. Obviously, the notations $P\{S\}Q$ and $\{P\}S\{Q\}$ are too symmetric to serve both purposes. We therefore introduce two new notations, using the previously defined form $\{P\}S\{Q\}$:

a. $\{P\}S[Q]$ is a forward rule.
   Its meaning is given by
   1. $\{P\}S\{Q\}$, and
   2. If $\{P\}S\{Q'\}$ then $Q \Rightarrow Q'$.

b. $[P]S\{Q\}$ is a backward rule.
   Its meaning is given by
   1. $\{P\}S\{Q\}$, and
   2. If $\{P'\}S\{Q\}$ then $P' \Rightarrow P$.

Examples of (valid) rules in this notation are:

$$\{|x| > 3\} \quad x := x * x \quad [x > 9] \quad ,$$

$$\{x > 3\} \quad x := x * x \quad [x > 9] \quad ,$$

$$[|x| > 3] \quad x := x * x \quad \{x > 9\} \quad ,$$

$$\{0 < |x| < 1\} \quad x := 1/x \quad [|x| > 1].$$

The following rules are invalid:

$$[x > 3] \quad x := x * x \quad \{x > 9\} \quad ,$$

$$\{|x| < 1\} \quad x := 1/x \quad [|x| > 1],$$

$$\{0 < |x| < 1\} \quad x := 1/x \quad [|x| > \tfrac{1}{2}].$$

Inherent preconditions

Associated with every statement S is a weakest precondition which ensures that S terminates. We call this condition the *inherent precondition* P_S of S. Formally P_S can be defined by

$$[P_S]S\{\underline{true}\}$$

(We identify equivalent conditions such as $x > 0$, $0 < x$ and $x^3 > 0$; similarly, true is identified with e.g. $1 + 1 = 2$ and $(x+1)(x-1) = x^2 - 1$.)

Examples of inherent preconditions are given within the square brackets in

$$[x \neq 0]\ x := 1/x\ \{\underline{true}\},$$

$$[i = 0,1,2,...]\ \underline{while}\ i \neq 0\ \underline{do}\ i := i-1\ \underline{od}\ \{\underline{true}\}.$$

There are many statements whose inherent preconditions are true. An example of such a statement is

$$\underline{while}\ i > 0\ \underline{do}\ i := i-1\ \underline{od}.$$

We call a precondition P *strong enough* (for S) if $P \Rightarrow P_S$. It follows from our definitions that $\{P\}S[Q]$ and $[P]S\{Q\}$ can be valid only if P is strong enough for S.

3. PROOF RULES FOR SPECIFIC STATEMENT TYPES

We shall now deal with forward and backward proof rules for assignment statements, conditional statements, and while statements. Notice that the latter two statement types may contain other statements and may therefore be arbitrarily complex. We shall restrict ourselves to unsubscripted variables. In a more semantical context, De Bakker [5] has shown how to cope with subscripted variables.

a1. Assignment statements (forward)

If P is strong enough then

$$\{P(x)\}\ x := \varphi(x)\ [\exists x_0 : P(x_0) \wedge x = \varphi(x_0)].$$

An example is:

$$\{x+y>0\}\ x := x-y\ [\exists x_0 : x_0+y>0 \wedge x = x_0-y].$$

By eliminating x_0 the derived postcondition in this example is simplified to $x+2y > 0$.
This forward rule was introduced by Floyd [1].

a2. Assignment statements (backward)

$$[Q(\varphi(x))]\ x := \varphi(x)\ \{Q(x)\}.$$

This backward rule was introduced by Hoare [2] and is often applied in textbooks on programming.

b1. Conditional statements (forward)

If $\{P \wedge B\}S_1[Q_1]$ and $\{P \wedge \neg B\}S_2[Q_2]$, then $\{P\}$ if B then S_1 else S_2 fi $[Q_1 \vee Q_2]$.
It is curious that this useful forward rule is missing in most papers on program verification. As an example, we infer from

$$\{x+y>0 \wedge x>0\}\ x := x-1\ [x+y > -1 \wedge x > -1]$$

and

$$\{x+y>0 \wedge x\leq 0\}\ y := y+1\ [x+y > 1 \wedge x \leq 0]$$

that

$$\{x+y>0\}\ \text{if } x>0 \text{ then } x := x-1 \text{ else } y := y+1 \text{ fi}$$
$$[(x+y > -1 \wedge x > -1) \vee (x+y > 1 \wedge x \leq 0)]$$

holds.

b2. Conditional statements (backward)

If $[P_1]S_1\{Q\}$ and $[P_2]S_2\{Q\}$, then $[(P_1 \wedge B) \vee (P_2 \wedge \neg B)]$ if B then S_1 else S_2 fi $\{Q\}$.
This most useful backward rule is well-known. Dijkstra [3] calls it the "Axiom of Binary Selection".

c1. While statements (forward)

If P is strong enough (for the while statement), then

$$\{P\}\ \text{while } B \text{ do } S \text{ od } [\neg B \wedge (U_0 \vee U_1 \vee U_2 \vee \ldots)],$$

where U_k are given by:

$$U_0 \equiv P, \qquad \{B \wedge U_{k-1}\}S[U_k] \qquad (k = 1,2,\ldots).$$

c2. While statements (backward)

$$[V_0 \vee V_1 \vee V_2 \vee \ldots] \text{ while } B \text{ do } S \text{ od } \{Q\},$$

where V_k are given by:

$$V_0 \equiv (\neg B \wedge Q), \quad V_k \equiv (B \wedge W_k) \text{ with } [W_k]S\{V_{k-1}\} \qquad (k = 1,2,\ldots).$$

Remark on while statements

Due to the infinite disjunctions occurring in the rules for while statements, these rules seem less attractive than the weaker but simpler "rule of iteration" introduced by Hoare [2]. On the other hand, a user of Hoare's rule must himself find an appropriate "loop invariant", which is not always easy. In [10] some simple applications of the rules given here are shown.

4. PROPERTIES OF PROOF RULES

A user of proof rules will sooner or later get interested in certain relationships between various rules. He might, for instance, wonder in what circumstances forward and backward rules are mutually inverse, or he might ask himself whether or not

$$\{P_1 \wedge P_2\}S[Q_1 \wedge Q_2]$$

is valid if it is given that $\{P_1\}S[Q_1]$ and $\{P_2\}S[Q_2]$ are valid. We shall therefore briefly deal with a number of those relationships. Their justification is based on simple and well-known set-theoretical facts, such as

$$f(X \cap Y) \subset f(X) \cap f(Y).$$

All proof rules under discussion can be formulated in terms of transformations in state spaces, which was the approach taken in [10]. Every condition P on program variables corresponds to a subset X_P of a state space; X_P is the set of exactly those states that satisfy P. Then $\{P\}S[Q]$ and $[P]S\{Q\}$ correspond, respectively, to

$$f(X_P) = X_Q \quad \text{and} \quad f^{-1}(X_Q) = X_P.$$

For some program statements the associated functions f are one-to-one. These program statements are characterized by the following definition. We call a

statement S *injective* if P' ≡ P" whenever both {P'}S[Q] and {P"}S[Q].

We mention the following six properties:

(i) If S is injective, then
{P}S[Q] implies [P]S{Q}.

(ii) If $\{P_S\}S[\underline{\text{true}}]$, then
[P]S{Q} implies {P}S[Q].

(iii) If $\{P_1\}S[Q_1]$ and $\{P_2\}S[Q_2]$, then
a. $P_1 \Rightarrow P_2$ implies $Q_1 \Rightarrow Q_2$;
b. $\{P_1 \vee P_2\}S[Q_1 \vee Q_2]$;
c. If S is injective, then
$\{P_1 \wedge P_2\}S[Q_1 \wedge Q_2]$.

(iv) If $[P_1]S\{Q_1\}$ and $[P_2]S\{Q_2\}$, then
a. $Q_1 \Rightarrow Q_2$ implies $P_1 \Rightarrow P_2$;
b. $[P_1 \vee P_2]S\{Q_1 \vee Q_2\}$;
c. $[P_1 \wedge P_2]S\{Q_1 \wedge Q_2\}$.

(Comparing (iii)c and (iv)c we see that backward rules behave better than forward ones with respect to conjunction.)

(v) $\{P\}S[\underline{\text{false}}]$ if and only if $P \equiv \underline{\text{false}}$.

(Here our convention that P must be "strong enough" for S is essential.)

(vi) [P]S{Q} implies $[P_S \wedge \neg P]S\{\neg Q\}$.

REFERENCES

[1] Floyd, R.W., *Assigning Meanings to Programs*, Proc. Symp., Appl. Math. 19, American Math. Soc. (1967) 19-32.

[2] Hoare, C.A.R., *An Axiomatic Basis for Computer Programming*, CACM 12 (1969) 576-580.

[3] Dijkstra, E.W., *A simple Axiomatic Basis for Programming Language Constructs*, Proc. Kon. Ned. Akad., Ser. A, 77 (or Indagationes Math., 36), (1974) 1-15).

[4] Manna, Z., *Mathematical Theory of Computation*, McGraw-Hill (1974).

[5] De Bakker, J.W., *Correctness proofs for assignment statements*, Report IW 55/76, Mathematisch Centrum, Amsterdam (1976).

[6] Igarashi, S., R.L. London, D.C. Luckham, *Automatic program verification I: A logical basis and its implementation*, Acta Informatica 4 (1975) 145-182.

[7] Naur, P., *An Experiment on Program Development*, BIT 12 (1972) 347-365.

[8] Mills, H.D., *The New Math of Computer Programming*, CACM 18, 43-48.

[9] Basu, S.K., R.T. Yeh, *Strong Verification of Programs*, IEEE Transactions on Software Engineering, 1 (1975) 339-346.

[10] Ammeraal, L., *How program statements transform predicates*, Informatik-Fachberichte 5 (1976) 109-120, Springer-Verlag.