Repetitions, known or unknown?

Wim H. Hesselink

Department of Computing Science, Rijksuniversiteit Groningen, P.O. Box 800, 9700 AV Groningen, Netherlands

Communicated by J. Misra
Received 4 January 1991

Keywords: Program correctness, program derivation, proof rules, refinement, repetition

Introduction

The purpose of this note is to discuss the meaning of repetitions in an imperative programming language with nondeterminacy, and to present some results and examples that are not widely known. We begin with a new form for well-known material.

1. The proof rule for total correctness

Imperative languages are characterised by commands with effect on the state of memory. This effect is described by means of a so-called Hoare triple, that is a judgement of the form

\[ \{ P \} \ S \ \{ S \} \]

We take this triple to express that if command \( S \) is given in a state that satisfies \( P \), then execution of \( S \) terminates in a state that satisfies \( Q \). So, for us, a Hoare triple indicates total correctness.

We consider repetitions of the form

\[ L: \ \text{while} \ B \ \text{do} \ S \ \text{od.} \]

We propose to use the following correctness rule for repetition \( L \):

\[ (0) \quad \text{Rule.} \quad \text{If} \ J \ \text{is a predicate and} \ uf \ \text{is an integer-valued state function with, for all integer values} \ n, \]

\[ \{ J \land B \land uf = n \} \ S \ \{ J \land n \geq 0 \land uf < n \} , \]

\[ \text{then it holds} \ \{ J \} \ L \ \{ J \land \neg B \}. \]

Notice that we included \( n \geq 0 \) in the postcondition, so that the usual bounding condition \([J \land B \Rightarrow uf \geq 0]\) can be omitted. It is a kind of coding trick, which enables us to present a repetition together with the proof as a single annotated program. Compare [1, Chapter 6] and [3, Theorem (11.6)].

Example. Let us illustrate rule (0) by means of an easy example that is not completely trivial. Let \( f \) be the function from integers to integers given by

\[ f(x) = 0 \quad \text{if} \ x \leq 0 , \]

\[ f(x) = x + f(x \ \text{div} \ 2) \quad \text{if} \ x > 0 . \]
Clearly, a functional implementation in a kind of Pascal is

\[
\text{function } f(x: \text{integer}): \text{integer}; \\
\quad \text{if } x \leq 0 \text{ then } f := 0 \\
\quad \text{else } f := x + f(x \div 2) \text{ fi.}
\]

Let us now try and replace the functional implementation by a repetition. So, using program variables \(x\) and \(y\), we want to implement

\[
\{ f(x) = Z \} \quad ? \quad \{ y = Z \}.
\]

We use a repetition \(L\), preceded by an initialisation of \(y\).

\[
\{ f(x) = Z \} \\
y := 0 \\
\{ f(x) = Z \land y = 0, \text{ and hence } J: y + f(x) = Z \} ; \\
L: \quad \text{while } x > 0 \text{ do } S \text{ od} \\
\{ J \land x \leq 0, \text{ and hence } y = Z \}.
\]

We use proof rule (0) with \(uf = x\). Therefore, it suffices to find a body \(S\) that satisfies for all integer values \(n\):

\[
\{ J \land B \land x = n \} \quad S \quad \{ J \land n \geq 0 \land x < n \}.
\]

Well, we expand the definitions and invent two assignments:

\[
\{ J \land B \land x = n: y + f(x) = Z \land n = x > 0 \\
\quad \text{and hence } y + x + f(x \div 2) = Z \land n = x > 0 \} \\
y := y + x \\
\{ y + f(x \div 2) = Z \land n = x > 0 \} ; \\
x := x \div 2 \\
\{ y + f(x) = Z \land n > 0 \land x < n \text{ and hence } J \land n \geq 0 \land x < n \}.
\]

In this way, we obtain that \(L\) is correct if we take

\[
S: \quad y := y + x; \quad x := x \div 2.
\]

(End of example.)

2. Nondeterminacy

In program development, it is important to postpone design decisions as long as possible. We therefore allow commands in which some choices are not specified. This is called nondeterminacy. As an example, we show the nondeterminate phase in the design of binary search (where the array need not yet be ordered).

\[
\text{const } m, w \in \text{integer} \quad \{ m > 0 \}; \\
\text{array[0..m] of integer} \quad \{ a[0] \leq w < a[m] \}; \\
\text{var } i: \text{integer}; \\
to be established \quad \{ a[i] \leq w < a[i + 1] \}.
\]
We can easily establish the postcondition if \( i + 1 \) is replaced by an auxiliary variable \( j \). We choose variant function \( \nu f = j - i \). In this way, we arrive at the implementation:

\[
i' = 0; \quad j' = m;
\]

\[
\{ J: 0 \leq i < j \leq m \land a[i] \leq w < a[j] \}
\]

\[
\text{while } j \neq i + 1 \text{ do}
\]

\[
\{ J \land B \land j - i = n \text{ for a new constant } n \}
\]

choose \( k \) with \( i < k < j \):

\[
\{ 0 \leq i < k < j \leq m \land a[i] \leq w < a[j] \land n = j - i > 0 \}
\]

\[
\text{if } w < a[i] \text{ then } j := k \text{ else } i' := k \text{ fi}
\]

\[
\{ J \land n \geq 0 \land j - i < n \} \quad \text{od}
\]

\[
\{ J \land j = i + 1 \text{ and hence } a[i] \leq w < a[i + 1] \}.
\]

In the next phase, the nondeterminacy can be resolved.

3. Weakest preconditions

Given a command \( S \), there may be many pairs \( P \) and \( Q \) for which \( \{ P \} S \{ Q \} \) holds. For example, for an integer program variable \( x \), we have

\[
\{ P: x \geq 7 \} \quad x := x \cdot x + 1 \quad \{ Q: x \geq 50 \}.
\]

Postcondition \( Q \) can be strengthened to

\[
Q': (\exists t:: t \geq 7 \land x = t^2 + 1).
\]

This is the strongest postcondition. It is more convenient to work with the weakest precondition, which is (for postcondition \( Q \))

\[
P': x^2 + 1 \geq 50.
\]

In general, the weakest precondition of command \( S \) with respect to postcondition \( Q \) (denoted \( \text{wp} . S . Q \)) is defined by

\[
\text{(1) } \text{wp} . S . Q = \{ P \} S \{ Q \} \text{ for all predicates } P.
\]

The square brackets denote universal quantification over all program variables:

\[
[x \geq 7 \Rightarrow x^2 + 1 \geq 50] = (\forall x:: x \geq 7 \Rightarrow x^2 + 1 \geq 50).
\]

The most important nondeterminate constructor is the bar “\( [] \)”. If \( S \) and \( T \) are commands, command \( S [] T \) specifies to execute either \( S \) or \( T \), but it does not specify the choice. Therefore, we have

\[
\text{wp} . (S [] T) . Q
\]

= “definition (1)”

\[
\{ P \} S [] T \{ Q \}
\]

= “you do not know the choice”

\[
\{ \{ P \} S \{ Q \} \} \land \{ \{ P \} T \{ Q \} \}
\]

= “definition (1)”

\[
\text{wp} . S . Q \land \text{wp} . T . Q
\]

= “calculation”

\[
\text{wp} . S . Q \land \text{wp} . T . Q
\]

53
This proves that

\[ wp.(S[T]T).Q = wp.S.Q \land wp.T.Q. \]

Actually, this formula is often taken as a definition.

By a similar argument, one can justify the definition of the weakest precondition of a composition of commands \( S \) and \( T \), which is given by

\[ wp.(S;T).Q = wp.S.(wp.T.Q). \]

For a repetition \( L: \text{while } B \text{ do } S \text{ od} \), the weakest precondition \( wp.L.Q \) is defined as the strongest predicate \( X \) with

\[ [(B \lor Q) \land (\neg B \lor wp.S.X) \equiv X], \]

see [2, Chapter 9]. By [2, (8, 25)], \( wp.L.Q \) is also the strongest solution of

\[ X: [(B \lor Q) \land (\neg B \lor wp.S.X) \Rightarrow X]. \]

4. The proof rule for necessity of preconditions

In correctness proofs we are only interested in the question whether a given predicate implies the weakest precondition. In program transformation or in proofs of incorrectness, we can also be interested in the necessity of certain preconditions. Necessity of preconditions is usually shown by means of scenarios. Since scenarios require careful operational reasoning, we prefer a formal instrument like the following necessity rule, which seems to be a new result.

\[ \text{Theorem. For every predicate } P, \text{ repetition } L: \text{while } B \text{ do } S \text{ od satisfies} \]
\[ [B \land wp.S.P \Rightarrow P] \Rightarrow [wp.L.(B \lor P) \Rightarrow P]. \]

Proof. Assume that

\[ [B \land wp.S.P \Rightarrow P]. \]

Since \( wp.L.Q \) is the strongest solution of (4), it suffices to prove that \( P \) is a solution of equation (4) with \( Q = (B \lor P) \). This is proved in

\[ \begin{align*}
(B \lor (B \lor P)) \land (\neg B \lor wp.S.P) \\
\Rightarrow \quad \{(6) \text{ yields } [wp.S.P \Rightarrow \neg B \lor P]\} \\
(B \lor (B \lor P)) \land (\neg B \lor (\neg B \lor P)) \\
= \quad \{\text{idempotency and distributivity}\} \\
(B \land \neg B) \lor P \\
= \quad \{\text{calculus}\} \\
P.
\end{align*} \]

(End of proof.)
Example. "Jumping over the threshold." As an application of the necessity rule we consider

\[ L: \quad \text{while } x > 0 \text{ do } x := x - 2 \text{[} x := x - 3 \text{ od}. \]

In this toy example we wish to give a formal proof of

\[ \text{wp}.L.(x < 0) \Rightarrow x \leq 1 \land x \neq 0. \]

We take \( B: x > 0 \) and \( P: x \leq 1 \land x \neq 0 \), so that \((B \lor P) = (x \neq 0)\). Now we calculate

\[ \text{wp}.L.(x < 0) \Rightarrow P \]

\[ \Rightarrow \{ \text{weakening of antecedent; } \text{wp}.L \text{ is monotone} \} \]

\[ \text{wp}.L.(x \neq 0) \Rightarrow P \]

\[ \Rightarrow \{ \text{proof rule (5)} \} \]

\[ x > 0 \land \text{wp}.S.(x \leq 1 \land x \neq 0) \Rightarrow P \]

\[ \Rightarrow \{ \text{wp}.S \text{ applied with } S \text{ body of loop } L; \text{ rule (2)} \} \]

\[ x > 0 \land x - 2 \leq 1 \land x - 2 \neq 0 \land x - 3 \leq 1 \land x - 3 \neq 0 \Rightarrow P \]

\[ \Rightarrow \{ \text{calculus} \} \]

\[ x > 0 \land x \leq 3 \land x \neq 2 \land x \leq 4 \land x \neq 3 \Rightarrow P \]

\[ \Rightarrow \{ \text{calculus} \} \]

\[ x = 1 \Rightarrow P \]

\[ \Rightarrow \{ \text{calculus} \} \]

true.

(End of example.)

5. Refinement and equivalence of commands

We say that a command \( S \) is refined by a command \( T \) if \( T \) satisfies every Hoare triple specification of \( S \), notation \( S \leq T \). We have

\[ (7) \]

\[ S \leq T \]

\[ = \{ \text{above definition} \} \]

\[ (\forall P, Q:: \{ P \} S \{ Q \} \Rightarrow \{ P \} T \{ Q \}) \]

\[ = \{ \text{definition (1)} \} \]

\[ (\forall P, Q:: [P \Rightarrow \text{wp}.S.Q] \Rightarrow [P \Rightarrow \text{wp}.T.Q]) \]

\[ = \{ \text{predicate calculus} \} \]

\[ (\forall Q:: [\text{wp}.S.Q \Rightarrow \text{wp}.T.Q]). \]

For the present discussion, let us define semantic equivalence by

\[ S \equiv T \equiv \text{wp}.S = \text{wp}.T. \]

It follows that \( S \equiv T \) is equivalent to \( S \leq T \land T \leq S \).
6. Commutation

We introduced refinements to discuss the following commutation theorem:

\[(8) \text{ Theorem. Let } B \text{ be a predicate and } S \text{ and } C \text{ commands with} \]
\[S;C \preceq C;S \land [B \Rightarrow wp.C.B] \land [\neg B \Rightarrow wp.C.(\neg B)].\]

Then repetition \(L: \text{while } B \text{ do } S \text{ od} \) satisfies \(L;C \preceq C;L\).

\textbf{Proof.} The proof obligation is
\[[wp.(L;C).Q \Rightarrow wp.(C;L).Q] \text{ for every predicate } Q.\]

By the definitions in Section 3, predicate \(wp.(L;C).Q\) is the strongest solution \(X\) of equation (4) with \(Q := wp.C.Q\). Therefore, it suffices to prove that \(wp.(C;L).Q\) also solves that equation, in other words that
\[[((B \lor wp.C.Q) \land (\neg B \lor wp.(S;C;L).Q)) \Rightarrow wp.(C;L).Q].\]

This is proved in
\[wp.(C;L).Q\]
\[= \{\text{composition and (3)}\}\]
\[wp.C.((B \lor Q) \land (\neg B \lor wp.(S;L).Q))\]
\[= \{\text{wp.C is conjunctive}\}\]
\[wp.C.(B \lor Q) \land wp.C.(\neg B \lor wp.(S;L).Q)\]
\[= \{\text{wp.C is monotone}\}\]
\[(wp.C.B \lor wp.C.Q) \land (wp.C.(\neg B) \lor wp.(C;S;L).Q)\]
\[= \{\text{the three assumptions and (7)}\}\]
\[(B \lor wp.C.Q) \land (\neg B \lor wp.(S;C;L).Q).\]

(End of proof.)

\textbf{Remark.} The theorem is less innocent than it may look. Indeed, if the refinements are reversed, it looks just as innocent to me, but it is no longer true (if \(C\) may have infinite nondeterminacy)! For generalisations of the theorem, we refer to [4, Chapter 11] and [5, Chapter 7].

\textbf{Example.} “A loop that does not inherit commutation”. We use one integer program variable \(x\) and consider the commands:

\(L:\) \text{ while } B: x \neq 0 \text{ do } \]
\(S:\) \text{ if } x > 0 \text{ then } x := x - 1 \]
\(\quad \text{ else } x := 0\fi \text{ od,} \]
\(C:\) \text{ if } x < 0 \text{ then choose } x \text{ with } x > 0 \text{ else } \text{skip} \fi.

In this case, it is easy to verify
\[[x \neq 0 \Rightarrow wp.C.(x \neq 0)] \text{ and } [x = 0 \Rightarrow wp.C.(x = 0)].\]
Moreover, both $S;C$ and $C;S$ turn out to be equivalent to
\[
\begin{align*}
&\text{if } x > 0 \rightarrow x := x - 1 \\
&\quad \cup \quad x = 0 \rightarrow \text{skip} \\
&\quad \cup \quad x < 0 \rightarrow \text{choose } x \text{ with } x \geq 0 \phi.
\end{align*}
\]

Now Theorem (8) implies that $L;C \leq C;L$.

By operational means (or using the proof rules), it is easy to see that
\[
wp.(L).true = (x \geq 0) \quad \text{and} \quad wp.(C).true = (x \geq 0).
\]

It follows that
\[
wp.(C;L).true = true \quad \text{and} \quad wp.(L;C).true = (x \geq 0).
\]

Since $[x \geq 0 \Rightarrow true]$, this confirms $L;C \leq C;L$, but of course it does not prove it. On the other hand, we have $-[true \Rightarrow x \geq 0]$, and hence $-(C;L \leq L;C)$. (End of example.)

7. Conclusion

We have shown some new facts about the repetition. So, even the realm of ordinary sequential programs still contains some surprises.

References