Proof Rules for Recursive Procedures

Hesselink, Wim H.

Published in:
Formal Aspects of Computing

Publication date:
1993

Formal Aspects of Computing (1993) 5: 554–570

Formal Aspects of Computing

Proof Rules for Recursive Procedures

Wim H. Hesselink

Rijksuniversiteit Groningen, Department of Computing Science, Groningen, The Netherlands

Keywords: Proof rule; Recursive procedure; Specification; Weakest precondition; Well-founded relation

Abstract. Four proof rules for recursive procedures in a Pascal-like language are presented. The main rule deals with total correctness and is based on results of Gries and Martin. The rule is easier to apply than Martin's. It is introduced as an extension of a specification format for Pascal-procedures, with its associated correctness and invocation rules. It uses well-founded recursion and is proved under the postulate that a procedure is semantically equal to its body.

This rule for total correctness is compared with Hoare's rule for partial correctness of recursive procedures, in which no well-founded relation is needed. Both rules serve to prove correctness, i.e. sufficiency of certain preconditions. There are also two rules for proving necessity of preconditions. These rules can be used to give formal proofs of nontermination and refinement. They seem to be completely new.

1. Announcement of Aims and Results

In iterative programming, the proof rules for correctness are well established and have led to powerful methods for program derivation, culminating in [Kal90]. For recursive procedures in a language like Pascal, however, a generally accepted proof rule is still lacking. In [Hoa71], Hoare presented a derivation system to prove partial correctness of recursive procedures. This system has the disadvantages that it can only yield partial correctness and that it is incompatible with the full power of predicate calculus. In [Heh79] and [Mar83], methods are given to treat total correctness, but a concrete proof rule is not given. In particular, the combination of recursion and parameters remains implicit.

The purpose of the present paper is to propose a proof rule for total correctness

Correspondence and offprint requests to: Wim H. Hesselink, Rijksuniversiteit Groningen, Department of Computing Science, P.O. Box 800, 9700 AV Groningen, The Netherlands. email: wim@cs.rug.nl

of recursive procedures with value and reference parameters in a language like Pascal. The specifications are given in terms of preconditions and postconditions. The proof rule is given in three versions of increasing abstractness. The most abstract version is compared with a version of Hoare's Rule that is compatible with predicate calculus. A second purpose of this paper is to present two proof rules complementary to the rules for total and partial correctness. These new rules serve to prove necessity of preconditions instead of sufficiency.

2. Introduction

The main role of the procedure mechanism in programming is that it allows abstraction and thus serves to separate the invocation of a command from its implementation. This separation is accomplished by a full specification. The user of a procedure relies on the specification and the implementer has only the task to fulfil it. The user has an invocation rule to derive properties of the invocation from the specification. The implementer has a correctness rule to prove that a body can serve as an implementation.

Recursion arises when the implementers take the opportunity to use the procedure(s) they are implementing. In that case, the correctness rule requires the invocation rule as an induction hypothesis. The resulting problem of circularity has two solutions. The circularity can be ignored or it can be broken by means of well–founded relations.

In [Hoa71], Hoare proposes the first solution. He gives a rule that guarantees only partial correctness: if execution terminates, a postcondition is guaranteed to be established. Hoare's rule requires a concept of derivability. It can be formalized in so–called Hoare logic or in dynamic logic, but it has no obvious rendering in predicate calculus.

In [Mar83], Martin opts for the second solution. He gives a method to prove total correctness, i.e. partial correctness as well as termination. The method is not formalized to a concrete proof rule, and it has the disadvantage that it requires predicate calculus on two different state spaces: the space of the call and the space of the declaration.

The main result of the present paper is a proof rule for total correctness more explicit than Martin's method. The rule enables complete and rather concise correctness proofs by annotation. It has the same power as Martin's method, though –at first sight– it may seem to be weaker. The rule uses predicates on only one state space, the space of the call. In exchange for this advantage, we get the burden that there are more specification values (logical variables) needed to relate preconditions and postconditions. In fact, following [Gri81] p. 151, we require that the postcondition does not contain the value parameters, so that specification values are needed instead.

In section 3, we present a format for specified declaration of Pascal procedures. This format is accompanied by a correctness rule for the implementer and an invocation rule for the user of the procedure. The proof rule for total correctness of recursive procedures is given in three versions: (10), (17) and (19). Section 4 contains rule (10), in which natural numbers are sufficient to prove termination. Section 5 contains the stronger rule (17), which captures well-founded recursion. Section 6 contains rule (19), an abstraction in which parametrization, specification values, invariant predicates and mutual recursion are unified. Subsequently, rule (19) is proved.

Hoare's rule for partial correctness of recursive procedures is presented as rule (25) in section 7. Instead of a variant function, it uses quantification over semantic functions w that generalize the weakest precondition functions *wp* and *wlp*. This quantification serves to eliminate the need for a separate logic or derivation system, as in [Hoa71]. The new setting of Hoare's rule is used in section 8 to present a new rule, (27), for proving that the weakest precondition of a procedure implies a given predicate. This necessity rule can serve in formal proofs of nontermination and incorrectness. In section 9, we give an example where rule (27) is used to prove the correctness of a program transformation.

The three rules (19), (25) and (27) suggest the existence of a fourth rule. Indeed, in section 10, we present necessity rule (31) for partial correctness, which uses a variant function in a way similar to the rule for total correctness.

We use Hoare triples to denote total correctness, so that

(0) $\{P\}\ S\ \{Q\} \quad \equiv \quad [P \Rightarrow wp.S.Q]$

for predicates P and Q and a command S. The truth of either side of (0) expresses that P is a precondition that guarantees termination of command S in any state in which Q holds. All results in this paper are sound under (unbounded) nondeterminacy.

3. Declaration and Invocation

We begin with the declaration format, which is inspired by [Gri81] and [Mar83] but is more closely tuned to Pascal, in that we consider only global variables, value parameters and reference parameters. For simplicity, we assume that the procedure has one value parameter x and one reference parameter y and that it refers to two global variables u and v. The types of x, y, u, v are left out because they don't concern us here. We assume that precondition P and postcondition Q use one specification value $i \in C$. In this way we arrive at the specified declaration:

(1) **proc** $h(x$; **var** $y)$
$\{$**glo** $u, v!$; **all** $i \in C$ :: **pre** P, **post** $Q\}$.

The meaning of the specification is defined by

Correctness Rule. An implementation **body**.h of procedure h is correct if it satisfies the conditions:

(2) all global variables used in **body**.h and in predicates P and Q are listed after the key word **glo**; global variables that are threatened to be modified (in the sense of [JeW85]) are indicated by "!",

(3) the value parameters (x) do not occur in postcondition Q,

(4) for all specification values $i \in C$, we have $\{P\}$ **body**.h $\{Q\}$.

Of course, variable modalities that do not occur can be omitted from the specification. The list after key word **glo** is needed to exclude aliasing upon invocation of h. Condition (3) may seem unnecessarily restrictive. There are three reasons for imposing it. Firstly, it encourages specifications with simple

postconditions (J.E. Jonker). Secondly, if value parameters would be allowed in the postcondition, the invocation rule (to be treated below) would be complicated by the fact that the value of the expression for the argument can be modified by the call. Finally, condition (3) is necessary if one wants to combine condition (4) with the exploitation of value parameters as local variables (cf. [Gri81] chapter 12).

For recursive implementations the correctness rule is correct but inadequate, for requirement (4) is too strong; we come back to this in section 4.

A call of procedure h declared in (1) is of the form $h(E,t)$ where E is an expression and t is a variable, both of which are well defined at the position of the call. Following [Gri81], we speak of E and t as *arguments* of the call. The term "actual parameter" (cf. [JeW85]) had better be avoided since the adjective "actual" tends to be forgotten.

The meaning of a call $h(E,t)$ can only be inferred from the specification if there is no aliasing between reference parameters and global variables. We therefore require that reference argument t be not used as a global variable:

(5) $t \notin glo$,

where *glo* is the list headed by **glo**.

Precondition P and postcondition Q of specification h need not specify that global variables outside of list *glo!* are unchanged, where *glo!* is the sublist of *glo* that consists of the variables that are threatened to modified. In the specification of the call, an additional predicate R can be used to express this fact. Such an invariant predicate R is required to satisfy

(6) $Var.R \cap (\{t\} \cup glo!) = \emptyset$

where *Var.R* is the list of the variables that occur in R. If $glo! = glo$, conditions (5) and (6) are more symmetrically expressed by stating that the three lists *glo*, *Var.R*, and the list of reference parameters of h are pairwise disjoint.

The call $h(E,t)$ is specified by

Invocation Rule. (7) If (5) and (6), then for all $i \in C$:

$$\{P^{x,y}_{E,t} \wedge R\} \quad h(E,t) \quad \{Q^{y}_{t} \wedge R\}\,.$$

In words: in the expressions for P and Q, the parameters are replaced by the arguments.

Remark. If there are more reference parameters, the avoidance of aliasing also requires that all reference arguments differ. For simplicity of presentation we do not treat calls of the form $h(E, a[F])$ where the reference argument is a component of an array. For that case, the reader is referred to [Gri81] Chapter 12.

Example. A procedure to compute natural powers of integers can be specified (perhaps unexpectedly) by

proc *pow* (x : *integer* ; **var** y : *integer*)
{ **all** $Z \in$ *integer* :: **pre** $y \geq 0 \wedge Z = x^y$, **post** $y = Z$} .

Let u and v be global variables of type integer. Assume that we are looking for

a call of *pow* that satisfies, for all values U and V,

(8) $\{u = U \wedge v = V \geq 0\}$ *pow* (?, ?) $\{v = (U+V)^V \wedge u = U\}$.

Invocation rule (7) yields, for every expression E, every variable t, every value Z and every predicate R that does not use t,

(9) $\{t \geq 0 \wedge Z = E^t \wedge R\}$ *pow* (E, t) $\{t = Z \wedge R\}$.

Since the precondition and the postcondition of (8) both have the conjunct $u = U$, we take $R : u = U$. Subsequently, we see that v should be the reference argument and that Z should be $(U+V)^V$. In this way, problem (8) is solved by the annotated invocation

$\{u = U \wedge v = V \geq 0\}$ (* and hence *)
$\{v \geq 0 \wedge (U+V)^V = (u+v)^v \wedge u = U\}$
pow $(u+v, v)$ (* formula (9) with $E : u+v$ *)
$\{v = (U+V)^V \wedge u = U\}$.

Of course, we do not recommend procedures with such unexpected parameter behaviour. The point is that even such procedures can be treated adequately. Notice that the call modifies the value of the argument $u+v$ and that the correctness proof is not influenced by this fact. □

Rule (7) is a variation of Theorem (12.2.12) of [Gri81]. At first sight, it seems to be weaker than the rule of [Mar83], since the latter rule allows an arbitrary postcondition. Actually, the adaptation A of [Mar83] plays the same rôle as our invariant predicate R, although it is a predicate on a different state space. If one needs a rule with an arbitrary postcondition X, rule (7) easily yields that, for a variable $t \notin glo$ and an arbitrary predicate R, we have

$Var.R \cap (\{t\} \cup glo.) = \emptyset \wedge [Q_t^y \wedge R \Rightarrow X]$
$\Rightarrow \{P_{E,t}^{x,y} \wedge R\}$ $h(E, t)$ $\{X\}$.

In our experience, rule (7) is more convenient.

Notice that, since we do not allow value parameters in the postcondition, specification values are indispensable. In [Mar83], they are also useful, but they are only treated there, rather implicitly, in example 4.3.

4. Correctness of Recursive Declarations

We now assume that the body of h in declaration (1) is recursive. This implies that the Correctness Rule of section 3 must be adapted in such a way that Invocation Rule (7) can be applied to the recursive calls in the proof of condition (4). In order to preclude circularity, we use a variant function *vf*, just as in the proof rule for the repetition (see e.g. [Gri81] Theorem (11.6)). Roughly speaking, the condition is that the value of *vf* is smaller at every recursive call and that there is no recursion when $vf < 0$.

Correctness Rule. (10) A recursive implementation **body**.h of procedure h of specification (1) is correct if conditions (2) and (3) are satisfied and there is a $\mathbb{Z}$-valued function *vf* in the specification value i, the parameters x, y and the

variables in *glo*, such that, for every $m \in \mathbb{Z}$ and every $i \in C$, the induction hypothesis that every recursive call $h(E,t)$ satisfies

$$(11)\quad \{(P \wedge vf < m)^{i,x,y}_{j,E,t} \quad \wedge \quad m \geq 0 \quad \wedge \quad R\}$$
$$h(E,t)$$
$$\{Q^{i,y}_{j,t} \quad \wedge \quad R\}$$

(for all $j \in C$ and for all predicates R with (6)), implies

$$(12)\quad \{P \wedge vf = m\} \quad \textbf{body}.h \quad \{Q\}\,.$$

Rule (10) is long and cumbersome, but of course one cannot expect it to be simpler than the rule for the repetition. Theorem (19) below is a more concise and more abstract version, in which parametrization, specification values and invariant predicates are unified with mutual recursion.

The paper [Mar83] also deals with recursive procedures, but it does not give a concrete proof rule but rather a method. The main advantage of rule (10) over Martin's adaptation method is that the invariant predicates R are predicates on the state space of the (recursive) calls and hence can serve directly in a proof by annotation. In fact, rule (10) enables complete and rather concise proofs by annotation.

In our experience, undergraduates in mathematics and computer science that have learned to program repetitions with invariants and variant functions (e.g. see [Gri81] Theorem 11.6) can be taught to use Correctness Rule (10) for programming recursive procedures.

Notice that condition $m \geq 0$ enters only in the precondition of induction hypothesis (11), that is, in the precondition of the recursive call(s). Condition $m \geq 0$ must not be forgotten, for it is this bound that implies termination.

Example. Consider a procedure for integer division as specified in

proc *divi* (y : *integer*)
{**glo** $x!, q!$: *integer* ;
all $X, Y \in$ *integer* ::
pre $P : x = X \geq 0 \quad \wedge \quad y = Y > 0,$
post $Q : X = q \cdot Y + x \quad \wedge \quad 0 \leq x < Y\}$.

Specification value X is the initial value of global variable x. We use a specification value Y for the value of parameter y.

Postcondition Q is easily established if $X < Y$. Therefore, we use the variant function $vf = X - Y$. Let $m \in \mathbb{Z}$ be given. In the present case, the induction hypothesis (11) is that, for every expression E and all values X' and Y' and all predicates R that do not refer to x or q,

$$(13)\quad \{x = X' \geq 0 \wedge E = Y' > 0 \wedge X' - Y' < m \wedge m \geq 0 \wedge R\}$$
$$divi\ (E)$$
$$\{X' = q \cdot Y' + x \quad \wedge \quad 0 \leq x < Y' \quad \wedge \quad R\}\,.$$

Fragment (14) contains the body of *divi*, annotated in such a way that formula (12) is verified at the same time.

(14) $\{P \wedge vf = m : \quad x = X \geq 0 \quad \wedge \quad y = Y > 0 \quad \wedge \quad X - Y = m\}$
if $x < y$ **then** $\{x = X \geq 0 \quad \wedge \quad y = Y > 0 \quad \wedge \quad x < y\}$
$\{X = 0 \cdot Y + x \quad \wedge \quad 0 \leq x < Y\}$
$q := 0 \quad \{Q\}$
else $\{x = X \geq 0 \quad \wedge \quad y = Y > 0 \quad \wedge \quad X - Y = m \quad \wedge \quad y \leq x\}$
$\{x = X \geq 0 \quad \wedge \quad 2 \cdot y = 2 \cdot Y > 0 \quad \wedge \quad X - 2 \cdot Y < m$
$\wedge \quad m \geq 0 \quad \wedge \quad y = Y\}$
$divi(2 \cdot y)$; (* (13) with $E : 2 \cdot y$ and $R : (y = Y)$ *)
$\{X = q \cdot 2 \cdot Y + x \quad \wedge \quad 0 \leq x < 2 \cdot Y \quad \wedge \quad y = Y\}$
$q := q \cdot 2$;
$\{X = q \cdot Y + x \quad \wedge \quad 0 \leq x < 2 \cdot Y \quad \wedge \quad y = Y\}$
if $x < y$ **then** *skip* $\{Q\}$
else $\{X = (q + 1) \cdot Y + x - y \quad \wedge \quad 0 \leq x - y < Y\}$
$x := x - y \quad \{X = (q + 1) \cdot Y + x \quad \wedge \quad 0 \leq x < Y\}$;
$q := q + 1 \quad \{Q\}$
fi {collect branches: Q}
fi {collect branches: Q} .

Clearly, if constant parameters are allowed (in the sense of [Gri81] Theorem (12.4.1)) and y is taken to be such a constant parameter, the specification value Y is superfluous. □

Remark. An annotated program should not be regarded as a presentation but as a witness of a proof of correctness. In particular, its sequential order need not be related to the order of reading. The nesting of the annotated program gives more indication of the structure. □

Example. In order to enable a comparison with the method of [Mar83], we treat J. McCarthy's 91-function as presented in [Mar83], section 7. The value parameter in the postcondition used there is eliminated by means of specification value Y. We use infix operator **max** to denote the maximum of its operands. The specified declaration is

proc $p\,(x$: *integer*; **var** y : *integer*)
{**all** $Y \in$ *integer* :: **pre** $Y = 91$ **max** $(x - 10)$, **post** $Y = y\}$;
var z : *integer*;
if $x > 100$ **then** $y := x - 10$
else $p\,(x + 11, z)$; $p(z, y)$
fi.

We use proof rule (10) to prove correctness. In view of the procedure body, we choose the variant function $vf = 100 - x$. Let $m \in \mathbb{Z}$ be given. The induction hypothesis is that, for every expression E, every variable t, every predicate R with $t \notin$ Var.R and every value Z,

$\{Z = 91$ **max** $(E - 10) \quad \wedge \quad 100 - E < m \quad \wedge \quad m \geq 0 \quad \wedge \quad R\}$
$p\,(E, t)$
$\{Z = t \quad \wedge \quad R\}$.

We now verify that the body of procedure p satisfies proof obligation (12).

$\{Y = 91 \textbf{ max } (x-10) \ \wedge \ 100 - x = m\}$
if $x > 100$ **then**
  $\{x > 100 \ \wedge \ Y = 91 \textbf{ max } (x-10) \ \wedge \ 100 - x = m\}$
  $\{Y = x - 10\}$
  $y := x - 10$
  $\{Y = y\}$
else $\{x \leq 100 \ \wedge \ Y = 91 \textbf{ max } (x-10) \ \wedge \ 100 - x = m\}$
  (* introduce a value Z *)
  $\{Z = 91 \textbf{ max } ((x+11)-10) \wedge 100 - (x+11) < m \wedge m \geq 0$
  $\wedge \ (Z = 91 \textbf{ max } (x+1) \wedge 100 - x = m \wedge m \geq 0 \wedge Y = 91)\}$
  $p(x+11, z)$ (* ind. hyp. with R between parentheses *)
  $\{Z = z \ \wedge \ Z = 91 \textbf{ max } (x+1) \ \wedge \ 100 - x = m$
  $\wedge \ m \geq 0 \ \wedge \ Y = 91)\}$
  (* $z - 10 \leq 81 \textbf{ max } (x-9) \leq 91$ and $z \geq x + 1$ *)
  $\{Y = 91 \textbf{ max } (z-10) \ \wedge \ 100 - z < m \ \wedge \ m \geq 0\}$
  $p(z, y)$ (* ind. hyp. with R : *true* *)
  $\{Y = y\}$
fi $\{$collect branches: $Y = y\}$.

In the first case, the invention of predicate R is driven largely by the postcondition of the call, which is the precondition of the next call. □

5. Termination and Well-founded Subsets

The method of [Mar83] is stronger than proof rule (10) since it allows the use of arbitrary well-founded sets in termination proofs. In fact, some termination arguments for repetitions and recursive procedures need a lexical order, e.g. the unification algorithm (cf. [Gal87] p.391) and the garbage collection algorithms of [Jon92].

On the other hand, it is often useful that the variant function is allowed to take values outside of the well-founded set. In fact, both examples in the previous section had negative values for *vf* in the nonrecursive alternative. Therefore, instead of a well-founded set, we use a well-founded subset, (cf. [DiS90] p. 174).

Let "$<$" be a binary relation on a set Z. A subset N of Z is said to be *well-founded* with respect to $<$ if and only if every nonempty subset, say S, of N has a minimal element. Here, an element x is called a *minimal* element of subset S if and only if

(15) $x \in S \ \wedge \ (\forall\, y : y < x : y \notin S)$.

The standard example is the subset $\mathbb{N}$, the set of the natural numbers in the set $\mathbb{Z}$ of the integers with the usual "less than" relation $<$.

We let N be a well-founded subset of a set Z with a relation $<$. The principle of well-founded induction over N (see e.g. [DiS90] p. 176) states that, for any

predicate f on Z,

(16) $(\forall\, x \in N :: f.x)$
$\equiv \quad (\forall\, x \in N :: f.x \Leftarrow (\forall\, y \in N : y < x : f.y))$.

Using this triple $(Z, <, N)$, rule (10) is generalized to the following rule, which has the same power as the method of [Mar83].

Correctness Rule. (17) A recursive implementation **body**.h of procedure h of specification (1) is correct if conditions (2) and (3) are satisfied and there is a Z-valued function vf in the specification value i, the parameters x, y and the variables in glo, such that, for every $m \in Z$ and every $i \in C$, the induction hypothesis that every recursive call $h(E,t)$ satisfies, for all $j \in C$ and for all predicates R with (6),

$$\{(P \wedge vf < m)^{i,x,y}_{j,E,t} \quad \wedge \quad m \in N \quad \wedge \quad R\}$$
$$h(E,t)$$
$$\{Q^{i,y}_{j,t} \quad \wedge \quad R\}$$

implies $\quad \{P \wedge vf = m\} \quad \textbf{body}.h \quad \{Q\}$.

6. A More Abstract Recursion Theorem

For the proof of the soundness of rule (17), some grip on the semantics of recursive procedures is necessary. We do not completely define the weakest precondition $wp.k$ for the call of an arbitrary procedure k. We only postulate

(18) $wp.k = wp.(\textbf{body}.k)$,

or equivalently, for all predicates P and Q,

$$\{P\}\, k\, \{Q\} \quad \equiv \quad \{P\}\, \textbf{body}.k\, \{Q\}\ .$$

Here we assume that possible parameters are part of the name k.

If the declaration of k does not contain recursion, postulate (18) is clearly consistent and strong enough to define the semantics of calls of k. In the case of a recursive declaration, it is not clear that equation (18) is consistent or applicable. In [Hes90], it is shown that, indeed, equation (18) has a solution. In general, however, it may have many solutions. The applicability of postulate (18) is shown below by proving an abstraction of rule (17).

A direct proof of correctness rule (17) would have to be based on the induction hypothesis with its mess of renamings. Therefore, we apply abstraction to unify parametrization, specification values and invariant predicates with mutual recursion. A procedure with parameters can be regarded as a family of procedures. If the procedure is recursive, it is a family of mutually recursive procedures. Each of these procedures, say $h.\alpha$, may be specified by a family of Hoare triples

$$\{P.\alpha.\beta\} \quad h.\alpha \quad \{Q.\alpha.\beta\}\ .$$

In this way, specification values and the invariant predicates R, as used above, can be accommodated. If we now encode the pair $\langle \alpha, \beta \rangle$ in a single symbol i and write $h.i = h.\alpha$, we get a family of procedures $h.i$ with preconditions $P.i$ and postconditions $Q.i$, where i ranges over some set. In this way, we obtain the following abstract version of rule (17).

Theorem. (19) Consider a family of procedures $h.i$ with preconditions $P.i$ and postconditions $Q.i$, where i ranges over some set I. Let N be a well-founded subset of a set Z with relation $<$. For every $i \in I$, let $vf.i$ be a Z–valued state function. Assume that, for every $m \in Z$,

$$(\forall\, i \in I :: \{P.i \wedge vf.i < m \wedge m \in N\}\ h.i\ \{Q.i\}\ )$$
$$\Rightarrow \quad (\forall\, i \in I :: \{P.i \wedge vf.i = m\}\ \mathbf{body}.(h.i)\ \{Q.i\}\ ).$$

Then $\{P.i\}\ h.i\ \{Q.i\}$ for all $i \in I$.

Remark. The antecedent of the implication is called the induction hypothesis. The theorem implicitly allows mutual recursion. In fact, the body of $h.i$ may call $h.j$. □

Proof. In view of postulate (18), we have

$$(20)\quad (\forall\, i \in I :: \{P.i \wedge vf.i < m \wedge m \in N\}\quad h.i\quad \{Q.i\}\ )$$
$$\Rightarrow \quad (\forall\, i \in I :: \{P.i \wedge vf.i = m\}\quad h.i\quad \{Q.i\}\ ).$$

If $m \notin N$, the precondition of the antecedent is false. Therefore, all Hoare triples of the antecedent are true, so that formula (20) implies

$$(21)\quad (\forall\, i \in I, m \in Z \setminus N :: \{P.i \wedge vf.i = m\}\quad h.i\quad \{Q.i\}\ ).$$

By well-founded induction (16) with (20) and (21), we obtain that, for all $m \in N$,

$$(\forall\, i \in I :: \{P.i \wedge vf.i = m\}\quad h.i\quad \{Q.i\}\ ).$$

This implies that, for all $i \in I$ and $m \in Z$,

$$\{P.i \wedge vf.i = m\}\quad h.i\quad \{Q.i\}\ .$$

On the other hand, for every $i \in I$, we have $[\,(\exists\, m \in Z :: vf.i = m)\,]$; in fact, for every state x there is a value m with $vf.i.x = m$. This implies that, for all $i \in I$,

$$\{P.i\}\quad h.i\quad \{Q.i\}\ .$$ □

7. Hoare's Rule for Partial Correctness

In this section, we present a version of Hoare's Induction Rule for recursive procedures. The main difference with Rule (19) is that there is no guarantee of termination and therefore no need for well-founded sets. So, Hoare's rule is about partial correctness. Recall from [DiS90] chapter 7, that partial correctness can be expressed by means of the predicate transformer *wlp* (weakest liberal precondition). As a first approximation, Hoare's Rule for a procedure h with precondition P and postcondition Q reads:

If $[P \Rightarrow wlp.(\mathbf{body}.h).Q\,]$ *can be inferred from* $[P \Rightarrow wlp.h.Q\,]$, *then* $[P \Rightarrow wlp.h.Q\,]$.

Of course, the words "can be inferred" must not be read as a material implication. For, otherwise, the proposition $A : [P \Rightarrow wlp.h.Q\,]$ would satisfy $\neg A \Rightarrow A$, so that A would be a tautology. Since we do not want to present a separate logic, we formalize Hoare's Rule in predicate calculus by quantifying over the interpretation *wlp*.

Let W be the set of functions w from commands to conjunctive predicate

transformers that satisfy the laws:

(22) $w.c.Q = wp.c.Q$,
$w.(s;t).Q = w.s.(w.t.Q)$,
$w.(s \,[\!]\, t).Q = w.s.Q \wedge w.t.Q$,

for all simple commands c, all commands s, t and all predicates Q. Here, the operator "$[\!]$" stands for nondeterminate choice. We assume that all simple commands c always terminate, so that $wlp.c = wp.c$. This implies that both wp and wlp are elements of W.

Every assignment $t := E$ is a simple command with $wp.(t := E).Q = Q^t_E$. For a predicate b, the *guard* $?b$ is defined to be the simple command given by

$$wp.(?b).Q = (b \Rightarrow Q).$$

Then the conditional construct can be expressed as

$$\textbf{if } b \textbf{ then } s \textbf{ else } t \textbf{ fi} = (?b \,;\, s \,[\!]\, ?(\neg b) \,;\, t).$$

It follows that every $w \in W$ satisfies

(23) $w.(\textbf{if } b \textbf{ then } s \textbf{ else } t \textbf{ fi}).Q = (b \Rightarrow w.s.Q) \wedge (\neg b \Rightarrow w.t.Q)$.

A command built from simple commands by means of the constructors ";" and "$[\!]$" is called a *straight-line* command. It is clear that every straight-line command s satisfies

(24) $w.s = wp.s = wlp.s$ for all $w \in W$.

Now the above version of Hoare's Rule is formalized to

If $[P \Rightarrow w.h.Q]$ *implies* $[P \Rightarrow w.(\textbf{body}.h).Q]$ *for every* $w \in W$, *then* $[P \Rightarrow wlp.h.Q]$.

This rule is valid but not very applicable, for it has no place for parameters, specification values or mutual recursion. A stronger version is formulated as follows.

Hoare's Rule. (25) Consider a family of procedures $h.i$ with preconditions $P.i$ and postconditions $Q.i$, where i ranges over some set. Assume that, for every $w \in W$,

$$(\forall\, i :: [P.i \Rightarrow w.(h.i).(Q.i)])$$
$$\Rightarrow \quad (\forall\, i :: [P.i \Rightarrow w.(\textbf{body}.(h.i)).(Q.i)]).$$

Then $[P.i \Rightarrow wlp.(h.i).(Q.i)]$ for all i.

Remark. The similarity of this rule to Theorem (19) is more striking if formula (0) is used to rewrite (19) in terms of wp. Rule (25) can be proved from the definition of wlp as the weakest solution of equation (18) in the set W. In [Hes92] Chapter 4, we give a proof of a stronger version, in which the set W is somewhat smaller. □

Example. We give an example in which Hoare's Induction Rule is used to prove partial correctness. In this example, total correctness fails, so that Theorem (19) cannot be used.

Let t be an integer program variable. Let procedure h be declared by

(26) $\textbf{body}.h = (skip \,[\!]\, t := t - 1 \,;\, h \,;\, t := t + 2)$.

Operationally, it is clear that h need not terminate, but if h terminates then t is not smaller than it was before. We therefore guess that

$$[\, t \geq i \;\; \Rightarrow \;\; wlp.h.(t \geq i)\,] \qquad \text{for all } i \in \mathbb{Z}.$$

This is proved by means of Hoare's Rule in the following way. We let i range over $\mathbb{Z}$, choose all $h.i = h$, and the predicates $P.i$ and $Q.i$ equal to $t \geq i$. We have to prove the proper instantiation of the assumption of (25). So, we let $w \in W$ be a function that satisfies the induction hypothesis

$$[\, t \geq i \;\; \Rightarrow \;\; w.h.(t \geq i)\,] \qquad \text{for all } i \in \mathbb{Z}.$$

Now it suffices to observe

$$\begin{array}{ll} & w.(\mathbf{body}.h).(t \geq i) \\ = & \{\text{declaration (26) of } h\} \\ & w.(skip \;[\!]\; t := t-1 \;;\; h \;;\; t := t+2).(t \geq i) \\ = & \{(22) \text{ and } skip = (?true)\} \\ & t \geq i \;\; \wedge \;\; w.(t := t-1).(w.h.(t \geq i-2)) \\ \Leftarrow & \{\text{induction hypothesis with } i := i-2 \\ & \quad \text{and monotony of } w.(t := t-1)\} \\ & t \geq i \;\; \wedge \;\; w.(t := t-1).(t \geq i-2) \\ = & \{(22) \text{ and calculus }\} \\ & t \geq i. \end{array}$$

A formal proof that h need not terminate is given below as an application of the next rule. □

8. A Necessity Rule for Total Correctness

As far as we know, the next induction rule is new. It deals with necessity of preconditions instead of sufficiency. In fact, when dealing with program correctness, we are interested only in the question whether a given predicate implies the weakest (liberal) 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 for *wp*.

Necessity Rule. (27) Assume that for every $w \in W$

$$\begin{array}{l} (\forall\, i :: [\, w.(h.i).(Q.i) \;\Rightarrow\; P.i\,]) \\ \Rightarrow \quad (\forall\, i :: [\, w.(\mathbf{body}.(h.i)).(Q.i) \;\Rightarrow\; P.i\,]). \end{array}$$

Then $[\, wp.(h.i).(Q.i) \;\Rightarrow\; P.i\,]$ for all i.

This rule is based on the postulate that *wp* is the strongest solution of (18) in the set W. The rule is not useful for proofs of program correctness. It can be used, however, for proofs of nontermination, proofs of incorrectness, and proofs of refinements (see below in Section 8). It is not useful to imagine an operational interpretation of the rule. The specialization of the rule to the repetition is given in [Hes91] Section 4.

Example. We use rule (27) to prove that, for every initial state, procedure h of declaration (26) need not terminate. This is formalized in $wp.h.true = false$, or equivalently $[\, wp.h.true \Rightarrow false\,]$. By Rule (27), it suffices to prove that, for every $w \in W$,

$$[\, w.h.true \Rightarrow false\,] \quad \Rightarrow \quad [\, w.(\mathbf{body}.h).true \Rightarrow false\,],$$

or equivalently

$$[\, \neg w.h.true\,] \quad \Rightarrow \quad [\, \neg w.(\mathbf{body}.h).true\,].$$

Therefore, it suffices to use the induction hypothesis $[\, \neg w.h.true\,]$ and to observe

$$\begin{array}{ll} & w.(\mathbf{body}.h).true \\ = & \{\text{declaration (26) of } h\} \\ & w.(skip \,[\!]\, t := t-1 \;;\; h \;;\; t := t+2).true \\ = & \{(22) \text{ and } skip = (?true)\} \\ & w.(t := t-1).(w.h.true) \\ = & \{\text{induction hypothesis and (22)}\} \\ & false\,. \end{array}$$

□

Example. In the previous example, nontermination is operationally obvious. We therefore present another case in which nontermination is not easy to see.

Let t be an integer program variable. For integer constants a, b and c, let procedure h be declared by

$$\begin{array}{lll} \mathbf{body}.h & = & \mathbf{if}\ t > a\ \mathbf{then}\ t := t-b \\ & & \mathbf{else}\ t := t+c \;;\; h \;;\; h\ \mathbf{fi}. \end{array}$$

If $a = 100$, $b = 10$ and $c = 11$, this is a different coding of McCarthy's 91–function, see Section 3. We now consider the case that b and c are integers with $b \geq c$. Then procedure h only terminates in the nonrecursive case, i.e., under precondition $t > a$. This is formalized in

(28) $[\, wp.h.true \Rightarrow t > a\,]$.

If one tries to prove (28) by means of rule (27), one also needs assertions of the form

$$[\, wp.h.(t > i) \Rightarrow t > b + i\,] \quad \text{for integer } i.$$

This family of assertions is unified with (28) in

(29) $[\, wp.h.(t > i) \Rightarrow t > a \ \mathbf{max}\ (b+i)\,]$ for all $i \in \mathbb{Z} \cup \{-\infty\}$.

Notice that formula (29) with $i = -\infty$ implies formula (28). Formula (29) is proved by means of rule (27). In fact, it suffices to consider $w \in W$ that satisfies

(30) $[\, w.h.(t > i) \Rightarrow t > a \ \mathbf{max}\ (b+i)\,]$ for all $i \in \mathbb{Z} \cup \{-\infty\}$

and to verify

$$\begin{array}{ll} & w.(\mathbf{body}.h).(t > i) \\ = & \{\text{declaration of } h \text{ and (23)}\} \\ & (t > a \;\Rightarrow\; w.(t := t-b).(t > i)) \\ & \wedge \quad (t \leq a \;\Rightarrow\; w.(t := t+c \;;\; h \;;\; h).(t > i)) \end{array}$$

$$
\begin{array}{ll}
\Rightarrow & \{(22) \text{ and } (30) \text{ twice}\} \\
& (t > a \Rightarrow t - b > i) \\
& \wedge \quad (t \leq a \Rightarrow t + c > a \ \mathbf{max}\ (b + a \ \mathbf{max}\ (b + i))) \\
= & \{\text{calculus}\} \\
& (t > a \Rightarrow t > b + i) \\
& \wedge \quad (t \leq a \Rightarrow t > (a - c) \ \mathbf{max}\ (a + b - c) \ \mathbf{max}\ (2 \cdot b + i - c)) \\
= & \{\text{since } b \geq c \text{ the second conjunct equivales } t > a\} \\
& t > a \ \mathbf{max}\ (b + i).
\end{array}
$$

This proves formula (29) and hence formula (28). □

9. Refinement of Procedures

In this section, we give an example of an application of necessity rule (27) to program transformation. The example shows formally that refining the constituents of a procedure leads to a refinement of the procedure itself. Of course, this result is not surprising. The point is that rule (27) does the job.

Let procedures h_0 and h_1 be declared by

$$\mathbf{body}.h_i \quad = \quad (r_i \,[\!]\, s_i \,;\, h_i \,;\, t_i)$$

for $i = 0$ or 1, where r_i, s_i, t_i are straight-line commands for $i = 0$ or 1. Assume that r_0, s_0 and t_0 are refined by r_1, s_1, t_1, respectively. This means that, for all predicates q,

$$
\begin{array}{l}
[\, wp.r_0.q \ \Rightarrow \ wp.r_1.q \,] \\
[\, wp.s_0.q \ \Rightarrow \ wp.s_1.q \,] \\
[\, wp.t_0.q \ \Rightarrow \ wp.t_1.q \,].
\end{array}
$$

Then we claim that h_0 is refined by h_1, that is

$$[\, wp.h_0.q \ \Rightarrow \ wp.h_1.q \,] \qquad \text{for all predicates } q.$$

Our proof obligation fits Rule (27) with q ranging over the set of all predicates and

$$h.q = h_0 \quad , \quad Q.q = q \quad , \quad P.q = wp.h_1.q.$$

Therefore, by Rule (27), it suffices to prove that for every $w \in W$

$$
\begin{array}{ll}
& (\forall\, q :: [w.h_0.q \Rightarrow wp.h_1.q]) \qquad \{\text{induction hypothesis}\} \\
\Rightarrow & (\forall\, q :: [w.(\mathbf{body}.h_0).q \Rightarrow wp.h_1.q]).
\end{array}
$$

This is proved by observing that for every predicate q

$$
\begin{array}{ll}
& w.(\mathbf{body}.h_0).q \\
= & \{\text{declaration } h_0\} \\
& w.(r_0 \,[\!]\, s_0 \,;\, h_0 \,;\, t_0).q \\
= & \{(22)\} \\
& w.r_0.q \quad \wedge \quad w.s_0.(w.h_0.(w.t_0.q)) \\
= & \{r_0, s_0, t_0 \text{ are straight-line commands and } (24)\}
\end{array}
$$

$$
\begin{array}{ll}
& wp.r_0.q \ \wedge \ wp.s_0.(w.h_0.(wp.t_0.q)) \\
\Rightarrow & \{\text{induction hypothesis with } q := wp.t_0.q, \text{ and monotony}\} \\
& wp.r_0.q \ \wedge \ wp.s_0.(wp.h_1.(wp.t_0.q)) \\
\Rightarrow & \{\text{assumption and monotony}\} \\
& wp.r_1.q \ \wedge \ wp.s_1.(wp.h_1.(wp.t_1.q)) \\
= & \{\text{same calculation as in the first two steps}\} \\
& wp.(\mathbf{body}.h_1).q \\
= & \{(18)\} \\
& wp.h_1.q.
\end{array}
$$

This example can be modified by substituting *wlp* for *wp*. In that case, one applies Rule (25) to procedure h_1 instead of applying Rule (27) to h_0.

10. The Remaining Rule

Comparing the three rules (19), (25) and (27), we see striking similarities. Rules (25) and (27) use induction over elements $w \in W$ and lead to partial correctness of sufficient preconditions and total correctness for necessary preconditions. Rule (19) uses well-founded induction to yield total correctness of sufficient preconditions. This suggests that there should also exist a rule with well-founded induction that yields partial correctness for necessary preconditions.

As in Theorem (19), we let N be a well-founded subset of a set Z with a relation $<$.

Theorem. (31) For every $i \in I$, let $vf.i$ be a Z-valued state function. Assume that, for every $m \in Z$,

$$
\begin{array}{l}
(\forall\, i :: [vf.i < m \wedge m \in N \wedge wlp.(h.i).(Q.i) \ \Rightarrow \ P.i]) \\
\Rightarrow \quad (\forall\, i :: [vf.i = m \wedge wlp.(\mathbf{body}.(h.i)).(Q.i) \ \Rightarrow \ P.i]).
\end{array}
$$

Then $[wlp.(h.i).(Q.i) \ \Rightarrow \ P.i]$ for all i.

Proof. The proof is completely analogous to the proof of (19) (see [Hes92] Chapter 5). □

Example. Again, we consider procedure h declared in (26). We proved already that h need not terminate. Operationally, it is clear that, if h terminates, the value of t may be arbitrarily large. This assertion is formalized in

$$(\forall\, i \in \mathbb{Z} :: [\neg wlp.h.(t < i)]).$$

This formula can be proved by means of Theorem (31) with $h.i = h$ and $P.i = false$ and $Q.i = (t < i)$ for all $i \in \mathbb{Z}$. We use the standard well-founded subset $\mathbb{N}$ of $\mathbb{Z}$. By (31) it suffices to give a family of $\mathbb{Z}$-valued state functions $vf.i$ such that for every $m \in \mathbb{Z}$ the induction hypothesis

$$(\forall\, i \in \mathbb{Z} :: [vf.i < m \wedge m \in \mathbb{N} \wedge wlp.h.(t < i) \ \Rightarrow \ false])$$

implies for every i

$$[\, vf.i = m \ \wedge \ wlp.(\mathbf{body}.h).(t < i) \ \Rightarrow \ false].$$

To this end we first observe that the induction hypothesis is equivalent to

(31) $(\forall\, i \in \mathbb{Z} :: [\, wlp.h.(t < i) \;\Rightarrow\; \neg(vf.i < m \wedge m \in \mathbb{N})\,])$.

Now our proof obligation is fulfilled by

$$
\begin{array}{ll}
 & vf.i = m \;\wedge\; wlp.(\textbf{body}.h).(t < i) \\
= & \{\text{declaration of } h\} \\
 & vf.i = m \;\wedge\; wlp.(skip \,[\!]\, t := t-1 \,;\, h \,;\, t := t+2).(t < i) \\
= & \{(22)\} \\
 & vf.i = m \;\wedge\; t < i \;\wedge\; wlp.(t := t-1 \,;\, h).(t < i-2) \\
\Rightarrow & \{(32) \text{ with } i := i-2\} \\
 & vf.i = m \;\wedge\; t < i \;\wedge \\
 & wlp.(t := t-1).(\neg(vf.(i-2) < m \wedge m \in \mathbb{N})) \\
\Rightarrow & \{\text{choose } vf.i = i - t, \text{ then first two conjuncts} \\
 & \quad \text{imply } m > 0 \text{ and hence } m \in \mathbb{N}; \\
 & \quad \text{since } m \text{ is constant, this can be used in third conjunct}\} \\
 & i - t = m \;\wedge\; wlp.(t := t-1).(\neg(i-2-t < m)) \\
= & \{\text{calculus}\} \\
 & i - t = m \;\wedge\; i - 2 - (t-1) \geq m \\
= & \{\text{calculus}\} \\
 & false\,.
\end{array}
$$

Notice that the choice of $vf.i$ is delayed until the point where the expression indicates a useful choice. □

11. Concluding Remarks

The quadruplet of rules (19), (25), (27), (31) seems to be complete in some language in which mutually recursive procedures are compared with specifications. For practical applications, rules (25), (27), (31) can be extended with parameters, specification values, and invariant predicates to yield rules like (10) and (17). The quadruplet can also be specialized to yield correctness rules and necessity rules for total and partial correctness of the repetition. In that case, the correctness rules are well known and the necessity rule for *wp* is contained in [Hes91].

Acknowledgements

Sections 1 and 2 have been deeply influenced by discussions with J.E. Jonker and D. de Vries. E. Voermans suggested the possibility of necessity rules. I am very grateful for the criticisms of two referees that led to a complete revision of the presentation.

References

[DiS90] Dijkstra, E. W. and Scholten, C. S.: Predicate calculus and program semantics. Springer V. 1990.

[Gal87] Gallier, J.H.: Logic for Computer Science. Foundations of automatic theorem proving. Wiley & Sons 1987.

[Gri81] Gries, D.: The science of programming. Springer V. 1981.

[Heh79] Hehner, E.C.R.: **do** Considered **od** : a contribution to programming calculus. Acta Informatica 11 (1979) 287–304.

[Hes90] Hesselink, W.H.: Command algebras, recursion and program transformation. Formal Aspects of Computing **2** (1990) 60–104.

[Hes91] Hesselink, W.H.: Repetitions, known or unknown? Information Processing Letters **40** (1991) 51–57.

[Hes92] Hesselink, W.H.: Programs, Recursion and Unbounded Choice, predicate transformation semantics and transformation rules. Cambridge University Press, 1992 (Cambridge Tracts in Theoretical Computer Science 27).

[Hoa71] Hoare, C.A.R.: Procedures and parameters: an axiomatic approach. In: Symposium on Semantics of Algorithmic Languages. (ed. E. Engeler), Springer V. (Lecture Notes in Math. 188) 1971, pp. 102–116.

[JeW85] Jensen, K. and Wirth, N.: Pascal User Manual and Report, third edition. Springer V. 1985.

[Jon92] Jonker, J.E.: On-the-fly garbage collection for several mutators. Distr. Comput. **5** (1992) 187–199.

[Kal90] Kaldewaij, A.: Programming: the Derivation of Algrithms. Prentice Hall International, 1990.

[Mar83] Martin, A.J.: A general proof rule for procedures in predicate transformer semantics. Acta Informatica 20 (1983), 301–313.

Received December 1990
Accepted in revised form February 1993 by D. Gries