PROCEDURES AND PARAMETERS: AN AXIOMATIC APPROACH

by

C. A. R. Hoare

Introduction

It has been suggested, Hoare (1969), that an axiomatic approach to formal language definition might simultaneously contribute to the clarity and reliability of programs expressed in the language, and to the efficiency of their translation and execution on an electronic digital computer. This paper gives an example of the application of the axiomatic method to the definition of procedure and parameter passing features of a high-level programming language. It reveals that ease of demonstrating program correctness and high efficiency of implementation may be achieved simultaneously, provided that the programmer is willing to observe a certain familiar and natural discipline in his use of parameters.

CONCEPTS AND NOTATIONS

The notations used in this paper are mainly those of symbolic logic and particularly natural deduction. They are supplemented by conventions introduced in Hoare (1969). The more important of them are summarized below.

1. $P\{Q\}R$ --where P and R are propositional formulae of logic and Q is a part of a program. Explanation: If P is true of the program variables before executing the first statement of the program Q, and if the program Q terminates, then R will be true of the program variables after execution of Q is complete.

2. S_e^x --where S is an expression or formula, x is a variable, and e is an expression. Explanation: The result of replacing all free occurrences of x in S by e. If e is not free for x in S, a preliminary systematic change of bound variables of S is assumed to be made.

3. $\frac{A\ ,\ B}{C}$ --where A , B , and C are propositional formulae. Explanation: A rule of inference which states that if A and B have been proved, then C may be deduced.

4. $\frac{A\ ,\ B \vdash C}{D}$ --where A , B , C , and D are propositional formulae. Explanation: A rule of inference which permits deduction of D if A and C are proved; however it also permits B to be assumed as a hypothesis in the proof of C . The deduction of C from B is known as a subsidiary deduction.

It is assumed that, with the exception of program material (usually enclosed in braces), all letters stand for formulae of some suitably chosen logical system. The formulae of this system are presumed to include:

(a) all expressions of the programming language;

(b) the familiar notations of predicate calculus (truth-functions, quantifiers, etc.).

The properties of the basic operands, operators and built-in functions of the language are assumed to be specified by some suitably chosen axiom set; and the proof procedures are assumed to be those of the first-order predicate calculus.

As a simple example of the use of these notations, let Q stand for the single assignment statement $k := (m+n) \div 2$. We wish to prove that after execution of this statement, k will take a value between m and n , whenever $m \leq n$; or more formally that the desired result R of execution of Q is $m \leq k \leq n$. In Hoare (1969) there was introduced the axiom schema

$$R_e^x \{x := e\} R \ . \qquad \text{Axiom of Assignment}$$

This indicates that for an assignment $x := e$, if R is the desired result of the assignment, a sufficient precondition for this result to obtain is that R_e^x is true before execution. R_e^x is derived by replacing all occurrences of the target variable x in R by the assigned expression e . In the present case, the target variable is k , and the assigned expression is $(m+n) \div 2$. Thus we obtain, as an instance of the axiom schema,

$$m \leq (m+n) \div 2 \leq n \ \{k := (m+n) \div 2\} \ m \leq k \leq n \ .$$

It is an obvious theorem of mathematics that $m \leq (m+n)+2 \leq n$ can be inferred from the truth of $m \leq n$. This may be written using the notation explained above

$$m \leq n \vdash m \leq (m+n)+2 \leq n \ .$$

Another obvious rule mentioned in Hoare (1969) states that if $S\{Q\}R$ has been proved, and also that the truth of S may be inferred from the truth of P , then $P\{Q\}R$ is a valid inference, or more formally

$$\frac{S\{Q\}R \ , \ P \vdash S}{P\{Q\}R} \ . \qquad \text{Rule of Consequence}$$

In applying this rule to our example, we take $m \leq n$ for P and $m \leq (m+n)+2 \leq n$ for S , and obtain

$$m \leq n \ \{k := m+n+2\} \ m \leq k \leq n \ ,$$

which is what was required to be proved.

A full list of rules of inference, together with associated conventions, is given in the Appendix. They are not supposed to give a "complete" proof procedure for correctness of programs, but they will probably be found adequate for the proof of most practical algorithms.

PROCEDURES

Before embarking on a treatment of parameters, it is convenient first to consider the simpler case of a procedure without parameters. Suppose p has been declared as a parameterless procedure, with body Q . We introduce the notation

$$p \ \underline{\text{proc}} \ Q$$

to represent this declaration. A call of this procedure will take the form

$$\underline{\text{call}} \ p \ .$$

It is generally accepted that the effect of each call of a procedure is to execute the body Q of the procedure in the place of the call. Thus if we wish to prove that a certain consequence R will follow from a call of p (under some precondition

P), all we have to do is to prove that this consequence will result from execution of Q (under the same precondition). This reasoning leads to an inference rule of the form

$$\frac{p \text{ proc } Q\,,\ P\{Q\}R}{P\ \{\underline{\text{call}}\ p\}\ R}\ .$$ Rule of Invocation

PARAMETERS

In dealing with parametrization, we shall treat in detail the case where all variables of the procedure body Q (other than locally declared variables) are formal parameters of the procedure. Furthermore, we shall make a clear notational distinction between those formal parameters which are subject to assignment in the body Q , and those which do not appear to the left of an assignment in Q nor in any procedure called by it.

These decisions merely simplify the discussion; they do not involve any loss of generality, since any program can fairly readily be transformed to one which observes the conventions.

Let $\underline{x}$ be a list of all non-local variables of Q which are subject to change by Q . Let $\underline{y}$ be a list of all other non-local variables of Q . We extend the notation for procedure declarations, thus

$$p(\underline{x}) : (\underline{y})\ \underline{\text{proc}}\ Q\ .$$

This asserts that p is the name of a procedure with body Q and with formal parameters $\underline{x}$, $\underline{y}$.

The notation for a procedure call is similarly extended:

$$\underline{\text{call}}\ p(\underline{a}) : (\underline{e})\ .$$

This is a call of procedure p with actual parameters $\underline{a}$, $\underline{e}$ corresponding to the formals $\underline{x}$, $\underline{y}$, where $\underline{a}$ is a list of variable names, and $\underline{e}$ is a list of expressions; and $\underline{a}$ is the same length as $\underline{x}$, and $\underline{e}$ is the same length as $\underline{y}$.

As before, we assume that P{Q}R has been proved of the body Q . Consider the call

$$\underline{\text{call}}\ p(\underline{x}) : (\underline{v})$$

in which the names of the formal parameters have been "fed back" as actual parameters, thus effectively turning the procedure back into a parameterless one. It is fairly obvious that this call has the same effect as the execution of the procedure body itself; thus we obtain the rule

$$\frac{p(\underline{x}) : (\underline{v})\ \underline{\text{proc}}\ Q\ ,\ P\{Q\}R}{P\ \{\underline{\text{call}}\ p(\underline{x}) : (\underline{v})\}\ R}\ .$$ Rule of Invocation

Of course, this particular choice of actual parameters is most unlikely to occur in practice; nevertheless, it will appear later that the rule of invocation is a useful basis for further advance.

Consider next the more general call

$$\underline{\text{call}}\ p(\underline{a}) : (\underline{e})\ .$$

This call is intended to perform upon the actual parameters $\underline{a}$ and $\underline{e}$ exactly the same operations as the body Q would perform upon the formal parameters $\underline{x}$ and $\underline{v}$. Thus it would be expected that $R\ \frac{\underline{x}, \underline{v}}{\underline{a}, \underline{e}}$ would be true after execution of the call, provided that the corresponding precondition $P\ \frac{\underline{x}, \underline{v}}{\underline{a}, \underline{e}}$ is true before the call. This reasoning leads to the rule

$$\frac{P\ \{\underline{\text{call}}\ p(\underline{x}) : (\underline{v})\}\ R}{P\ \frac{\underline{x}, \underline{v}}{\underline{a}, \underline{e}}\ \{\underline{\text{call}}\ p(\underline{a}) : (\underline{e})\}\ R\ \frac{\underline{x}, \underline{v}}{\underline{a}, \underline{e}}}\ .$$ Proposed Rule of Substitution

Unfortunately, this rule is not universally valid. If the actual parameter list $\underline{a}, \underline{e}$ contains the same variable more than once, the proof of the body of the subroutine is no longer valid as a proof of the correctness of the call. This may be shown by a trivial counter example, contained in Table 1. In order to prevent such contradictions, it is necessary to formulate the conditions that all variables in $\underline{a}$ are distinct, and none of them is contained in $\underline{e}$. We shall henceforth insist that every procedure call satisfy these readily tested conditions, and thus reestablish the validity of the rule of substitution. In a programming language standard, we shall see later that there are other reasons for leaving undefined the effect of a procedure call which fails to satisfy the conditions.

Counterexample

Assume:	p(x) : (v) proc x: = v + 1		(1)
	v + 1 = v + 1 {x: = v + 1} x = v + 1	[Assignment]	(2)
	true ⊢ v + 1 = v + 1	[Logical theorem]	(3)
From 2, 3:	true {x: = v + 1} x = v + 1	[Consequence]	(4)
From 1, 4:	true {call p(x) : (v)} x = v + 1	[Invocation]	(5)
From 5:	true {call p(a) : (a)} a = a + 1	[Substitution]	(6)

Since the conclusion is an obvious contradiction, we must prohibit calls of the form call p(a) : (a) .

TABLE 1

As an example of the successful use of the rule of substitution, assume that a declaration has been made,

$$\text{random } (k) : (m , n) \text{ proc } Q ,$$

where Q is a procedure body of which it has been proved

$$m \leq n \ \{Q\} \ m \leq k \leq n .$$

The rule of invocation permits deduction of

$$m \leq n \ \{\text{call random } (k) : (m , n)\} \ m \leq k \leq n ,$$

and applying the rule of substitution to a particular call, it is possible to obtain

$$1 \leq q + 1 \ \{\text{call random } (r) : (1 , q + 1)\} \ 1 \leq r \leq q + 1 .$$

In some cases it is necessary to use a slightly more powerful rule of substitution. Suppose P and/or R contain some variables $\underline{k}$ which do not occur in $\underline{x}$ or $\underline{y}$, but which happen to occur in $\underline{a}$ or $\underline{e}$. In such a case it is necessary first to substitute some entirely fresh variables, $\underline{k}'$ for $\underline{k}$ in P and R, before applying the rule given above. This is justified by a more powerful version of the rule

$$\frac{P\ \{\underline{\text{call}}\ p(\underline{x}) : (\underline{v})\}\ R}{P^{\underline{k},\underline{x},\underline{v}}_{\underline{k}',\underline{a},\underline{e}}\ \{\underline{\text{call}}\ p(a) : (e)\}\ R^{\underline{k},\underline{x},\underline{v}}_{\underline{k}',\underline{a},\underline{e}}}\ .$$

Rule of Substitution

DECLARATION

In most procedures it is highly desirable to declare that certain of the variables are local to the procedure, or to some part of it, and that they are to be regarded as distinct from any other variables of the same name which may exist in the program. Since local variables do not have to be included in parameter lists, a considerable simplification in the structure of the program and its proof may be achieved. We will introduce the notation for declarations,

$$\underline{\text{begin}}\ \underline{\text{new}}\ x\ ;\ Q\ \underline{\text{end}}\ ,$$

where x stands for the declared variable identifier, and Q is the program statement (scope) within which the variable x is used; or, in ALGOL terms, the block to which it is local.

The effect of a declaration is merely to introduce a new working variable, and its introduction is not intended to have any effect on any of the other variables of the program; it cannot therefore affect the truth of any provable assertion about these variables.

Thus in order to prove

$$P\ \{\underline{\text{begin}}\ \underline{\text{new}}\ x\ ;\ Q\ \underline{\text{end}}\}\ R\ ,$$

all that is in principle necessary is to prove the same property of the body of the block, namely,

$$P\{Q\}R\ .$$

However, this rule is not strictly valid if the variable x happens to occur in either of the assertions P or R. In this case, the validity of the rule can be reestablished by first replacing every occurrence of x in Q by some entirely fresh variable y, which occurs neither in P, Q, nor R. It is a general property of declarations that such a systematic substitution can have no effect on the

meaning of the program. Thus the rule of declaration takes the form

$$\frac{P\ \{Q^x_y\}\ R}{P\ \{\underline{new}\ x\,;\,Q\}\ R}\ , \qquad \text{Rule of Declaration}$$

where y is not free in P or R, nor does it occur in Q (unless y is the same variable as x).

In practice it is convenient to declare more than one variable at a time, so that the rule of declaration needs to be strengthened to apply to lists $\underline{x}$ and $\underline{y}$ rather than single variables.

RECURSION

The rules of inference given above are not sufficient for the proof of the properties of recursive procedures. The reason is that the body Q of a recursive procedure contains a call of itself, and there is no way of establishing what are the properties of this recursive call. Consequently, it is impossible to prove any properties of the body Q. This means that it is impossible to use even the simple rule of invocation,

$$\frac{\underline{p}\ \underline{proc}\ Q\ ,\ P\{Q\}R}{P\ \{\underline{call}\ p\}\ R}\ ,$$

since the proof of the second premise $P\{Q\}R$ remains forever beyond our grasp.

The solution to the infinite regress is simple and dramatic: to permit the use of the desired conclusion as a hypothesis in the proof of the body itself. Thus we are permitted to prove that the procedure body possesses a property, on the assumption that every recursive call possesses that property, and then to assert categorically that every call, recursive or otherwise, has that property. This assumption of what we want to prove before embarking on the proof explains well the aura of magic which attends a programmer's first introduction to recursive programming.

In formal terms, the rule of invocation for a recursive procedure is

$$\frac{\underline{p}(\underline{x})\ :\ (\underline{v})\ \underline{proc}\ Q,\quad P\{\underline{call}\ \underline{p}(\underline{x})\ :\ (\underline{v})\}\ R \vdash P\ \{Q\}R}{P\ \{\underline{call}\ \underline{p}(\underline{x})\ :\ (\underline{v})\}\ R} \qquad \text{Rule of Recursive Invocation}$$

Unfortunately, this relatively simple rule is not adequate for the proof of the properties of recursive procedures. The reason is that it gives no grounds for supposing that the local variables of the procedure (other than those occurring in the left hand parameter list) will remain unchanged during a recursive call. What is required is a rather more powerful rule which permits the assumed properties of a recursive call to be adapted to the particular circumstances of that call. The formulation of a rule of adaptation is designed in such a way as to permit a mechanically derived answer to the question, "If S is the desired result of executing a procedure call, call $p(\underline{a}) : (\underline{e})$, and $P\ \{\text{call } p(\underline{a}) : (\underline{e})\}\ R$ is already given, what is the weakest precondition W such that $W\ \{\text{call } p(\underline{a}) : (\underline{e})\}\ S$ is universally valid?"

It turns out that this precondition is

$$\exists\, \underline{k}(P \wedge \forall \underline{a}\ (R \supset S)) \ ,$$

where $\underline{k}$ is a list of all variables free in P, R but not in $\underline{a}, \underline{e}$, or S. This fact may be formalized

$$\frac{P\ \{\text{call } p(\underline{a}) : (\underline{e})\}\ R}{\exists\, \underline{k}(P \wedge \forall \underline{a}\ (R \supset S))\ \{\text{call } p(a) : (e)\}\ S\ .} \qquad \text{Rule of Adaptation}$$

In the case where $\underline{k}$ is empty, it is understood that the $\exists$ will be ommitted.

The rule of adaptation is also extremely valuable when applied to non-recursive procedures, since it permits a single proof of the properties of the body of a procedure to be used again and again for every call of the procedure. In the absence of recursion, the rule of adaptation may be justified as a derived inference rule, since it can be shown that every theorem proved with its aid could also have been proved without it. However, successful use of the rule of adaptation to simplify proofs still depends on observance of the conditions of the disjointness of actual parameters.

Example

As an example of the application of the rules given above, we will take the trivial, but familiar, problem of the computation of the factorial $\underline{r}$ of a

non-negative integer a . The procedure is declared:

```
fact (r) : (a) proc
     if a = 0 then r: = 1
          else begin new w ;
               call fact (w) : (a - 1) ;
               r: = axw
          end .
```

It is required to prove that

$$a \geq 0 \ \{\underline{\text{call}}\ \text{fact}\ (r) : (a)\}\ r = a! \quad \ldots\ (I)\ .$$

This is achieved by proving

$$a \geq 0\ \{Q\}\ r = a!$$

where Q stands for the body of the procedure, on the hypothesis that (I) already holds for the internal recursive call. The proof is given in Table 2; it contains a number of lemmas which can be readily proved as theorems in the arithmetic of integers. A list of inference rules used is contained in the Appendix.

Proof of Factorial Program

1.	$axw = a!\ \{r: = axw\}\ r = a!$	D0
2.	$a - 1 \geq 0 \wedge \forall w(w = (a-1)! \supset axw = a!)\{\underline{\text{call}}\ \text{fact}\ (w): (a-1)\}axw = a!$	D6, D7, Hypothesis
3.	$a > 0 \vdash a - 1 \geq 0 \wedge \forall w\ (w = (a-1)! \supset axw = a!)$	Lemma 1
4.	$a > 0\ \{\underline{\text{begin}}\ \underline{\text{new}}\ w\ ;\ \underline{\text{call}}\ \text{fact}\ (w) : (a-1)\ ;\ r: = axw\ \underline{\text{end}}\}\ r = a!$	D1, D2, D8 (1, 2, 3)
5.	$1 = a!\ \{r: = 1\}\ r = a!$	D0
6.	$\underline{\text{if}}\ \ a = 0\ \ \underline{\text{then}}\ \ 1 = a!\ \ \underline{\text{else}}\ \ a > 0\ \{Q\}\ r = a!$	D4 (5, 4)
7.	$a \geq 0 \vdash \underline{\text{if}}\ \ a = 0\ \ \underline{\text{then}}\ \ 1 = a!\ \ \underline{\text{else}}\ \ a > 0$	Lemma 2
8.	$a \geq 0\ \{Q\}\ r = a!$	D1 (7, 6)
9.	$a \geq 0\ \{\underline{\text{call}}\ \text{fact}\ (r) : (a)\}\ r = a!$	D5 (8)

TABLE 2

IMPLEMENTATION

It has been suggested by Floyd (1967 a) that a specification of proof techniques for a language might serve well as a formal definition of the semantics of that language, for it lays down, as an essential condition on any computer implementation, that every provable property of any program expressed in the language shall, in practice, obtain when the program is executed by the implementation. It is, therefore, interesting to inquire what implementation methods for parameter passing will be valid in a language whose definition includes the inference rules described in the previous sections. It appears that there is a wide choice of valid implementation methods, covering all the standard methods which have been used in implementations of widely familiar languages.

This means that each implementor of the language defined by these rules can make his selection of method in order to maximize the efficiency of his implementation, taking into account not only the characteristics of his particular machine, but also varying his choice in accordance with the properties of each particular procedure and each particular type of parameter; and he is free to choose the degree of automatic optimization which he will bring to bear, without introducing any risk that an optimized program will have different properties from an unoptimized one.

The remainder of this section surveys the various familiar parameter mechanisms, and assesses the circumstances under which each of them gives the highest efficiency. The reader may verify that they all satisfy the requirements imposed by the proof rules stated above, bearing in mind that every procedure call,

$$\underline{\text{call}}\ p(\underline{a}):(e)\ ,$$

conforms to the condition that all the parameters $\underline{a}$ are distinct from each other, and none of them appears in any of the expressions $\underline{e}$. The observance of this restriction is the necessary condition of the validity of most of the commonly used parameter passing methods.

1. Compile-time macro-substitution. Macro-substitution is an operation which replaces all calls of a procedure by a copy of the body of the procedure, after this copy has itself been modified by replacing each occurrence of a formal parameter

within it by a copy of the corresponding actual parameter. The normal process of translation to machine code takes place only after these replacements are complete. This technique has been comonly used for Assembly Languages and for Business Oriented Languages.

Macro-substitution will be found to be a satisfactory technique in any of the following circumstances:

(1) The procedure body is so short that the code resulting from macro-substitution is not appreciably longer than the parameter planting and calling sequence would have been.

(2) The procedure is called only once from within the program which incorporates it.

(3) The procedure is called several times, but on each occasion some of all of the parameters are identical. Substitution can be applied only to those parameters which are identical, leaving the remaining parameters to be treated by some run-time mechanism.

(4) In a highly optimizing compiler, macro-substitution will ensure not only that no time is wasted on parameter passing, but also that each call can be fully optimized in the knowledge of the identity and properties of its actual parameters.

The technique is not applicable to recursive procedures.

2. Run-time code construction. An alternative to substitution in the source code at compile time is the execution of a logically identical operation on the object code at run time. This involves planting the addresses of the actual parameters within the machine code of the procedure body on each occasion that the procedure is called. The technique may be favored whenever all the following conditions are satisfied:

(1) The computer has an instruction format capable of direct addressing of the whole store.

(2) The actual parameter is an array (or other large structure) which would be expensive to copy.

(3) The called procedure is not recursive.

(4) The called procedure contains at least one iteration.

This technique was used in FORTRAN implementations on the IBM 704/709 series of computers.

3. Indirect addressing. Before jumping to the procedure body, the calling program places the addresses of the actual parameters in machine registers or in the local workspace of the called procedure. Whenever the procedure refers to one of its parameters, it uses the corresponding address as an indirect pointer (modifier).

This technique is suitable in the following circumstances:

(1) The computer has an instruction format with good address-modification facilities.

(2) The actual parameter is an array (or other large structure) which would be expensive to copy.

If a single parameter mechanism is to be used in all circumstances, this is undoubtedly the correct one. However, on fast computers with slave stores, operand pre-fetch queues, or paging methods of storage control, it could cause some unexpected inefficiencies.

This technique is used in PL/1 and in many implementations of FORTRAN on the IBM 360 series of computers.

4. Value and result. Before jumping to the subroutine, the calling program copies the current values of the actual parameters into machine registers or local workspace of the called subroutine. After exit from the subroutine, the calling program copies back all values that might have been changed (i.e., those to the left of the colon in the actual parameter list).

This technique is to be preferred when either of the following conditions hold:

(1) The size of the parameter is sufficiently small that copying is cheap, accomplished in one or two instructions.

(2) The actual parameter is "packed" in such a way that it cannot readily be accessed by indirect addressing.

This technique is available in ALGOL W, and is used in several current implementations of FORTRAN.

5. Call by name (as in ALGOL 60). The calling program passes to the procedure the addresses of portions of code corresponding to each parameter. When the procedure wishes to access or change the value of a parameter, it jumps to the corresponding portion of code.

Since the restrictions on the actual parameters prohibit the use of Jensen's device, there is no reason why this technique should ever be used. It is difficult to envisage circumstances in which it could be more efficient than the other techniques listed.

CONCLUSION

It has been shown that it is possible by axiomatic methods to define an important programming language feature in such a way as to facilitate the demonstration of the correctness of programs and at the same time to permit flexibility and high efficiency of implementation. The combination of these two advantages can be achieved only if the programmer is willing to observe certain disciplines in his use of the feature, namely that all actual parameters which may be changed by a procedure must be distinct from each other, and must not be contained in any of the other parameters. It is believed that this discipline will not be felt onerous by programmers who are interested in the efficient and reliable solution of practical problems in a machine-independent fashion.

It is interesting to note that the discipline imposed is combination of the disciplines required by the ISO standard FORTRAN and by the IFIP recommended subset of ALGOL 60. The former insists on the distinctness of all parameters changed by a procedure, and the latter insists that each of them be an unsubscripted identifier.

Acknowledgement

The basic approach adopted in this paper was stimulated by an investigation reported in Foley (1969).

Appendix

P, P_1, P_2, R, S stand for propositional formulae.

Q, Q_1, Q_2 stand for program statements.

x, y stand for variable names (y not free in P or R).

e stands for an expression.

B stands for a Boolean expression.

p stands for a procedure name.

$\underline{x}$ stands for a list of non-local variables of Q which are subject to change in Q.

$\underline{v}$ stands for a list of other non-local variables of Q.

$\underline{a}$ stands for a list of distinct variables.

$\underline{e}$ stands for a list of expressions, not containing any of the variables $\underline{a}$.

$\underline{k}$ stands for a list of variables not free in $\underline{x}, \underline{v}$.

$\underline{k}'$ stands for a list of variables not free in $\underline{a}, \underline{e}, S$.

D0 $R_e^x \{x := e\} R$ — Assignment

D1 $\dfrac{P \{Q\} S, S \vdash R}{P \{Q\} R} \quad \dfrac{P \vdash S, S \{Q\} R}{P \{Q\} R}$ — Consequence

D2 $\dfrac{P \{Q_1\} S, S \{Q_2\} R}{P \{Q_1 ; Q_2\} R}$ — Composition

D3 $\dfrac{P \{Q\} S, S \vdash \underline{\text{if}}\ B\ \underline{\text{then}}\ P\ \underline{\text{else}}\ R}{P \{\underline{\text{while}}\ B\ \underline{\text{do}}\ Q\} R}$ — Iteration

D4 $\dfrac{P_1 \{Q_1\} R, P_2 \{Q_2\} R}{\underline{\text{if}}\ B\ \underline{\text{then}}\ P_1\ \underline{\text{else}}\ P_2 \{\underline{\text{if}}\ B\ \underline{\text{then}}\ Q_1\ \underline{\text{else}}\ Q_2\} R}$ — Alternation

D5 $\dfrac{p(\underline{x}) : (\underline{v})\ \underline{\text{proc}}\ Q, P \{\underline{\text{call}}\ p(\underline{x}) : (\underline{v})\} R \vdash P \{Q\} R}{P \{\underline{\text{call}}\ p(\underline{x}) : (\underline{v})\} R}$ — Recursion

D6 $\dfrac{P \{\underline{\text{call}}\ p(\underline{x}) : (\underline{v})\} R}{P^{\underline{k}, \underline{x}, \underline{v}}_{\underline{k}', \underline{a}, \underline{e}} \{\underline{\text{call}}\ p(\underline{a}) : (\underline{e})\} R^{\underline{k}, \underline{x}, \underline{v}}_{\underline{k}', \underline{a}, \underline{e}}}$ — Substitution

D7 $\dfrac{P \{\underline{\text{call}}\ p(\underline{a}) : (\underline{e})\} R}{\exists\, \underline{k}'(P \wedge \forall \underline{a}(R \supset S))\ \{\underline{\text{call}}\ p(\underline{a}) : (\underline{e})\} S}$ — Adaptation

D8 $\dfrac{P \{Q_y^x\} R}{P \{\underline{\text{new}}\ x ; Q\} R}$ (where y is not in Q unless y and x are the same) — Declaration