A GENERAL PROOF RULE FOR PROCEDURES IN PREDICATE TRANSFORMER SEMANTICS

Alain J. Martin
Computer Science
California Institute of Technology

5075:TR:83

The research described in this paper was sponsored by the Defense Advanced Research Projects Agency, ARPA Order number 3771, and monitored by the Office of Naval Research under contract number N00014-79-C-0597

A General Proof Rule For Procedures In Predicate Transformer Semantics

Alain J. Martin
Computer Science
California Institute Of Technology

Abstract Given a general definition of the procedure call based on the substitution rule for assignment, a general proof rule is derived for procedures with unrestricted value, result, and value-result parameters, and global variables in the body of the procedure. It is then extended for recursive procedures. Assuming that it has been proved that the body establishes a certain postcondition I, the "intention", for a certain precondition J, the proof rule permitting to determine under which conditions a certain procedure call establishes the postcondition E, the "extension", is based on finding an "adaptation" A, as weak as possible, such that A ∧ I ==> E'. (E' is derived from E by some substitution of parameter variables.) It is preferable, but not essential, that the body be "transparent" for the value parameters, i.e., that the value parameters are not changed by the body.

Pasadena, 9 February 1983

Introduction

In programming, the procedure mechanism fulfills two roles. First, it allows recursion; second, it provides a general abstraction mechanism. It is the procedure as an abstraction mechanism that is studied here, although the results will be used to prove properties of recursive programs as well.

Apart from making recursion possible, the procedure mechanism, as we define it, does not introduce a new semantic concept - a new "predicate

transformer". It permits to define arbitrary primitives whose semantics, unlike those of usual primitives, are only a partial function. More precisely, the "procedure declaration" mechanism permits to construct a program part - the "procedure" - of an arbitrary complexity. This procedure is designed for a certain intended net-effect: it is specified how the procedure establishes a certain postcondition, but not how it establishes any postcondition. The "procedure call" mechanism permits to use the procedure as a primitive, i.e., by knowing only what it does, and ignoring how it does it. Hence, the main issue with procedures is how to deal with partial semantic specification.

Although we shall couch the procedure paradigm in the notation of an imperative programming language, the concepts of abstraction and of partial specification that this paradigm embodies are quite general, and can be encountered in several forms (modules, components, VLSI-cells, etc...) in many other disciplines. We believe that the different properties we describe (transparency of the procedure body, adaptation of a specification to its use), and the proof rule we propose, are also applicable in these other disciplines where an equivalent notion of abstraction is used.

1. Definitions

We use Dijstra's guarded command language, and Dijkstra's predicate transformer semantics [1] (preferably extended with a fixed-point definition for the semantics of the repetition). We are thus interested in total correctness properties of programs.

Consider the procedure declaration

$$\underline{\text{proc}}\ p(x?,\ y?!,\ z!);\ S. \qquad (0)$$

It defines a procedure named p, whose "procedure body" is the - here unspecified - program part S. The variables x,y,z listed between parentheses are the "formal parameters" of the procedure. They are local variables of S that have been given a special status.

In general, S will contain other, usual local variables. S will also contain global variables (we use the terms "local" and "global" in the Algol 60 sense). For the sake of clarity, we shall first assume that the procedure body does not contain global variables. We shall then show how global variables in the procedure body can be added without difficulty.

The parameters postfixed with ? - x in the above example - are called "input" or "value" parameters. The parameters postfixed with ! - z in the above example - are called "output" or "result" parameters. The parameters postfixed with ?! - y in the above example - are called "input-output" or "value-result" parameters.

By definition, the "procedure call" p(a,b,c) is identical to the program part

$$x,y := a,b;\ S;\ b,c := y,z\ . \qquad (1)$$

The variables a,b, and c are the "actual parameters" corresponding to the formal parameters x,y, and z, respectively. This definition allows expressions as actual input parameters. It excludes procedures as parameters.

In this presentation, we shall simplify notations by allowing the formal and actual parameter identifiers to stand for multiple variables. For instance, x may stand for the list of simple variables x1, x2, ..., xn. In such a case, the corresponding actual parameter variable - here, a - must stand for a list of the same length - here, a1, a2, ..., an-.

We consider solved the problem of binding the identifiers a,b,c of the actual parameters to a particular declaration in the (static or dynamic) scope of the procedure call. We assume that the actual parameters are uniquely defined upon procedure call. And we allow no side effect in the evaluation of expressions, which are assumed to be everywhere defined.

From (1) we deduce the general semantic definition of the procedure call with input/output parameters:

$$(\forall Q::wp(p(a,c,b),Q) = (wp(S,Q^{b,c}_{y,z}))^{x,y}_{a,b}) \qquad (2)$$

Substitution rule

The notation P^u_l denotes the predicate derived from P by substituting l - the "lower variable" - for all free occurrences of u - the "upper variable" -. If u and l are (necessarily matching) multiple variables, a "simultaneous" substitution of elementary l's for elementary u's is performed. If all upper variables are different, the substitution is unique. If several upper variables are identical, P^u_l is defined as the conjunction of all predicates obtained by performing the substitutions of the identical upper variables in all possible orders.

Observe that in (2) the outermost substitution is unique, since all formal parameters are by definition different. But the inner substitution is not.

2. Specification of the procedure declaration

Definition (2) gives the semantics of the procedure call as a general predicate transformer, but it defeats the purpose of the procedure as an abstraction mechanism since for each procedure call, one needs to use the program text of the body S.

The problem can be defined as follows: Assume that the "designer" of the procedure has defined its intended net effect by establishing that for a certain postcondition I, called the "intention" of the procedure, the "specification" :

$$wp(S,I) = J \qquad (3)$$

holds for a certain precondition J. How can a "user" of the procedure derive from (3)

$$wp(p(a,b,c),E) \qquad (4)$$

for some predicate E, called the "extension" of the procedure at the place of the call p(a,b,c)?

First, since all local variables of S other than x,y,z are hidden from the "user," the intention I should contain only x,y,z as variables.

Second, the precondition J is independent of z.
Proof: Since a variable zi of z is local to S, it has to be initialized inside S. Hence the first statement in S involving zi is of the form zi := exp, where exp does not contain zi. Hence, whatever the postcondition of zi := exp, the precondition does not contain zi. □

Consider the procedure declaration:

proc p(x?,y!); S: x := x+1; y := 2*x ,

with the specification: wp(S,y = 2*x) = true. Without knowing the structure of S, it is impossible to determine, for instance, wp(p(a,b), b = 2*a) since it is not known in which way S modifies the input parameter x. (This wp happens to be false.)

We must therefore add to the specification (3) some information about how the procedure body "distorts" the input parameter x. But there is obviously no reason for a procedure body to modify its input parameters, and we can therefore, without loss of generality, impose that the procedure body S fulfill the

Transparency requirement: If x is an input formal parameter of S, no assignment to x, and no procedure call with x as an actual output or input-output parameter, may occur in S. □

Corollary: For S transparent, and x an input formal parameter of S

$$(\forall X::wlp(S,x=X) = (x=X)) \qquad (5)$$

holds, which is equivalent to

$$(\forall R{:}R \text{ independent of } y \text{ and } z{:}\, wlp(S,R) = R).\ \square \qquad (6)$$

(Proof omitted.)

(The transparency requirement is not necessary for the validity of the proof rule we shall now propose. The proof rule can be extended very easily to non-transparent procedures.)

3. Proof rule for procedure call

Consider the general procedure declaration (0), for which the specification (3) and the transparency of the body have been proved.

For a given extension E, and a given procedure call p(a,b,c), we want to determine a precondition pre as weak as possible such that

$$pre \Rightarrow wp(p(a,b,c),E)\ .$$

We would like to investigate under which conditions

$$(\forall x,y,z :: I \Rightarrow E') \qquad (7)$$

where E' is the predicate $E^{b,c}_{y,z}$.
In general, (7) does not hold because I is a predicate in x,y,z, whereas E' is independent of x since E is independent of x. Hence, we have to look for a predicate A, the "adaptation", such that:

$$(\forall x,y,z :: A \wedge I \Rightarrow E')\ . \qquad (8)$$

In view of our intention to apply the wp(S,)-transformation to (8), we require that A be independent of y and z, since I is the only predicate in y and z for which wp(S,I) is known.

Assume such an A has been found:

(8)

==> {properties of weakest preconditions}

$$(\forall x,y,z :: wlp(S,A) \wedge wp(S,I) \Rightarrow wp(S,E'))$$

==> {transparency of S, and A independent of y and z}

$$(\forall x,y,z :: A \wedge J \Rightarrow wp(S,E'))$$

==> {universal quantification}

$$(A \wedge J)^{x,y}_{a,b} \Rightarrow (wp(S,E'))^{x,y}_{a,b}$$

==> {definition (2)}

$$(A \wedge J)^{x,y}_{a,b} \Rightarrow wp(p(a,b,c),E) \ .$$

(Observe that the quantification $\forall z$ has disappeared because A,J, and wp(S,E') are independent of z.) Hence the

Proof rule Let A, the "adaptation", be the weakest predicate independent of y and z, such that

$$A \wedge I \Rightarrow E^{\,b,c}_{y,z} \ . \tag{9}$$

Then

$$(A \wedge J)^{x,y}_{a,b} \Rightarrow wp(p(a,b,c),E) \ . \quad \square \tag{10}$$

Remarks

1) There exist always at least two predicates A satisfying (9): the strongest: false, and the weakest: (I => E'). But the former is not very interesting, and the latter is in general not independent of y and z.

2) The closer the extension is from the intention, the weaker the adaptation will be. If I ==> E', the adaptation is true. It may also happen that extension and intention are so different that the weakest adaptation is false. In this case, the intention is said to be "nonadaptable" to the extension. It does not mean, however, that wp(p(a,b,c),E) = false, but that it is just impossible to derive it from the given intention.

3) In many cases, the adaptation need not be the weakest solution of (9). A stronger predicate may be easier to establish, and sufficient for the particular case.

4) In many cases, A will just be a relation between the different constants of I and E' - in example 4.3, for instance.

4. Examples

In all the examples, the procedure bodies are obviously transparent due to Theor. 1. It will not be specified any longer. All parameters are of type integer.

4.1 Procedure declaration:

proc inc1(x?,y!) ; y := x+1 .

Specification: wp(S,y=x+1) = true .

Establish pre(inc1(a,a), a = ao+1), as weak as possible, for a free constant ao - pre is a precondition such that pre(S,P) ==> wp(S,P). The weakest A such that

$$A \wedge y = x+1 \Longrightarrow y = ao+1$$

is $x = ao$.

Proof rule (10) gives the precondition

$$(x = ao)^{x}_{a} \wedge \underline{true}$$

i.e. $a = ao$. □

4.2 For the same procedure and the same specification, establish

$$pre(inc1(a,b),\ b = 3*a) .$$

The weakest A such that

$$A \wedge y = x+1 \Longrightarrow y = 3*a$$

is $x+1 = 3*a$.

Proof rule (10) gives the precondition

$$(x+1 = 3*a)^{x}_{a} \wedge \underline{true}$$

i.e., $2*a = 1$.

For a of type integer, the precondition is false. □

4.3 Procedure declaration:

$$\underline{proc}\ swap(x?!,y?!)\ ;\ x,y := y,x .$$

Specification:

$$wp(S, x = X \wedge y = Y) = (x = Y \wedge y = X)$$

for any free constants X and Y. Establish

$$pre(swap(a,b),\ Q(a,b))$$

for a given predicate Q, and the actual variables a and b. The weakest A such that

$$A \wedge x = X \wedge y = Y \Longrightarrow Q(a,b)$$

is $Q(X,Y)$.

Proof rule (10) gives the precondition

$$Q(X,Y) \wedge (y = X \wedge x = Y)\ \begin{matrix} x,y \\ a,b \end{matrix}$$

i.e., $Q(X,Y) \wedge b = X \wedge a = Y$

which is equivalent to Q(b,a) since X and Y are free. □

5. Global variables

Consider the procedure declared:

proc p(x?,y?!,z! ; glo u?,v?!) ; S

the body of which contains the global variables u and v. (The variables x,y,z listed before the reserved word glo are usual formal parameters.)

Among the global variables that S contains, we distinguish the ones that are not assigned to inside S (neither directly in an assignment statement nor as the actual parameters of a procedure call), from the others. We call the former input global variables, and denote them by postfixing the identifier by "?" - here, u? -; we call the latter input-output global variables, and denote them by postfixing the identifier by "?!" - here, v?!-.

By definition, the body is transparent to input global variables.

Let us consider the extension E for a call p(a,b,c,u,v). In the specification:

$$wp(S,I) = J \ ,$$

I is now a function of x,y,z,u,v; J is a function of x,y,u,v.

As adaptation A, we are now looking for the weakest predicate independent of y,z, and v, such that

$$(\forall\ x,y,z :: A \wedge I \Rightarrow E') \ .$$

The same derivation as in Section 3 leads to the same proof rule. Hence the

General proof rule: Given the procedure declaration

proc p(x?,y?!,z!; glo u?, v?!) ; S

(S transparent to x and u) and the specification

$$wp(S,I) = J \ ,$$

let A, the "adaptation," be the weakest predicate independent of y,z,and v such that

$$A \wedge I \Rightarrow E^{b,c}_{y,z}$$

for the extension E and the call p(a,b,c,u,v). Then

$$(A \wedge J)^{x,y}_{a,b} \Rightarrow wp(p(a,b,c,u,v),E). \quad \square$$

Example:

Procedure declaration:

proc incd(x?,y!; glo d?); y := x+d .

Specification: wp(S,y = x+d) = true .

Establish pre(incd(d,d,d), d=D) for a free constant D.

The weakest A such that:

$$A \wedge y = x+d \Longrightarrow y=D$$

is $x+d = D$.

The proof rule gives the precondition:

$$(x+d = D)^{x}_{d} \wedge \underline{true}$$

i.e., $2d = D$. □

Which global variables should be declared in the procedure heading cannot always be decided by a simple inspection of the program text of the body. In the case a global variable is of a structured type - array, list, etc...-, whether the whole structured variable or only some components should be declared depends on how the net effect of the body on the variable is specified. This will be illustrated by the following simple example.

Consider a procedure p whose net-effect is to swap two elements B(i) and B(j) of an array B(i:0≤i<N) of integers. B is not passed as parameters, but only i and j (input parameters). The body S is:

if $0 \leq i,j < N$ —> B(i), B(j) := B(j), B(i) fi .

We can define the specification of S as:

wp(S,B = X) = ($0 \leq i,j < N$ cand B:swap(i,j) = X)

where X is a free constant array isomorphic to B, and B:swap(i,j) is the array derived from B by swapping B(i) and B(j).

Such a specification defines the net effect of S on the whole array B. And thus B as a whole must be declared as an input-output global variable. Then, if we want to determine

$$pre(p(m,n,B),\ P(B))\ ,$$

the adaptation A may not contain B. But this will not be a problem since the intention is defined on the whole B. For

$$A \wedge (B{=}X) \Longrightarrow P(B)$$

we find $A : P(X)$,

which gives the precondition

$$(P(X) \wedge (0 \leq i,j < N \ \underline{cand}\ B{:}swap(i,j) = X))^{i,j}_{m,n}$$

i.e., $P(X) \wedge (0 \leq m,n < N \ \underline{cand}\ B{:}swap(m,n) = X)$.

Which, for a free X, is equivalent to

$$0 \leq m,n < N \ \underline{cand}\ P(B{:}swap(m,n))\ .$$

We can also define the specification of S as:

$$wp(S,(B(i),B(j) = X,Y)) = (0 \leq i,j < N \ \underline{cand}\ (B(j),B(i) = X,Y))$$

for X,Y free integer constants. Now, only the elements B(i) and B(j) need to be declared as global input-output variables. For the same call, an adaptation A satisfying

$$A \wedge (B(i),\ B(j) = X,Y) \Longrightarrow P(B)$$

is $P(B_{X,Y}^{B(i),B(j)})$.

Which leads to the precondition

$$0 \leq m,n < N \text{ cand } P(B_{B(n),Bm)}^{B(m),Bn)})$$

i.e., $0 \leq m,n < N$ cand $P(B:swap(m,n))$.

Hence, the choice of the variables in terms of which the specification of the body is expressed dictates the choice of the global variables to be declared in the procedure heading. (This is not a problem since the declaration of global variables is for specification purposes only.)

6. Proof rule for recursive procedures

Consider a recursive procedure "rec" for which we want to prove that the specification

$$SP : wp(S,I) = J$$

holds for the body S. Since S contains at least one call of rec, we need to use SP in order to establish the truth of SP. This obviously calls for mathematical induction.

Let us use the most general definition of mathematical induction, based on well-founded sets, and sometimes called "Noetherian induction". (We borrow the following formulation from [2]. An equivalent one can be found in [6] under the name "structural induction".)

From the input formal parameters, and possibly some constants, of S, let us construct a well-founded set U, i.e., a set with a partial order relation (<), such that any descending chain u0 > u1 > ... is of finite length. Then

$$(\forall u:u\varepsilon U:SP(u)) \equiv (\forall u:u\varepsilon U:SP(u) \text{ v } (\exists v:v<u:\neg SP(v))) \qquad (11)$$

(SP(u) means "SP considered as a predicate in u.") Hence, in order to prove that the specification SP holds, we can prove the right-hand side of (11). But that is a predicate on predicate transformers, which makes it unpractical to manipulate. We shall therefore partition the proof in two:

I) The "base step":

Prove SP(um) for any minimal element um of U

II) The "induction step":

Prove that if SP(v) holds for all $v < k$, for an arbitrary constant k, then SP(v) holds for v=k. For this part of the proof, we have to determine (at least once) pre(rec,E) for a certain postcondition E. (The well-founded set has been chosen in such a way that that is not necessary for the base step.)

Applying the proof rule, we have to determine an A independent of y and z such that

$$A \wedge I \Longrightarrow E' .$$

But in order to apply the wp(S,)-transformation to the above relation, we must assume that $v < k$ holds for this instance of S. Hence the above relation has to be extended as:

$$(v < k) \wedge A \wedge I \Longrightarrow E'$$

which gives the precondition

$$((v < k) \wedge A \wedge J)^{\,x,y}_{\,a,b} .$$

This minor extension of the proof rule makes it possible to prove specifications of recursive procedures.

7. Example

(The example chosen, McCarthy's "91-function", is quite artificial, and the proof rule is applied in quite a "brute-force" way. Most likely, another example will be used in the final version. But this one presents the advantage of being far from trivial, and yet with a simple program and parameter structure.)

Consider the procedure declaration

```
proc p(x?,y!),
S:begin if x > 100 —> y := x-10
       □ x ≤ 100 —> T:begin y':local ;
                          p(x+11,y'); p(y',y)
                          end
       fi
end.
```

We want to prove that

$$wp(S,I) = true \tag{13}$$

holds, I being the predicate

$$x > 100 \wedge y = x-10 \quad \vee \quad x \leq 100 \wedge y = 91 \ .$$

From the semantics of the alternative construct and a little predicate calculus, we derive:

$$wp(S,I) = x > 100 \vee wp(T,y=91)$$

Hence, proving (13) is equivalent to establishing the truth of

$$x > 100 \vee wp(T,y=91) \ . \tag{14}$$

For $x > 100$, (14) obviously holds. We shall prove that if (14) holds for $x > k$, it holds for $x=k$. In order to evaluate pre(T,y=91), we proceed backwards and first evaluate pre(p(y',y), y=91). The adaptation A must be such that

$$A \wedge I \Rightarrow y=91$$

(The same identifier y is used for the actual and formal parameter.)
Which gives for A : $x \leq 101$.

The proof rule now gives the precondition:

$$(x > k \wedge x \leq 101)\ {}^{x}_{y'},$$

i.e.: $(k < y' \leq 101)$.

The second step is now to evaluate

$$pre(p(x+11,y'),\ (k < y' \leq 101)) .$$

The adaptation A must be such that

$$A \wedge I \Rightarrow k < y \leq 101 ,$$

which gives for A :

$$k < x-10 \wedge 100 < x \leq 111 \vee k < 91 \wedge x \leq 100.$$

The proof rule gives the precondition

$$(x > k \wedge A)\ {}^{x}_{x+11}$$

which is, after simplification

$$k < x+1 \wedge 89 < x \leq 100 \vee k < 91 \wedge k-11 < x \leq 89. \qquad (15)$$

On the other hand

$$x = k \wedge x \leq 100$$

$\equiv$ {predicate calculus}

$$x = k \wedge 89 < x \leq 100 \vee x = k \wedge x \leq 89$$

$\Longleftarrow$ {arithmetic}

$$k < x+1 \wedge 89 < x \leq 100 \vee k < 91 \wedge k-11 < x \leq 89$$

$\Longleftarrow$ {since (15) $\Longrightarrow$ wp(T,y=91)}

$$wp(T,y=91).$$

Hence, $x=k \Longleftarrow (x > 100 \vee wp(T,y=91))$. □

8. Concluding remarks

As we said before, the transparency requirement for the procedure body can be removed. If one adds to the specification of the body some extra information on how the body "distorts" the input parameters - by computing wlp(S,x=X), for input parameters x and a free constant X -, the extended proof rule for the case the adaptation A is transformed by the body is of the form $(A' \wedge J)^{x,y}_{a,b}$ with $A' \Longrightarrow wlp(S,A)$.

The proof rule can also easily be extended to procedures with reference parameters and to functions. We can easily prove that reference parameters are eqiuvalent to input-output parameters provided that all actual reference parameters are different from each other and from other actual parameters and global variables. With these restrictions the rule can be applied to reference parameters as if they were input-output parameters. And it is easy to transform functions into procedures in the way we have done it for the 91-function.

Hence, the proof rule proposed is quite general, since it allows unrestricted use of input and output parameters with or without global variables in the body. It allows a restricted form of reference parameters, and can deal with recursive procedures and functions.

The most closely related work is C.A.R. Hoare's adaptation rule as proposed in [5], and later refined by David Gries in [3] and [4]. Translated into our framework the Hoare-Gries rule consists in taking as adaptation A the predicate (∀x,y,z::I⟹E'). We have found this rule difficult to apply even in simple cases.

Observe that the only postulate in this paper is (2), giving the general semantic definition of the procedure call. The proof rule itself is proved to be valid. Since (2) is based on the substitution rule for the assignment statement, the inclusion of our proof rule in a logical system already containing assignments (or other implementations of the substitution rule) does not add new issues concerning the logical soundness of the system.

References

[1] Dijkstra, E.W., A Discipline of Programming. Prentice-Hall, 1976

[2] Dijkstra, E.W., On Mathematical Induction. EWD803-22, Nov. 1981

[3] Gries, D. and G. Levin, Assignment and Procedure Call Proof Rules, TOPLAS 2 (Oct. 1980), 564-579

[4] Gries, D. The Science of Programming, Springer-Verlag, 1981

[5] Hoare, C.A.R. Procedures and Parameters: An Axiomatic Approach. In Symposium on Semantics of Programming Languages. Springer-Verlag, 1971, 102-116.

[6] Manna, Z. Mathematical Theory of Computation. McGraw-Hill, 1974