Proofs of Procedures

Mark Ardis
Rose-Hulman Institute of Technology
September 2000

So far we have looked at rules for describing the behavior of simple program statements: assignment, if-then-else, and while loops. Real programs also include procedures and functions, which introduce local scopes and parameter-passing mechanisms. This handout describes the rules we need to reason about procedures. It borrows heavily from [Hoare 71], a paper that extended invariant assertions to handle procedures.

One of the key ideas we need to preserve about procedures is their reuse. That is, just as a programmer only writes a procedure once and then reuses it several times, we want to reason about a procedure definition only once and then reuse that work each time we encounter a call to the procedure. The rules we will use are designed to do that.

Rules of Inference

Just as we did with simpler program statements we define inference rules to use with each type of program construct. For procedures we will have more than one rule, as some simple procedures are easier to reason about than others. We will need rules to handle parameter passing and local variables as well.

Here is the first rule:

Statement	**Rule**	**Informal Meaning**
Procedure with no parameters	p() { S } /* P */ S /* R */ **implies** /* P */ p() /* R */	Whatever is proved about the body of a procedure with no parameters is true about a call to that procedure.

The no-parameter rule allows us to reason about simple procedures that take no parameters. The rule has two preconditions: 1) a procedure `p` has been declared with no parameters and with body `S` (a sequence of statements), and 2) some proof has been

constructed about the body `S` . The conclusion is that the same pre- and post-conditions as were found for `S` can be used for any procedure call `p()` .

For example, suppose we defined a procedure `dumb` as follows:

```
dumb () {
   x = 10;
}
```

and then we proved the following about the procedure body:

```
        /*  True  */
x = 10;
        /*  x == 10  */
```

We could use that proof whenever procedure `dumb` was called:

```
        /*  True  */
dumb()
        /*  x == 10  */
```

Although this rule will probably not be used very often, it illustrates the pattern that will be used for the more interesting procedure rules.

The next rule introduces parameters, but assumes that calls to the procedure will use the same names for actual parameters as were used for formal parameters.

Statement	**Rule**	**Informal Meaning**

Procedure with actuals equal to formals	`p(x1, ... xn) {  S  }`
	
	`/* P */`
	`S`
	`/* R */`
	
	`implies`
	
	`/* P */`
	`p(x1, ... xn)`
	`/* R */`
		Whatever is proved about the body of a procedure is true about a call to that procedure that uses the same names for parameters.

If we had defined the `dumb` procedure a little differently:

```
dumb (x) {
   x = 10;
}
```

and proved the following about its body:

```
         /*  True  */
x = 10;
         /*  x == 10  */
```

then we could use that proof whenever we encountered a call to `dumb` that used the same name for its parameter:

```
         /*  True  */
dumb(x)
         /*  x == 10  */
```

Now we consider procedure calls that use different parameters.

Statement	**Rule**	**Informal Meaning**

Substitution	`/* P */`
	`p(x1, ..., xn)`
	`/* R */`
	
	`implies`
	
	`/* P(x/a) */`
	`p(a1, ... an)`
	`/* R(x/a) */`
		Whatever has been proved about a call to a procedure is true about a different call, as long as the parameters are substituted consistently in the pre- and post-conditions.

This rule assumes that substitution of names does not cause any conflicts. We shall soon see how to guarantee that.

Here is another example:

```
sum( tot | x, y) {
   tot = x + y;
}
```

The bar symbol ("|") in the declaration of parameters separates modified parameters (call by reference) from unmodified parameters (call by value). The modified parameters come first in the list.

Suppose we encountered a call to `sum` and wished to calculate its weakest pre-condition:

```
sum(c | a, b)
                /*  a == 2, b == 3, c == 5  */
```

First we would reason about a call to `sum` where the actual parameters were the same as the formals:

```
                /* x == 2, y == 3  */
tot = x + y;
                /*  x == 2, y == 3, tot == 5  */
```

implies

```
                /*  x == 2, y == 3
sum(tot | x, y)
                /*  x == 2, y == 3, tot == 5  */
```

Next we reason about a call to `sum` that substitutes different names for actual parameters.

```
                /*  a == 2, b == 3  */
sum( c | a, b)
                /*  a == 2, b == 3,  c == 5  */
```

Invalid Substitutions

Let us turn now to cases where substitution of parameters causes problems. The first is *aliasing*. Aliasing occurs when two names exist for the same program variable. This causes problems when one of those references is updated. For example, suppose we attempted the following call to `sum` :

```
sum(a | a, b)
```

If we blindly substitute parameter names in the proof we constructed earlier we get:

```
                /*  a == 2, b == 3  */
sum(a | a, b)
                /*  a == 2, b == 3, a == 5  */
```

Clearly, the post-condition is impossible. We eliminate this possibility by stating a rule about parameter substitutions:

Rule	**Informal Meaning**
No aliasing: Actual parameters that may be modified in a procedure must be distinct and may not appear in the values of parameters.	This rule prevents one parameter from acquiring more than one name (an alias).

Another way in which substitutions can cause problems is when global variables are used in the pre- and post-conditions and in the list of actual parameters. Let us modify our earlier proof about calls to the procedure `sum` so that it includes references to a global variable `g` :

```
                /* x  == g, y == 3  */
sum(tot | x, y)
                /*  x == g, y == 3, tot == g + 3  */
```

In this version we do not know the exact value of `x` , we only know that it is equal to the value of the global variable `g` . This is a legitimate statement about the behavior of the procedure.

But, consider what happens if we now try to reason about a call to `sum` that includes `g` as a parameter:

```
                /*  a == g, b == 3  */
sum(g | a, b)
                /*  a == g, b == 3, g == g + 3  */
```

Again, the post-condition is impossible. We eliminate this possibility by stating another rule about parameter substitutions.

Rule	**Informal Meaning**
No collisions: Global variables that appear in the pre- and post-conditions of a procedure must be renamed if they appear in the actual parameters.	This rule prevents global variable names from colliding with parameter names.

The proper way to handle the last case is to rename the global variable `g`, say to `g'`, wherever it appears in the pre- and post-conditions:

```
                /*  a == g', b == 3  */
sum(g | a, b)
                /*  a == g', b == 3, g == g' + 3  */
```

There is one more place where substitutions may cause problems: pre- and post-conditions that were not used in the original proof of the procedure, but which are intended to remain constant across a procedure call. We state a rule to handle these invariant conditions:

Statement	**Rule**	**Informal Meaning**

Adaptation (Invariant pre- and post-conditions)	`/* P */`
	`p(a)`
	`/* Q */`
	
	implies
	
	`/* P AND I */`
	`p(a)`
	`/* Q AND I */`
	
	Restriction:
	`I` may not contain any of the actual parameters or globals that may be modified in `p`.
		Whatever is true about a procedure call may be augmented with invariant assertions that are not affected by the procedure.
		
		The invariant assertions may not include references to actual parameters or global variables that may be modified by the procedure.

Here is an example of a valid application of the rule of adaptation. We wish to augment our proof about `sum` to include an assertion about the invariance of another variable `d`:

```
                /*  a == 2, b == 3, d == 6  */
sum(c | a, b)
                /*  a == 2, b == 3, c == 5, d == 6  */
```

This is valid, since the variable `d` does not appear as a global variable in the body of `sum`, and it does not appear in the list of parameters that may be modified. If the added clause included `c` there would be problems.

Local Variables

Local variables are easy to accommodate, as long as they do not collide with the pre- and post-conditions.

Statement	Rule	Informal Meaning

Declaration	`/* P */`
	`S(x/y)`
	`/* R */`
	
	implies
	
	`/* P */`
	`var x;`
	`S`
	`/* R */`
		Prove that some pre- and post-conditions hold for a sequence of statements `S` where every occurrence of variable `x` is replaced by a new variable `y`.
		
		Those same pre- and post-conditions hold for the original sequence of statements `S`, where the variable `x` is declared local.

For example, consider the following sequence of statements:

```
var A;
A = x;
x = y;
y = A;
```

The notation "`var A;`" is used to mean a local declaration of `A`.

If the pre- and post-conditions do not refer to the local variable `A` there should be no problem:

```
                /*  y == C, x == B  */
var A;
                /*  y == C, x == B  */
A = x;
                /*  y == C, A == B  */
x = y;
                /*  x == C, A == B  */
y = A;
                /*  x == C, y == B  */
```

On the other hand, the following will not work:

```
                /*  ?  */
var A;
                /*  y == x, x == B  */
A = x;
                /*  y == A, A == B  */
x = y;
                /*  x == A, A == B  */
y = A;
                /*  x == A, y == B  */
```

The solution is to rename all occurrences of the variable `A`, say to `A'`, in the sequence of statements, so that it will not collide with the pre- and post-conditions:

```
                /*  y == A, x == B  */
var A';
                /*  y == A, x == B  */
A' = x;
                /*  y == A, A' == B  */
x = y;
                /*  x == A, A' == B  */
y = A';
                /*  x == A, y == B  */
```

Recursion

The final topic to consider is recursive procedures. The rules we have introduced so far require that something be proved about the body of a procedure before anything can be said about calls to that procedure. Unfortunately, recursive procedures have calls to themselves within their bodies.

The solution is to do a proof by induction. First, prove that a pair of pre- and post-conditions correctly describe the base cases. Then assume that the pre- and post-conditions apply to a call of the procedure. Using that assumption, show that they apply to the body of the procedure. The base cases turn out to be the paths through the procedure that do not involve any recursive calls. There has to be at least one, or the procedure would never terminate.

Here is the rule that describes this method:

Statement	Rule	Informal Meaning

Recursion	`p(x) { S }`
	
	Assume
	
	`/* P */`
	`p(x)`
	`/* R */`
	
	in the proof:
	
	`/* P */`
	`S`
	`/* R */`
	
	Conclude:
	
	`/* P */`
	`p(x)`
	`/* R */`
		Prove that some pre- and post-conditions hold for the body of a procedure `S`, assuming that those same pre- and post-conditions hold for a call to the same procedure.
		
		Those same pre- and post-conditions hold for any call to the recursive procedure.

This rule appears to violate common sense. How can you assume that something is true in order to prove that it is? The answer is that part of the proof does not depend on the assumption---the paths through the body that do not include recursive procedure calls.

Here is a simple recursive procedure that computes the product of two numbers:

```
mult(r | a, b) {
  if (b > 0) {
     mult(r | a, b - 1);
     r = r + a;
  } else {
     r = 0;
  }
}
```

A plausible assumption for a call to this procedure is:

```
                /*  b >= 0  */
mult(r | a, b);
                /*  r == a * b, b >= 0  */
```

Let us see if we can prove that these pre- and post-conditions hold for the body:

```
                /*  b >= 0  */
if (b > 0) {
                /*  b - 1 >= 0, b > 0  */
                /*  b - 1 >= 0  */
   mult(r | a, b - 1);
                /*  r == a * (b - 1), b - 1 >= 0  */
                /*  r + a == a * b, b >= 0  */
   r = r + a;
                /*  r == a * b, b >= 0  */
} else {
                /*  b >= 0, b <= 0  */
                /*  0 == a * b, b >= 0  */
   r = 0;
                /*  r == a * b, b >= 0  */
}
                /*  r == a * b, b >= 0  */
```

Notice that the post-condition for the recursive call substitutes "`b - 1`" for "`b`", since the call passes "`b - 1`" as its second argument. That condition is a little stronger than the pre-condition we constructed for the following statement, so it is valid.

Termination

How can you show that a recursive procedure always terminates? You must show that each recursive call passes a different set of parameters than the original call, and that the parameters will eventually reach a value where no recursive call is made. For the example above we see that the second parameter (`b`) is strictly decreasing, and it will eventually cause the "`else`" branch to be taken, with no recursive calls.

References

[Hoare 71] Hoare, C.A.R., Procedures and parameters: An axiomatic approach. In E. Engeler, editor, *Symposium On Semantics of Algorithmic Languages -- Lecture Notes in Mathematics 188*, pages 102-116. Springer-Verlag, 1971.