9.5.2.1 Pass-by-Value

When a parameter is **passedby value**, the value of the actual parameter is used to initialize the corresponding formal parameter, which then acts as a local variable in the subprogram, thus implementing in-mode semantics.

Pass-by-value is normally implemented by copy, because accesses often are more efficient with this approach. It could be implemented by transmitting an access path to the value of the actual parameter in the caller, but that would require that the value be in a write-protected cell (one that can only be read). Enforcing the write protection is not always a simple matter. For example, suppose the subprogram to which the parameter was passed passes it in turn to another subprogram. This is another reason to use copy transfer. As we will see in Section 9.5.4, C++ provides a convenient and effective method for specifying write protection on pass-by-value parameters that are transmitted by access path.

The advantage of pass-by-value is that for scalars it is fast, in both linkage cost and access time.

The main disadvantage of the pass-by-value method if copies are used is that additional storage is required for the formal parameter, either in the called subprogram or in some area outside both the caller and the called subprogram. In addition, the actual parameter must be copied to the storage area for the corresponding formal parameter. The storage and the copy operations can be costly if the parameter is large, such as an array with many elements.

9.5.2.2 Pass-by-Result

Pass-by-result is an implementation model for out-mode parameters. When a parameter is passed by result, no value is transmitted to the subprogram. The corresponding formal parameter acts as a local variable, but just before control is transferred back to the caller, its value is transmitted back to the caller's actual parameter, which obviously must be a variable. (How would the caller reference the computed result if it were a literal or an expression?)

The pass-by-result method has the advantages and disadvantages of pass-by-value, plus some additional disadvantages. If values are returned by copy (as opposed to access paths), as they typically are, pass-by-result also requires the extra storage and the copy operations that are required by pass-by-value. As with pass-by-value, the difficulty of implementing pass-by-result by transmitting an access path usually results in it being implemented by copy. In this case, the problem is in ensuring that the initial value of the actual parameter is not used in the called subprogram.

One additional problem with the pass-by-result model is that there can be an actual parameter collision, such as the one created with the call

```
sub(p1, p1)
```

In `sub`, assuming the two formal parameters have different names, the two can obviously be assigned different values. Then, whichever of the two is copied to their corresponding actual parameter last becomes the value of p1 in the caller. Thus, the order in which the actual parameters are copied determines their value. For example, consider the following C# method, which specifies the pass-by-result method with the out specifier on its formal parameter.[5]

```
void Fixer(out int x, out int y) {
  x = 17;
  y = 35;
}
. . .
f.Fixer(out a, out a);
```

If, at the end of the execution of Fixer, the formal parameter x is assigned to its corresponding actual parameter first, then the value of the actual parameter a in the caller will be 35. If y is assigned first, then the value of the actual parameter a in the caller will be 17.

Because the order can be implementation dependent for some languages, different implementations can produce different results.

Calling a procedure with two identical actual parameters can also lead to different kinds of problems when other parameter-passing methods are used, as discussed in Section 9.5.2.4.

Another problem that can occur with pass-by-result is that the implementor may be able to choose between two different times to evaluate the addresses of the actual parameters: at the time of the call or at the time of the return. For example, consider the following C# method and following code:

```
void DoIt(out int x, int index){
  x = 17;
  index = 42;
}
. . .
sub = 21;
f.DoIt(list[sub], sub);
```

The address of list[sub] changes between the beginning and end of the method. The implementor must choose the time to bind this parameter to an address—at the time of the call or at the time of the return. If the address is computed on entry to the method, the value 17 will be returned to list[21]; if computed just before return, 17 will be returned to list[42]. This makes programs unportable between an implementation that chooses to evaluate the addresses for out-mode parameters at the beginning of a subprogram and one that chooses to do that evaluation at the end. An obvious way to avoid this problem is for the language designer to specify when the address to be used to return the parameter value must be computed.

5. The **out** specifier must also be specified on the corresponding actual parameter.

9.5.2.3 Pass-by-Value-Result

Pass-by-value-result is an implementation model for inout-mode parameters in which actual values are copied. It is in effect a combination of pass-by-value and pass-by-result. The value of the actual parameter is used to initialize the corresponding formal parameter, which then acts as a local variable. In fact, pass-by-value-result formal parameters must have local storage associated with the called subprogram. At subprogram termination, the value of the formal parameter is transmitted back to the actual parameter.

Pass-by-value-result is sometimes called **pass-by-copy**, because the actual parameter is copied to the formal parameter at subprogram entry and then copied back at subprogram termination.

Pass-by-value-result shares with pass-by-value and pass-by-result the disadvantages of requiring multiple storage for parameters and time for copying values. It shares with pass-by-result the problems associated with the order in which actual parameters are assigned.

The advantages of pass-by-value-result are relative to pass-by-reference, so they are discussed in Section 9.5.2.4.

9.5.2.4 Pass-by-Reference

Pass-by-reference is a second implementation model for inout-mode parameters. Rather than copying data values back and forth, however, as in pass-by-value-result, the pass-by-reference method transmits an access path, usually just an address, to the called subprogram. This provides the access path to the cell storing the actual parameter. Thus, the called subprogram is allowed to access the actual parameter in the calling program unit. In effect, the actual parameter is shared with the called subprogram.

The advantage of pass-by-reference is that the passing process itself is efficient, in terms of both time and space. Duplicate space is not required, nor is any copying required.

There are, however, several disadvantages to the pass-by-reference method. First, access to the formal parameters will be slower than pass-by-value parameters, because of the additional level of indirect addressing that is required.[6] Second, if only one-way communication to the called subprogram is required, inadvertent and erroneous changes may be made to the actual parameter.

Another problem of pass-by-reference is that aliases can be created. This problem should be expected, because pass-by-reference makes access paths available to the called subprograms, thereby providing access to nonlocal variables. The problem with these kinds of aliasing is the same as in other circumstances: It is harmful to readability and thus to reliability. It also makes program verification more difficult.

There are several ways pass-by-reference parameters can create aliases. First, collisions can occur between actual parameters. Consider a C++ function that has two parameters that are to be passed by reference (see Section 9.5.3), as in

6. This is further explained in Section 9.5.3.

```
void fun(int &first, int &second)
```

If the call to `fun` happens to pass the same variable twice, as in

```
fun(total, total)
```

then `first` and `second` in `fun` will be aliases.

Second, collisions between array elements can also cause aliases. For example, suppose the function `fun` is called with two array elements that are specified with variable subscripts, as in

```
fun(list[i], list[j])
```

If these two parameters are passed by reference and `i` happens to be equal to `j`, then `first` and `second` are again aliases.

Third, if two of the formal parameters of a subprogram are an element of an array and the whole array, and both are passed by reference, then a call such as

```
fun1(list[i], list)
```

could result in aliasing in `fun1`, because `fun1` can access all elements of `list` through the second parameter and access a single element through its first parameter.

Still another way to get aliasing with pass-by-reference parameters is through collisions between formal parameters and nonlocal variables that are visible. For example, consider the following C code:

```
int * global;
void main() {
   . . .
   sub(global);
   . . .
}
void sub(int * param) {
   . . .
}
```

Inside `sub`, `param` and `global` are aliases.

All these possible aliasing situations are eliminated if pass-by-value-result is used instead of pass-by-reference. However, in place of aliasing, other problems sometimes arise, as discussed in Section 9.5.2.3.

9.5.2.5 Pass-by-Name

Pass-by-name is an inout-mode parameter transmission method that does not correspond to a single implementation model. When parameters are passed by name, the actual parameter is, in effect, textually substituted for the corresponding