10.3.4 Pass by Name and Delayed Evaluation

Pass by name is the term used for this mechanism when it was introduced in Algol60. At the time it was intended as a kind of advanced inlining process for procedures, so that the semantics of procedures could be described simply by a form of textual replacement, rather than an appeal to environments and closures. (See Exercise 10.14.) It turned out to be essentially equivalent to the normal order delayed evaluation described in the previous chapter. It also turned out to be difficult to implement, and to have complex interactions with other language constructs, particularly arrays and assignment. Thus, it was rarely implemented and was dropped in all Algol60 descendants (AlgolW, Algol68, C, Pascal, etc.). Advances in delayed evaluation in functional languages, particularly pure functional languages such as Haskell (where interactions with side effects are avoided), have increased interest in this mechanism, however, and it is worthwhile to understand it as a basis for other delayed evaluation mechanisms, particularly the more efficient **lazy evaluation** studied in Chapter 3.

The idea of pass by name is that the argument is not evaluated until its actual use as a parameter in the called procedure. Thus, the *name* of the argument, or its textual representation at the point of call, replaces the name of the parameter to which it corresponds. As an example, in the C code

```
void inc(int x)
{ x++; }
```

if a call such as `inc(a[i])` is made, the effect is of evaluating `a[i]++`. Thus, if `i` were to change before the use of `x` inside `inc`, the result would be different from either pass by reference or pass by value-result:

```
int i;
int a[10];

void inc(int x)
{ i++;
  x++;
}

main()
{ i = 1;
  a[1] = 1;
  a[2] = 2;
  inc(a[i]);
  return 0;
}
```

This code has the result of setting `a[2]` to 3 and leaving `a[1]` unchanged.

Pass by name can be interpreted as follows. The text of an argument at the point of call is viewed as a function in its own right, which is evaluated every time the corresponding parameter name is reached in the code of the called procedure. However, the argument will always be evaluated in the environment of the caller, while the procedure will be executed in its defining environment. To see how this works, consider the example in Figure 10.4.

```
(1)  #include <stdio.h>
(2)  int i;

(3)  int p(int y)
(4)  { int j = y;
(5)    i++;
(6)    return j+y;
(7)  }

(8)  void q(void)
(9)  { int j = 2;
(10)   i = 0;
(11)   printf("%d\n", p(i + j));
(12) }

(13) main()
(14) { q();
(15)   return 0;
(16) }
```

Figure 10.4 Pass by name example (in C syntax)

The argument `i + j` to the call to `p` from `q` is evaluated every time the parameter `y` is encountered inside `p`. The expression `i + j` is, however, evaluated as though it were still inside `q`, so on line 4 in `p` it produces the value 2. Then, on line 6, since `i` is now 1, it produces the value 3 (the `j` in the expression `i + j` is the `j` of `q`, so it hasn't changed, even though the `i` inside `p` has). Thus, if pass by name is used for the parameter `y` of `p` in the program, the program will print 5.

Historically, the interpretation of pass by name arguments as functions to be evaluated during the execution of the called procedure was expressed by referring to the arguments as **thunks**.[7] For example, the above C code could actually imitate pass by name using a function, except for the fact that it uses the local definition of `j` inside `q`. If we make `j` global, then the following C code will actually print 5, just as if pass by name were used in the previous code:

```
#include <stdio.h>
int i,j;

int i_plus_j(void)
{ return i+j; }
```

(continues)

[7]Presumably, the image was of little machines that "thunked" into place each time they were needed.

(continued)

```
int p(int (*y)(void))
{ int j = y();
  i++;
  return j+y();
}

void q(void)
{ j = 2;
  i = 0;
  printf("%d\n", p(i_plus_j));
}

main()
{ q();
  return 0;
}
```

Pass by name is problematic when side effects are desired. Consider the `intswap` procedure we have discussed before:

```
void intswap (int x, int y)
{ int t = x;
  x = y;
  y = t;
}
```

Suppose that pass by name is used for the parameters `x` and `y` and that we call this procedure as follows,

```
intswap(i,a[i])
```

where `i` is an integer index and `a` is an array of integers. The problem with this call is that it will function as the following code:

```
t = i;
i = a[i];
a[i] = t;
```

Note that by the time the address of `a[i]` is computed in the third line, `i` has been assigned the value of `a[i]` in the previous line. This will not assign `t` to the array `a` subscripted at the original `i`, unless `i = a[i]`.

It is also possible to write bizarre (but possibly useful) code in similar circumstances. One of the earliest examples of this is called **Jensen's device** after its inventor J. Jensen. Jensen's device uses pass by name to apply an operation to an entire array, as in the following example, in C syntax:

```
int sum (int a, int index, int size)
{ int temp = 0;
  for (index = 0; index < size; index++)  temp += a;
  return temp;
}
```

If a and index are pass by name parameters, then in the following code:

```
int x[10], i, xtotal;
...
xtotal = sum(x[i],i,10);
```

the call to sum computes the sum of all the elements x[0] through x[9].

10.3.5 Parameter-Passing Mechanism versus Parameter Specification

One can fault these descriptions of parameter-passing mechanisms in that they are tied closely to the internal mechanics of the code that is used to implement them. While one can give somewhat more theoretical semantic descriptions of these mechanisms, all of the questions of interpretation that we have discussed still arise in code that contains side effects. One language that tries to address this issue is Ada. Ada has two notions of parameter communication, in parameters and out parameters. Any parameter can be declared in, out, or in out (i.e., both). The meaning of these keywords is exactly what you would expect: An in parameter specifies that the parameter represents an incoming value only; an out parameter specifies an outgoing value only; and an in out parameter specifies both an incoming and an outgoing value. (The in parameter is the default, and the keyword in can be omitted in this case.)

Any parameter implementation whatsoever can be used to achieve these results, as long as the appropriate values are communicated properly (an in value on entry and an out value on exit). Ada also declares that any program that violates the protocols established by these parameter specifications is **erroneous**. For example, an in parameter cannot legally be assigned a new value by a procedure, and the value of an out parameter cannot be legally used by the procedure. Thus, pass by reference could be used for an in parameter as well as an out parameter, or pass by value could be used for in parameters and copy out for out parameters.

Fortunately, a translator can prevent many violations of these parameter specifications. In one case, an in parameter cannot be assigned to or otherwise have its value changed, because it should act like a constant. In another case, an out parameter can only be assigned to, because its value should never be used. Unfortunately, in out parameters cannot be checked with this specificity. Moreover, in the presence of other side effects, as we have seen, different implementations (reference and copy in-copy out, for example) may have different results. Ada still calls such programs erroneous, but translators cannot in general check whether a program is erroneous under these conditions. Thus, the outcome of this specification effort is somewhat less than what we might have hoped for.