ubicación del argumento. Por lo mismo, este método es a veces conocido como **copia al entrar, copia al salir**, o como **copia-restauración**.

El paso por valor-resultado sólo se distingue del paso por referencia por la presencia del alias. Por lo tanto, en el código que sigue (escrito por comodidad en sintaxis C):

```
void p(int x, int y)
{ x++;
  y++;
}

main()
{ int a = 1;
  p(a,a);
  ...
}
```

a tiene valor 3 después de que p es llamado en el caso de que se utilice el paso por referencia, en tanto que a tiene el valor 2 si se utiliza el paso por valor-resultado.

Los problemas que se han quedado sin especificar en este mecanismo, y que posiblemente varían en los distintos lenguajes o implementaciones, son el orden en el que los resultados se copian de regreso a los argumentos y si las ubicaciones de los argumentos se calculan solamente a la entrada y se almacenan o son recalculados a la salida.

Una opción adicional es que un lenguaje pueda ofrecer también un mecanismo de **paso por resultados**, en el que no hay un valor de entrada, solamente el de salida.

8.3.4 Paso por nombre y evaluación retardada

Paso por nombre es el término que se utilizó para este mecanismo cuando fue introducido en Algol60. Entonces se pretendía que fuera un proceso avanzado de "código de línea" para los procedimientos, de modo que los procedimientos puedan ser descritos simplemente mediante una forma de reemplazo textual, en vez de un llamado a ambientes y cerraduras; vea el ejercicio 8.14. Resultó ser en esencia equivalente a la evaluación retardada de orden normal descrita en el capítulo anterior. También se descubrió que era difícil de implementar, y que tenía interacciones complejas con otros constructores de lenguaje, particularmente con los arreglos y la asignación. Por ello, raramente fue implementada y fue desechada en todos los descendientes de Algol60 (AlgolW, Algol68, C, Pascal, etcétera). Los adelantos en la evaluación retardada en los lenguajes funcionales, particularmente en lenguajes puramente funcionales como Haskell (donde se evitan las interacciones con efectos colaterales), ha aumentado, sin embargo, el interés en este mecanismo y vale la pena comprenderlo como base para otros mecanismos de evaluación retardada, en particular para la **evaluación perezosa** más eficiente, que se estudiará en el capítulo 11.

La idea del paso por nombre es que no se evalúa el argumento hasta su uso real (como parámetro) en el procedimiento llamado. Por lo que el *nombre* del argumento, o su representación textual en el punto de la llamada, reemplaza el nombre del parámetro al cual corresponde. Como un ejemplo, en el código (en sintaxis de C)

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

si una llamada, digamos, `inc(a[i])`, se hace, el efecto es evaluar `a[i]++`. Entonces, si `i` cambiara antes del uso de `x` en el interior de `inc`, el resultado sería diferente tanto del paso por referencia como del paso por valor-resultado:

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

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

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

Este código tiene el resultado de establecer a `a[2]` en 3 y dejar a `a[1]` sin modificación.

El paso por nombre puede interpretarse de la siguiente forma. El texto de un argumento en el punto de llamada se considera como una función por su propio derecho, misma que es evaluada cada vez que se llega al nombre del parámetro correspondiente en el código del procedimiento llamado. Sin embargo, el argumento se evaluará siempre en el ambiente del invocador, en tanto que el procedimiento se ejecutará en su ambiente de definición. Para ver cómo funciona esto, considere el ejemplo de la figura 8.1.

```
(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) }
```

Figura 8.1 Ejemplo de paso por nombre (en sintaxis de C).

El argumento `i + j` para la llamada a `p` desde `q` es evaluado cada vez que se encuentra el parámetro `y` en el interior de `p`. La expresión `i + j` es, sin embargo, evaluada como si todavía estuviera en el interior de `q`, por lo que en el renglón 4 en `p` produce el valor 2. Entonces, en el renglón 6, ya que `i` es ahora 1, produce el valor 3 (la `j` en la expresión `i + j` es la `j` de `q`, por lo que no ha cambiado, a pesar de que la `i` en el interior de `p` sí lo ha hecho). Por lo que si se utiliza paso por nombre en el programa para el parámetro `y` de `p`, el programa imprimirá 5.

Históricamente, la interpretación de los argumentos de paso por nombre como funciones a ser evaluadas durante la ejecución del procedimiento llamado fue expresado haciendo referencia a argumentos como **thunks.**[7] Por ejemplo, el código C antes citado podría de hecho imitar el paso por nombre utilizando una función, excepto por el hecho de que utiliza la definición local de `j` en el interior de `q`. Si convertimos a `j` en global, entonces, el código en C que sigue imprimirá realmente 5, igual que si se hubiera utilizado paso por nombre en el código anterior:

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

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

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;
}
```

El paso por nombre resulta problemático cuando se desean efectos colaterales. Considere el procedimiento `intswap` que vimos anteriormente:

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

Suponga que se utiliza el paso por nombre para los parámetros `x` y `y`, y que llamamos este procedimiento de la siguiente manera:

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

donde `i` es un índice de enteros y `a` es un arreglo de enteros. El problema con este tipo de llamada es que funcionará como lo hace el siguiente código:

---

[7] Es de suponer que la imagen era de máquinas pequeñas que hacían "thunk" en su sitio cada vez que eran necesitadas.

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

Note que para cuando la dirección de `a[i]` es calculada en el tercer renglón, a `i` se le ha asignado el valor de `a[i]` del renglón anterior, y esto no asignará a `t` al arreglo `a` con subíndices en el `i` original, a menos de que `i = a[i]`.

También es posible escribir código estrafalario (pero posiblemente útil) en circunstancias similares. Uno de los primeros ejemplos de lo anterior se conoce como **Jensen's device** en honor a su inventor J. Jensen. Jensen's device utiliza paso por nombre para aplicar una operación a la totalidad de un arreglo, como en el ejemplo que sigue, en sintaxis de C:

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

Si `a` e `index` son parámetros de paso por nombre, entonces en el código siguiente

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

la llamada a `sum` calcula la suma de todos los elementos `x[0]` hasta `x[9]`.

8.3.5 Mecanismo de paso de parámetros vs. la especificación de los parámetros

Es posible criticar estas descripciones de mecanismos de paso de parámetros con base en que están íntimamente ligados a la mecánica interna del código utilizado para su implementación. Aunque pueden darse más descripciones semánticas teóricas de estos mecanismos, todas las interrogantes con respecto a la interpretación que hemos analizado siguen presentándose en el código que contiene efectos colaterales. Un lenguaje que intenta dar solución a este problema es Ada. Ada tiene dos conceptos de comunicación de parámetros: parámetros `in` y parámetros `out`. Cualquier parámetro puede ser declarado como `in`, `out` o `in out` (es decir, los dos juntos). El significado de estas palabras clave es exactamente lo que usted puede esperar. Un parámetro `in` especifica que éste representa solamente un valor de entrada; un parámetro `out` se refiere solamente a un valor de salida; y un parámetro `in out` define tanto un valor de entrada como de salida (el parámetro por omisión es `in`, y en ese caso la palabra clave `in` puede omitirse).

Absolutamente cualquier implementación puede utilizarse para lograr estos resultados, siempre y cuando los valores apropiados se comuniquen adecuadamente (un valor `in` a la entrada y un valor `out` a la salida). Ada también declara que cualquier programa que viole los protocolos establecidos por las especificaciones de estos parámetros es un programa **erróneo.** Por ejemplo, no es posible asignar correctamente un parámetro `in` a un nuevo valor mediante un procedimiento, así como el valor de un parámetro `out` no puede ser usado correctamente por el procedimiento. Por lo tanto, podría utilizarse paso por referencia para los parámetros tanto `in` como `out`, o puede utilizarse paso por valor para los parámetros `in` y copia al salir para los parámetros `out`.

Afortunadamente, muchas violaciones a estas especificaciones de los parámetros pueden hacerse respetar mediante un intérprete —particularmente el hecho de que un parámetro `in`