UNICIENCIA 26
pp. 99-111
2012

CÁLCULO DE FUNCIONES POR RECURSIÓN DE COLA Y SUSTITUCIÓN DE PARÁMETROS

EVALUATING FUNCTIONS USING TAIL RECURSION AND PARAMETER SUBSTITUTION

Georges E. Alfaro Salazar[1]

RESUMEN

En este artículo se muestra una forma general de implementar el cálculo de funciones recursivas por medio de recursión lineal de cola. Se enfatiza en el uso de la recursividad de cola para realizar cálculos de manera eficiente.

Palabras clave: Recursividad, recursión de cola, secuencia, relación de recurrencia, serie finita.

ABSTRACT

This article shows a general way to implement recursive functions calculation by linear tail recursion. It emphasizes the use of tail recursion to perform computations efficiently.

Keywords: recursion, tail recursion, sequence, recurrence relation, finite series.

INTRODUCCIÓN

La recursión o recursividad es una técnica poderosa de programación. Muchas veces, el uso de una forma recursiva es más simple que su equivalente iterativa. Sin embargo, el uso incorrecto o ingenuo de la recursividad ha hecho pensar a muchos programadores que se trata de una técnica compleja e ineficiente en sí misma. La recursión lineal de cola es una forma de recursividad que puede ser tan eficiente (en términos de tiempo y espacio de memoria) como la iteración. En este artículo se explicará el uso de la recursión lineal de cola para implementar el cálculo de funciones definidas por relaciones de recurrencia. Además, se detallará como en algunos casos la implementación recursiva es, incluso, más sencilla que la programación de una fórmula directa.

SECUENCIAS Y RELACIONES DE RECURRENCIA

Una **secuencia** es una lista ordenada y numerable de elementos (o términos) tomados de un conjunto X.
Hay varias maneras de representar secuencias; sin embargo, la más común es utilizar una letra o símbolo para representar los elementos de la secuencia, junto con un subíndice, el cual identifica su posición relativa.

$$a_0, a_1, a_2, \cdots$$

La secuencia anterior puede escribirse como

$$A = A_n = \{a_n\}$$

Es importante recordar que una secuencia es un conjunto **ordenado**, por esta razón, la posición de los elementos es importante. De

[1] Universidad Nacional de Costa Rica - georgesalfaro@racsa.co.cr

Recibido 15 octubre de 2011- Aceptado 26 de noviembre de 2011- Corregido 23 de enero de 2012

la misma manera, es posible que dos o más elementos de una secuencia sean iguales.

La **longitud** de una secuencia es la cantidad o número de elementos componentes. Una secuencia puede ser de longitud infinita.

A la **suma de los términos** de una secuencia se le denomina **serie**. Muchas funciones se calculan al utilizar aproximaciones de series finitas.

Una secuencia se puede definir de manera explícita o por medio de una expresión que describa cada término.

Una **relación de recurrencia** es una fórmula o ecuación, la cual define **de manera recursiva** los elementos que forman una secuencia. Cada uno de los términos de la secuencia es calculado como una función de los términos anteriores. Los primeros *d* elementos de la secuencia son dados de manera explícita y se denominan las **condiciones iniciales** de la secuencia.

El **orden** de una relación de recurrencia está dado por la posición de los términos anteriores que son requeridos para el cálculo. Si se necesita únicamente el término inmediatamente anterior, entonces decimos que la relación es de orden 1. Si se requieren los últimos tres términos, la relación es de orden 3. Cuando la ecuación especifica hasta el n-ésimo término anterior, se trata de una relación de orden *n*.

Una relación también es orden *n* cuando usa hasta el n-ésimo término anterior aunque no haga referencia a términos intermedios.

Por ejemplo, la relación:

$$x_n = x_{n-1} - x_{n-2}^3 + 2x_{n-5}$$

es de orden 5, porque exige conocer el quinto término anterior de la secuencia, pero no necesita saber los valores del tercer y cuarto términos.

Cuando existe una fórmula o expresión **no recurrente** para calcular un término cualquiera de una secuencia, esta se denomina la **forma cerrada** de la secuencia (Chow, 1999).

Los términos de una secuencia definen una función cuyo dominio es un subconjunto de los números naturales. De la misma manera, cualquier función cuyo dominio sea un subconjunto de los números naturales puede definir una secuencia[2].

$$f(n) = a_n$$

Cuando tenemos una fórmula recurrente, los términos de una secuencia se pueden calcular por medio de una función recursiva. Las funciones empleadas pueden clasificarse según el número de veces que se invoque a la función recursiva. Si hay una o más invocaciones recursivas, la función se denomina **recursiva simple**[3]. Pero, si la invocación recursiva se hace una única vez dentro del cuerpo de la función, ésta se denomina **recursiva lineal**. Una función recursiva lineal es de orden $O(n)$ n el peor de los casos.

CÁLCULO DE LOS TÉRMINOS DE UNA SECUENCIA

En muchos casos, se puede definir directamente cada elemento de una secuencia. Una secuencia como la siguiente:

$$0, 1, 2, 3, 4, \ldots$$

puede definirse por medio de la expresión (fórmula cerrada) para el n-ésimo término:

$$a_n = n, n \geq 0$$

Por ejemplo, la secuencia:

$$1, 3, 5, 7, 9, \ldots$$

estaría definida por:

$$a_n = 2n+1$$

2 La definición puede extenderse por supuesto a cualquier conjunto numerable.

3 Algunas veces, si la invocación se hace dos veces, se dice que se está empleando **recursión binaria**. En cualquier caso, suponemos que la invocación recursiva no implica una composición de la función consigo misma.

Si embargo, ésta también podría expresarse por medio de la relación de recurrencia:

con la condición inicial:

$$a_0 = 1$$

La función factorial:

$$f(n) = n!$$

es simplemente una función que calcula el n-ésimo término de la secuencia definida por la relación de recurrencia:

$$a_n = na_{n-1}$$

y la condición inicial:

$$a_0 = 1$$

La relación de recurrencia asociada a la función factorial es de orden 1, pues la fórmula para a_n está escrita solamente en términos del elemento inmediatamente anterior en la secuencia (a_{n-1}).

Además, la función de Fibonacci:

$$f(n) = f(n-1) + f(n-2)$$

es una función que utiliza una relación de recurrencia de **orden 2**, porque requiere conocer al menos dos de los términos anteriores de la secuencia.

La siguiente secuencia:

$$1, 0, 0, 2, -2, 2, 2, 6, \ldots$$

está descrita por la relación:

$$a_n = 2a_{n-3} - a_{n-1}$$

y tiene 3 condiciones iniciales:

$$a_0 = 1, a_1 = 0, a_2 = 0$$

El orden de la relación es 3, porque requiere esa posición anterior (a_{n-3}) para el cálculo del término a_n, aunque no se necesiten todos los valores intermedios.

En general, el n-ésimo término de una secuencia de orden *d* se calcula como una función de *n* y los *d* términos anteriores, además, esta puede calcularse de manera iterativa o recursiva:

$$F(n) = a_n = \varphi(n, a_{n-1}, a_{n-2}, \cdots, a_{n-d})$$

sujeto a las condiciones iniciales:

$$a_0 = k_0, a_1 = k_1, \cdots, a_{d-1} = k_{d-1}$$

La relación de recurrencia puede ser una función lineal:

$$F(n) = \sum_{i=1}^{d} k_i(n) F(n-i) + h(n)$$

Una relación de recurrencia lineal se denomina **homogénea** si $h(n) = 0$. La función factorial y la función de Fibonacci son ejemplos de funciones con relaciones de recurrencia lineales homogéneas. En algunos casos los coeficientes $k_i(n)$ son constantes como en la función de Fibonacci. En cambio, en la función factorial los coeficientes no son constantes (dependen del valor de los parámetros de la función)[4].

Considere, por ejemplo, la función:

$$s(n) = \sum_{i=0}^{n} g(i)$$

4 Para la función de Fibonacci

$$d = 2, k_i(n) = 1, h(n) = 0$$

$$F(n) = \sum_{i=1}^{2} F(n-i) = F(n-1) + F(n-2)$$

Y para la función factorial:

$$d = 1, k_i(n) = n, h(n) = 0$$

$$F(n) = \sum_{i=1}^{1} nF(n-i) = nF(n-1)$$

Esta podría implementarse iterativamente de la siguiente manera[5]:

```
public int s1(int n) {
      int r = 0;
      for (int i = 0; i <= n; i++) {
         r += g(i);
      }
      return r;
   }
```

Observando que:

$$s(n) = \sum_{i=0}^{n} g(i) = \sum_{i=0}^{n-1} g(i) + g(n)$$

$$s(n) = s(n-1) + g(n)$$

y definiendo:

$$s(0) = 0$$

Podemos volver a escribir la función de manera recursiva:

```
public int s2(int n) {
      int r = 0;
      if (n != 0) {
         r = s2(n - 1) + g(n);
      }
      return r;
   }
```

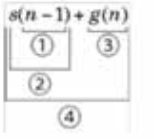
en la cual podemos eliminar el uso de la variable local que almacena el resultado y escribir de manera abreviada:

```
public int s2(int n) {
      return (n == 0) ? 0 : s2(n - 1) + g(n);
   }
```

La versión iterativa de la función utiliza una variable local *r* para guardar los valores intermedios resultantes entre la suma de cada término. Estos valores intermedios se pierden en cada iteración, cada vez que se hace la suma. Esto es característico de la programación imperativa (las operaciones son **destructivas**: la operación de asignación cambia el valor de una variable de manera que no puede recuperarse después).

La versión recursiva no destruye los totales parciales, los cuales se suman al valor del parámetro *n* en cada iteración. Observe que el valor del parámetro *n* se guarda en la pila. Este valor es necesario para completar la operación pendiente, la cual debe calcularse después del llamado recursivo.

La expresión de la relación de recurrencia para esta función necesita evaluarse de manera recursiva antes de efectuar la suma, que es la última operación calculada.



La suma del resultado por el llamado recursivo y el valor del término n-ésimo de la secuencia no puede calcularse antes, pues el valor no está disponible.

La función utiliza **recursión lineal**, esta es invocada **recursivamente**, sin embargo; **solo se llama una vez** dentro del cuerpo de la función. Esto garantiza que no se realicen llamadas innecesarias y mayores al número de iteraciones requeridas en la primera versión de la función, es decir; es una función de orden $O(n)$. Por otra parte, una función que utilice recursión binaria, en cambio, es de orden $O(n^2)$[6].

Los parámetros de la función deben resguardarse en la pila cada vez que haga el llamado recursivo. Esto ocasiona que la cantidad de memoria reservada en la pila sea proporcional al parámetro de la función: cuanto mayor sea el valor de este, mayor es la cantidad de memoria requerida para poder hacer el cálculo posterior. A veces, la cantidad de memoria requerida no es importante, pero en otras ocasiones, este factor hace imposible completar el cálculo.

[5] Los ejemplos de código están escritos en el lenguaje Java, pero podrían implementarse casi sin ningún cambio en C++ o C#.

[6] Asumiendo en ambos casos que el cálculo de cada término requiere un tiempo constante $O(1)$.

Analizaremos mediante un ejemplo la estructura de la invocación recursiva.

Consideraremos la función *g* definida por:

$$g(i) = i$$

la función *s* se convierte en[7]:

La función puede calcularse fácilmente utilizando iteración de la siguiente manera:

```
public int s1(int n) {
      int r = 0;
      int i = 0;
      while (i <= n) {
         r += i++;
      }
      return r;
   }
```

La variable *r* almacena el resultado. En cada iteración, la variable *r* contiene el total parcial de la suma hasta el i-ésimo término.

$$r_i \leftarrow r_{i-1} + i \quad \begin{aligned} r_0 &= 0 \\ r_1 &= 1 \leftarrow r_0 + 1 \leftarrow 0 + 1 \\ r_2 &= 3 \leftarrow 1 + 2 \\ r_3 &= 6 \leftarrow 3 + 3 \\ r_4 &= 10 \leftarrow 6 + 4 \\ r_5 &= 15 \leftarrow 10 + 5 \end{aligned}$$

La variable *i* es utilizada como índice de la sumatoria, pero al mismo tiempo se usa para calcular el i-ésimo término de la sumatoria. Se puede eliminar la variable y utilizar directamente el valor del parámetro *n*. En este caso, el argumento de la función no cambiaría, porque el parámetro es recibido por valor, no por referencia[8].

```
public int s2(int n) {
      int r = 0;
      while (n >= 0) {
         r += n--;
      }
      return r;
   }
```

En el listado anterior vemos que en cada iteración reducimos el valor de *n* antes de volver al inicio del ciclo.

El valor de la función es el mismo, pero los totales parciales se calculan en sentido inverso.

$$r_i \leftarrow r_{i-1} + n \quad \begin{aligned} r_0 &= 5 \leftarrow 0 + 5 \\ r_1 &= 9 \leftarrow 5 + 4 \\ r_2 &= 12 \leftarrow 9 + 3 \\ r_3 &= 14 \leftarrow 12 + 2 \\ r_4 &= 15 \leftarrow 14 + 1 \\ r_5 &= 15 \leftarrow 15 + 0 \end{aligned}$$

El resultado es el mismo, pues siempre se evalúan los mismos términos. En este ejemplo, podríamos eliminar el caso donde $i = n = 0$, porque no cambia el valor del resultado, pero no podemos suponer $g(0) = 0$ en el caso general.

[7] Existe una forma cerrada conocida para calcular el valor de la función directamente:

$$s(n) = \sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

Sin embargo, se utiliza la definición anterior de la función g para simplificar el ejemplo.
Muchas de las funciones mencionadas en los ejemplos tienen también una forma cerrada conocida, como la de Fibonacci (Subramani, 2004). No obstante, la evaluación de la forma cerrada implica cálculos realizados con aritmética de punto flotante y que además utilizan aproximaciones de constantes irracionales:

$$F(n) = \frac{1}{\sqrt{5}}(\varphi^n - \hat{\varphi}^n)$$

donde

$$\varphi = \frac{1+\sqrt{5}}{2}, \quad \hat{\varphi} = \frac{1-\sqrt{5}}{2}$$

Esta forma no siempre es práctica cuando se desea realizar el cálculo de manera exacta o cuando se necesita limitarlo a operaciones con aritmética entera.

[8] Se supone que todas las funciones se han de implementar siempre con paso de parámetros por valor.

Si consideramos la forma recurrente para la función:

$$s(0) = 0$$
$$s(n) = s(n-1) + n$$

podemos implementar la función de manera recursiva así:

```
public int s3(int n) {
      if (n >= 0) {
         return n + s3(n - 1);
      } else {
         return 0;
      }
   }
```

El cálculo recursivo de $s(5)$ se resolvería de la siguiente manera:

$$\begin{aligned} s(5) &= 5 + s(4) \\ &= 5 + 4 + s(3) \\ &= 5 + 4 + 3 + s(2) \\ &= 5 + 4 + 3 + 2 + s(1) \\ &= 5 + 4 + 3 + 2 + 1 + s(0) \\ &= 5 + 4 + 3 + 2 + 1 + 0 \\ &= 5 + 4 + 3 + 2 + 1 \\ &= 5 + 4 + 3 + 3 \\ &= 5 + 4 + 6 \\ &= 5 + 10 \\ &= 15 \end{aligned}$$

La suma no se efectúa hasta que el control regresa de la invocación recursiva cada vez, porque los valores aún no están disponibles. Si el llamado recursivo fuera la última operación a ejecutar, no se usaría el parámetro de la función para algún cálculo pendiente. Esta forma de recursión se conoce como **recursión lineal de cola**, o simplemente, **recursión de cola** (*tail recursion*).

Definimos una función:

$$s(n) = s'(n,i,A_i)$$

de manera que:

$$s'(n,i,A_i) = s'(n,i+i,A_{i+1})$$
$$A_{i+1} = A_i + i$$

El valor de la función depende solamente del valor de n, pero cuando se tiene:

$$A_0 = s(0)$$

los valores de A_i son los correspondientes a la función s calculada para cada valor de i.

Si tenemos:

$$s'(n,i,A) = \begin{cases} A & , i > n \\ s'(n,i+1,A+(i+1)) & , i \le n \end{cases}$$

entonces:

$$s(n) = s'(n,0,s(0))$$

Veamos que el caso donde $n = 0$:

$$s(n) = s'(n,0,s(0))$$

Por lo que:

$$\begin{aligned} s(n) &= s'(n,0,s(0)) \\ &= s'(n,0,0) \\ &= s'(n,1,0+1) \\ &= s'(n,2,0+1+2) \\ &\cdots \\ &= s'(n,k,0+1+2+\cdots+k) \\ &= s'(n,n,0+1+2+\cdots+n) \\ &= s'(n,n+1,\cdots) \\ &= 0+1+2+\cdots+n \end{aligned}$$

La implementación sería la siguiente:

```
public int s4(int n) {
      return s4b(n, 0, 0);
   }

   public int s4b(int n, int i, int A) {
      if (i >= n) {
         return A;
      } else {
         return s4b(n, i + 1, A + (i + 1));
      }
   }
```

El cálculo de cada término de la secuencia no se hace dentro del cuerpo de la función sino cuando se definen los parámetros para la siguiente invocación. El valor de la función

original lo obtenemos cuando esté disponible como argumento.

El cálculo de $s(5)$ sería como sigue:

$$\begin{aligned} s(5) = s'(5,0,0) &= s'(5,1,1) \\ &= s'(5,2,3) \\ &= s'(5,3,6) \\ &= s'(5,4,10) \\ &= s'(5,5,15) \\ &= 15 \end{aligned}$$

En el momento que el valor del índice alcance al parámetro n, quedaría:

$$s(n) = s'(n,n,s(n))$$

y se puede regresar el valor de $s(n)$ directamente.

Una ventaja de la recursión de cola es que permite eliminar el paso de parámetros en la pila (ya que no se necesita "recordar" su valor) y la evaluación puede hacerse con una cantidad de memoria constante. Además, es simple convertir una función recursiva la cual utiliza recursión de cola en una versión iterativa equivalente. Bastaría con escribir un ciclo controlado por la condición inicial de la recursión y utilizar variables para el valor de cada parámetro.

Podemos eliminar el parámetro i de la definición de s', y obtener la forma equivalente:

$$s'(n,i,A) = \begin{cases} A & ,i > n \\ s'(n,i+1,A+(i+1)) & ,i \le n \end{cases}$$

Se puede implementar esta función utilizando recursión de cola

```
public int s5(int n) {
      return s5b(n, 0);
   }

   public int s5b(int n, int A) {
      return (n > 0) ? s5b(n - 1, A + n) : A;
   }
```

La versión iterativa de ésta corresponde a

```
public int s6(int n) {
      int A = 0;
      while (n > 0) {
         A += n--;
      }
      return A;
   }
```

Si se tiene una secuencia tal que

$$a_0, a_1, a_2, \cdots$$

se puede derivar otra distinta por composición de una función, la cual se realiza sobre los elementos de la secuencia original:

$$b_n = \zeta(a_n, a_{n-1}, \cdots, a_0)$$

En particular, son importantes todas las series de la forma

$$b_n = \sum_{i=0}^{n} k_i a_i$$

pues muchas funciones se calculan por medio de una aproximación a través de una serie finita.

Esto es

$$\begin{aligned} b_0 &= k_0 a_0 \\ b_1 &= k_0 a_0 + k_1 a_1 \\ b_2 &= k_0 a_0 + k_1 a_1 + k_2 a_2 \\ &\cdots \\ b_n &= k_0 a_0 + k_1 a_1 + \cdots + k_n a_n \end{aligned}$$

Una función descrita por medio de una serie infinita puede aproximarse

$$f(x) = \sum_{i=0}^{\infty} t_i(x) \approx \sum_{i=0}^{n} t_i(x)$$

siempre que la serie converja.

A continuación, se explica cómo replantear una función recursiva simple, la cual utilice una relación de recurrencia en una función recursiva lineal de cola de manera general.

ESTRUCTURA GENERAL DE LAS FUNCIONES RECURSIVAS DE COLA

Al escribir una función por medio de una fórmula recurrente, se deben realizar invocaciones recursivas para obtener el valor de los términos anteriores requeridos para el cálculo.

$$f(n) = n! = \begin{cases} 1 & , n = 0 \\ n(n-1)! & , n > 0 \end{cases}$$

La función utiliza una relación de recurrencia de orden 1. Para calcular el término

$$a_n = f(n) = n!$$

se requiere únicamente el valor inmediatamente anterior (a_{n-1}). Como la función solo necesita un llamado recursivo para evaluar la expresión, se trata de una función recursiva lineal. La multiplicación por el valor de *n* no se completará hasta tener el resultado obtenido por dicha invocación recursiva.

La solución para no aplazar el cálculo de la multiplicación hasta obtener el resultado de la invocación recursiva es enviar dicho valor como parámetro de la función.

Definimos, entonces

$$f'(n,A) = \begin{cases} A & , n = 0 \\ f'(n-1,nA) & , n > 0 \end{cases}$$

Cada llamado de la función multiplica el valor del parámetro *n* por *A*. Este contiene el producto de todas las invocaciones anteriores. Compare la definición de esta función con la utilizada en el ejemplo previamente estudiado.

Si queremos evaluar la función original, basta con calcular:

$$f(n) = f'(n, f(0))$$

Por ejemplo, si se desea calcular $f(4) = 4!$, se tiene:

$$\begin{aligned} f(4) = f'(4, f(0)) &= f'(4,1) \\ &= f'(3,4) \\ &= f'(2,12) \\ &= f'(1,24) \\ &= f'(0,24) = 24 \end{aligned}$$

Obsérvese que los valores de *n* se multiplican comenzando por el mayor hasta que $n = 0$. Aquí, el orden en que se efectúen las operaciones de multiplicación no es relevante, pero lo podría ser dependiendo de la forma de la función.

Cuando es importante determinar cada i-ésimo término en cierto orden, se incluye como parámetro el índice del ese término que se está calculando:

$$f'(n,i,A) = \begin{cases} A & , i \geq n \\ f'(n,i+i,(i+1)A & , i < n \end{cases}$$

El proceso del cálculo se efectúa de la siguiente manera:

$$\begin{aligned} f(4) = f'(4,0,f(0)) &= f'(4,0,1) \\ &= f'(4,1,1) \\ &= f'(4,2,2) \\ &= f'(4,3,6) \\ &= f'(4,4,24) = 24 \end{aligned}$$

Se puede comparar este ejemplo con el anterior y comprobar que tienen una forma similar.

El parámetro *A*, el cual se arrastra en la función hasta que se da la condición trivial o básica de la recursión se denomina un **acumulador** (Clocksin, 1997).

El acumulador es el término anterior de la relación de recurrencia de la función, de orden 1.

Podemos extender la idea, si se envían como parámetro los términos anteriores de una función f, la cual utilice una relación de recurrencia de orden d.

Si se tiene una función:

$$f(n)=\begin{cases} a_0 & ,n=0 \\ a_1 & ,n=1 \\ \cdots & \\ a_{d-1} & ,n=d-1 \\ \varphi(n,a_{n-1},a_{n-2},\cdots,a_{n-d}) & ,n\geq d \end{cases}$$

donde

$$a_i = f(i)$$

Se construye una función recursiva lineal de cola de la siguiente manera:

$$f'(n,i,p_{d-1},\cdots p_0)=\begin{cases} p_0 & ,n=0 \\ p_1 & ,n=1 \\ \cdots & \\ p_{d-1} & ,n=d-1 \\ f'(n,i+1,\varphi(n,p_{d-1},\cdots p_0),p_{d-1},\cdots p_1) & ,n\geq d \end{cases}$$

con las condiciones iniciales:

$$p_i = f(i) = a_i,\ 0 \leq i < d$$

Así, el cálculo se realiza antes de la próxima invocación. En todas las invocaciones, se sustituye cada parámetro por el valor siguiente:

$$p_i \leftarrow p_{i+1}$$

El primer parámetro es sustituido por el valor de la ecuación de recurrencia evaluada sobre todos los valores iniciales. La fórmula recurrente es evaluada **antes** de hacer la invocación recursiva.

Aplicando esta regla general al ejemplo de la función de Fibonacci, podemos escribir:

$$f(n) = f(n-1) + f(n-2)$$

Con las condiciones iniciales:

$$f(0) = 0\ ,\ f(1) = 1$$

Entonces, definimos parámetros para una nueva función, los cuales corresponden con las condiciones iniciales de una función recursiva lineal de orden 2:

$$f(n) = f'(n,0,f(1),f(0))$$

$$f'(n,i,f_1,f_0)=\begin{cases} f_0 & ,n=0 \\ f_1 & ,n=1 \\ f'(n,i+1),f_0+f_1,f_1) & ,n>1 \end{cases}$$

Para calcular $f(5)$, tenemos

$$\begin{aligned} f(5) = f'(5,0,1,0) &= f'(5,1,1,1) \\ &= f'(5,2,2,1) \\ &= f'(5,3,3,2) \\ &= f'(5,4,5,3) \\ &= f'(5,5,8,5) \\ &= 8 \end{aligned}$$

Cuando $n = i$, el primer parámetro de la lista de condiciones iniciales contiene el valor buscado para la función original

$$f'(n,n,f_n,f_{n-1}) = f_n = f(n)$$

Los elementos de la secuencia original aparecen en la lista de parámetros de la nueva función.

Cualquier polinomio puede calcularse por medio de una relación de recurrencia. Si tenemos una secuencia formada por los coeficientes de un polinomio de grado n de la siguiente manera:

$$c_0, c_1, \cdots, c_n$$

$$c_i = 0\ ,\ i > n$$

El polinomio

$$p_n(x) = \sum_{i=0}^{n} c_i x^i$$

puede calcularse con la fórmula recurrente

$$q_i(x) = c_{n-1} + xq_{i-1}(x)$$

$$q_i(x) = q_{i-1}(x)\,, i > n$$

Que expresamos en forma recursiva de cola

$$q_n(x) = q'_n(x, c_n)$$

$$q'_i(x, c_i) = q'_{i-1}(x, c_{i-1} + xc_i)$$

$$q'_0(x, c_0) = c_0$$

Observe que

$$q'_i(x, c_i) = q'_{i-1}(x, c_{i-1})\,, i > n$$

Si $i = n$:

$$\begin{aligned} q_n(x) = q'_n(x, c_n) &= q'_{n-1}(x, c_{n-1} + xc_n) \\ &= q'_{n-2}(x, c_{n-2} + x(c_{n-1} + xc_n)) \\ &= q'_{n-2}(x, c_{n-2} + c_{n-1}x + c_n x^2) \\ &\cdots \\ &= q'_0(x, c_0 + x(c_1 + c_2 x \cdots + c_n x^{n-1})) \\ &= q'_0(x, c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n) \\ &= c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n \end{aligned}$$

Los coeficientes del polinomio pueden almacenarse dentro de un arreglo

```
public double p(double x, double c[]) {

   // La longitud del arreglo de
   // coeficientes determina el
   // grado del polinomio.

   int n = c.length - 1;
   return q(n, x, c, c[n]);
}

public double q(int n, double x,
   double c[], double cn) {
   return (n == 0) ? cn :
      q(n - 1, x, c, c[n - 1] + x * cn);
}
```

Un caso de aplicación práctica

Un buen ejemplo práctico del uso de la recursión de cola es el cálculo de una función definida o expresada por medio de una serie infinita. Esta se aproximará por medio de una serie finita.

Se utilizará una función trigonométrica para mostrar cómo el uso de una forma recurrente puede simplificar la evaluación.

La función coseno se expresa por medio de la serie infinita

$$\begin{aligned} \cos x &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots \\ &= \sum_{i=0}^{\infty} \frac{(-1)^i x^{2i}}{(2i)!} \end{aligned}$$

la cual puede aproximarse por medio de una serie finita

$$\cos x \approx x_n = \sum_{i=0}^{n} \frac{(-1)^i x^{2i}}{(2i)!}$$

donde el valor de n se determina por la precisión necesaria en el cálculo.

Una implementación ingenua para aproximar el valor de la función utilizaría una función factorial, y alguna otra para evaluar las potencias de x, de manera iterativa o utilizando logaritmos[8].

Si definimos una fórmula de recurrencia para la secuencia

$$X = \{x_n\}$$

podemos obtener el valor de la función de manera sencilla y eficiente, sin hacer uso de funciones para la evaluación del factorial o el cálculo de potencias.

En este caso

$$x_n = \sum_{i=0}^{n} t_n$$

donde

$$t_n = \frac{(-1)^n x^{2n}}{(2n)!}$$

Se puede observar que

$$t_n = \frac{(-1)^n x^{2n}}{(2n)!} = \frac{((-1)^{n-1} x^{2(n-1)}) \times ((-1)x^2)}{1 \times 2 \times \cdots \times (2n-2) \times (2n-1) \times (2n)}$$

$$t_n = \left(\frac{(-1)^{n-1} x^{2(n-1)}}{1 \times 2 \times \cdots \times (2n-2)} \right) \times \left(\frac{-x^2}{(2n-1) \times (2n)} \right)$$

Entonces, una relación de recurrencia para t_n sería

$$t_n = t_{n-1} \left(\frac{-x^2}{(2n-1) \times (2n)} \right)$$

con la condición inicial

$$x_0 = t_0 = \frac{(-1)^0 x^{2(0)}}{(2(0))!} = \frac{x^0}{0!} = \frac{1}{1} = 1$$

y una expresión general para x_n de la siguiente manera

$$x_n = x_{n-1} + t_n$$

```
public double f(double x, int n) {
  return f2(x, n, 0, 1.0, 1.0);
}

public double f2(double x, int n, int i,
      double xi, double ti) {
  double r = xi;
  if (i <= n) {
    ti *= -(x * x) /
        ((2 * i + 1) * (2 * i + 2));
    r = f2(x, n, i + 1, xi + ti, ti);
  }
  return r;
}
```

En el código anterior, el valor del parámetro t_i corresponde al valor siguiente de la secuencia, es decir, si el nuevo valor del índice es $j = i + 1$

$$t_j = t_i \frac{-x^2}{(2j-1)(2j)} = t_i \frac{-x^2}{(2(i+1)-1)(2(i+1))} = t_i \frac{-x^2}{(2i+1)(2i+2)}$$

es posible cambiar el criterio trivial de la función recursiva, al eliminar el valor del parámetro n y sustituirlo por un margen máximo de error, pues la secuencia siempre es convergente.

8 Una forma común de implementar el cálculo de potencias con exponente no entero es utilizar:

$$b^x = e^{\ln(b)x}$$

```
public double f(double x) {
   return f2(x, ERROR_MAXIMO, 0, 1.0, 1.0);
}

public double f2(double x, double e, int i,
      double xi, double ti) {
   double r = xi;
   if (e < Math.abs(ti / xi)) {
      ti *= -(x * x) /
         ((2 * i + 1) * (2 * i + 2));
      r = f2(x, e, i + 1, xi + ti, ti);
   }
   return r;
}
```

con un margen de error arbitrario

$$\left|\cos x - f_\varepsilon(x)\right| < \varepsilon$$

$$f_\varepsilon(x) = f'(x,\varepsilon,0,x_0,t_0) = f'(x,\varepsilon,0,1,1)$$

En la tabla 1 se muestran los resultados obtenidos con ambas funciones. La primera columna muestra el valor del ángulo, en el intervalo [0…π]. La segunda columna muestra el valor del coseno calculado con la función del paquete Math de la API de Java. En las dos últimas columnas aparece el valor calculado por la función recursiva anterior, la cual utiliza un número fijo de iteraciones (9) o una magnitud máxima de error (10^{-12}).

La función recursiva de cola se puede transformar fácilmente en una versión iterativa, pues no requiere la composición de otras funciones y además se puede sustituir el llamado recursivo por un ciclo.

```
public double fb(double x) {
   int i = 0;
   double xi = 1.0;
   double ti = 1.0;
   while (ERROR_MAXIMO < Math.abs(ti / xi)) {
      ti *= -(x * x) /
         ((2 * i + 1) * (2 * i + 2));
      i++;
      xi += ti;
   }
   return xi;
}
```

El resultado de evaluar el coseno mediante la función iterativa es exactamente igual que el obtenido con la función recursiva lineal de cola. Es posible obtener valores del coseno con una precisión de hasta 12 posiciones decimales ($\varepsilon < 10^{-12}$) con un máximo de 13 invocaciones recursivas a la función en el peor de los casos.

Tabla 1

0.000	1.0000000000	1.0000000000	1.0000000000
0.050	0.9987502604	0.9987502604	0.9987502604
0.100	0.9950041653	0.9950041653	0.9950041653
0.150	0.9887710779	0.9887710779	0.9887710779
0.200	0.9800665778	0.9800665778	0.9800665778
0.250	0.9689124217	0.9689124217	0.9689124217
0.300	0.9553364891	0.9553364891	0.9553364891
0.350	0.9393727128	0.9393727128	0.9393727128
0.400	0.9210609940	0.9210609940	0.9210609940
0.450	0.9004471024	0.9004471024	0.9004471024
0.500	0.8775825619	0.8775825619	0.8775825619
0.550	0.8525245221	0.8525245221	0.8525245221
0.600	0.8253356149	0.8253356149	0.8253356149
0.650	0.7960837985	0.7960837985	0.7960837985
0.700	0.7648421873	0.7648421873	0.7648421873
0.750	0.7316888689	0.7316888689	0.7316888689
0.800	0.6967067093	0.6967067093	0.6967067093
0.850	0.6599831459	0.6599831459	0.6599831459
0.900	0.6216099683	0.6216099683	0.6216099683
0.950	0.5816830895	0.5816830895	0.5816830895
1.000	0.5403023059	0.5403023059	0.5403023059
1.050	0.4975710479	0.4975710479	0.4975710479
1.100	0.4535961214	0.4535961214	0.4535961214
1.150	0.4084874409	0.4084874409	0.4084874409
1.200	0.3623577545	0.3623577545	0.3623577545
1.250	0.3153223624	0.3153223624	0.3153223624
1.300	0.2674988286	0.2674988286	0.2674988286
1.350	0.2190066871	0.2190066871	0.2190066871
1.400	0.1699671429	0.1699671429	0.1699671429
1.450	0.1205027694	0.1205027694	0.1205027694
1.500	0.0707372017	0.0707372017	0.0707372017
1.550	0.0207948278	0.0207948278	0.0207948278
1.600	-0.0291995223	-0.0291995223	-0.0291995223
1.650	-0.0791208888	-0.0791208888	-0.0791208888
1.700	-0.1288444943	-0.1288444943	-0.1288444943
1.750	-0.1782460556	-0.1782460556	-0.1782460556
1.800	-0.2272020947	-0.2272020947	-0.2272020947
1.850	-0.2755902468	-0.2755902468	-0.2755902468
1.900	-0.3232895669	-0.3232895669	-0.3232895669
1.950	-0.3701808314	-0.3701808314	-0.3701808314
2.000	-0.4161468365	-0.4161468365	-0.4161468365
2.050	-0.4610726914	-0.4610726914	-0.4610726914
2.100	-0.5048461046	-0.5048461046	-0.5048461046
2.150	-0.5473576655	-0.5473576655	-0.5473576655
2.200	-0.5885011173	-0.5885011173	-0.5885011173
2.250	-0.6281736227	-0.6281736227	-0.6281736227
2.300	-0.6662760213	-0.6662760213	-0.6662760213
2.350	-0.7027130768	-0.7027130768	-0.7027130768
2.400	-0.7373937155	-0.7373937155	-0.7373937155
2.450	-0.7702312540	-0.7702312540	-0.7702312540
2.500	-0.8011436155	-0.8011436155	-0.8011436155
2.550	-0.8300535352	-0.8300535352	-0.8300535352
2.600	-0.8568887534	-0.8568887534	-0.8568887534
2.650	-0.8815821959	-0.8815821959	-0.8815821959
2.700	-0.9040721420	-0.9040721420	-0.9040721420
2.750	-0.9243023786	-0.9243023786	-0.9243023786
2.800	-0.9422223407	-0.9422223407	-0.9422223407
2.850	-0.9577872376	-0.9577872375	-0.9577872376
2.900	-0.9709581651	-0.9709581651	-0.9709581651
2.950	-0.9817022030	-0.9817022030	-0.9817022030
3.000	-0.9899924966	-0.9899924966	-0.9899924966
3.050	-0.9958083245	-0.9958083245	-0.9958083245
3.100	-0.9991351503	-0.9991351502	-0.9991351503
3.150	-0.9999646585	-0.9999646584	-0.9999646585

Optimización de código

No todos los compiladores o interpretadores permiten la optimización de la recursión de cola. Cuando el compilador habilita la optimización, se puedan invocar las funciones al utilizar una cantidad constante de memoria. En el compilador gcc, por ejemplo, existe una opción de compilación (-O2), la cual habilita la optimización (Free Software Foundation, Inc., 2010). En otros lenguajes de programación, como Scheme, la optimización de los llamados recursivos de cola es parte de la definición estándar del lenguaje (Massachusetts Institute of Technology, 2003).

Cuando alguna implementación de un lenguaje permite la optimización, es posible sustituir cualquier ciclo por una invocación recursiva de cola, incluso cuando existe un ciclo indefinido (Gamboa & Cowles, 2004) (como un ciclo de atención de eventos en una GUI), pues la función puede invocarse de manera infinita (nunca provocará un desborde de la pila de llamadas).

Conclusiones

Buscar un planteamiento recursivo para calcular una función no siempre es complicado o ineficiente. Muchas veces un planteamiento recursivo es más natural y sencillo que una formulación iterativa o un cálculo directo. Una función recursiva puede ser tan eficiente como su equivalente iterativa.

Aún cuando la forma recursiva de la función no utilice recursión lineal, es posible conseguir una forma lineal de cola. En particular, se puede construir una forma lineal de cola cuando la función de recurrencia es lineal (homogénea o no).

BIBLIOGRAFÍA

Apostol, T. M. (1978). *Calculus, Cálculo con funciones de una variable, con una introducción al álgebra lineal*. Barcelona, España: Editorial Reverté.

Brookshear, J. G. (1993). *Teoría de la Computación. Lenguajes formales, autómatas y complejidad*. Wilmington, Delaware, USA: Addison-Wesley Iberoamericana.

Chow, T. Y. (1999). What is a Closed-Form Number? *The American Mathematical Monthly , 106* (5), 440-448.

Clocksin, W. F. (1997). *Clause and Effect*. Berlin, Heidelberg, Germany: Springer-Verlag.

Free Software Foundation, Inc. (2010). *Optimize Options*. Recuperado el 27 de Octubre de 2011, de Using the GNU Compiler Collection (GCC): http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Gamboa, R., & Cowles, J. (2004). Contributions to the Theory of Tail Recursive Functions. Austin, Texas, USA.

Massachusetts Institute of Technology. (23 de Octubre de 2003). *Scheme*. Recuperado el 27 de Octubre de 2011, de The Scheme Programming Language: http://groups.csail.mit.edu/mac/projects/scheme/

Skiena, S. S., & Revilla, M. (2003). *Programming Challenges*. New York, USA: Springer-Verlag.

Subramani, K. (31 de Agosto de 2004). Proof of Fibonacci Sequence closed form. Morgantown, West Virginia, USA.