

PROGRAMACIÓN FUNCIONAL: CONCEPTOS Y PERSPECTIVAS

*Fabio Augusto González Osorio
Instructor Asociado
Departamento de Ingeniería de Sistemas
Universidad Nacional de Colombia
fgonza@ingenieria.ingsala.unal.edu.co.*

Resumen

El presente artículo muestra conceptos subyacentes a la programación funcional, así como características que los hacen un enfoque particular y novedoso de la programación que lo convierten en una clara opción frente al enfoque imperativo convencional en el área del desarrollo de software.

INTRODUCCIÓN

El estilo de programación imperativa (es decir, la programación a través de acciones que modifican el estado del computador) ha dominado el panorama de la programación desde sus inicios; los lenguajes de más amplio uso están basados en este paradigma: Fortran, C, C++, Pascal, Basic, etc. Una razón fundamental de este dominio reside en que los lenguajes imperativos son más cercanos a la forma como realmente funciona la máquina.

Existen otros paradigmas de programación diferentes al imperativo como la programación funcional y la programación lógica, cuyo estudio, desarrollo y uso han estado principalmente restringidos al ámbito académico. La programación funcional, es casi tan antigua como la imperativa; el primer lenguaje funcional, LISP, fue desarrollado en la misma época en la que se desarrolló FORTRAN. Sin embargo, la programación funcional ha estado tradicionalmente circunscrita a áreas de aplicación específicas como la inteligencia artificial y la computación simbólica.

A pesar de la hegemonía de la programación imperativa la programación funcional cada vez toma más fuerza gracias a su capacidad expresiva, que permite escribir programas más compactos, y a su transparencia referencial que posibilita la sencilla verificación matemática de propiedades de los

programas. Igualmente, características como la recolección automática de basura, los sistemas de inferencia de tipos, el polimorfismo, la orientación a objetos, el emparejamiento de patrones, los algoritmos eficientes de compilación, se han desarrollado gracias al gran trabajo investigativo de los últimos años.

Todo lo anterior permite ubicar a la programación funcional como una importante opción para el desarrollo de *software* que facilite hacer realidad los ideales de la ingeniería de *software* como son: la reusabilidad, la modularidad, la mantenibilidad y la corrección[3].

En el presente artículo se pretende mostrar las ideas subyacentes a la programación funcional, así como ilustrar las características que la hacen un enfoque particular y novedoso de la programación. De igual manera, se hablará brevemente de su historia desarrollo y perspectivas.

I. ¿QUÉ ES LA PROGRAMACIÓN FUNCIONAL?

Una definición de programación funcional generalmente aceptada es la siguiente:

“El estilo de programación que enfatiza la evaluación de expresiones, antes que la ejecución de comandos”[6].

La definición anterior es bastante amplia y tal vez ambigua, pues no se precisa a qué se refiere el *énfasis* del cual habla. Esto refleja la frontera difusa que existe entre lenguajes funcionales puros y lenguajes no funcionales; en esta frontera se ubican lenguajes como LISP, SCHEME y ML, que nadie dudaría en catalogar como funcionales a pesar de que tienen características *no puras* como asignaciones y efectos laterales, a diferencia de los lenguajes funcionales puros, los cuales desarrollan todos su cómputos exclusivamente a través de la aplicación de funciones.

Una programa funcional está constituido enteramente por funciones; el programa principal es una función que toma como argumento la entrada al programa y genera la salida del programa como su resultado. Típicamente, la función principal se

define en términos de otras funciones, y éstas, a su vez, en término de más funciones; esta cadena finaliza en funciones predefinidas o primitivas.

A simple vista, un programa en lenguaje C se ajustaría a la definición de programa funcional (de hecho algunas personas consideran que el C es un lenguaje funcional); sin embargo, la principal diferencia de los programas funcionales puros respecto a los programas convencionales (imperativos) es que los únicos elementos constructores en los primeros son la definición y la aplicación de funciones, mientras que en los programas imperativos se utilizan, además, variables, asignaciones ciclos iterativos, etcétera.

Parece muy restrictivo el hecho de no poder utilizar variables, ni asignaciones, ni ciclos iterativos; sin embargo se ha demostrado matemáticamente que la definición y la aplicación de funciones era suficiente para construir cualquier función computable¹.

Para ilustrar la diferencia entre el enfoque imperativo y el funcional considérese el problema de construir una función *f* que reciba como argumento un natural *n* y retorne la suma de los naturales desde 1 hasta *n*, es decir:

$$f(n) = \sum_{i=1}^n i$$

En un lenguaje imperativo como C, se podría definir la función de la siguiente forma:

```
int f(int n)
{
    int i;
    int suma=0;
    for ( i=1; i<=n; i++)
        suma=suma+ i;
    return suma;
}
```

1. *El fundamento de los lenguajes funcionales es el cálculo lambda, desarrollado por Haskell Curry en la década del 30. Es un cálculo de funciones basado en la abstracción y la aplicación. Curry demostró que las funciones definibles dentro de este cálculo corresponden a las computables por una máquina de Turing[1].*

La implementación de esta función en un lenguaje funcional definitivamente exigiría otra estrategia, pues en estos lenguajes no se cuenta con variables, ni asignaciones, ni ciclos. Por tanto, se recurrirá a una definición recursiva² equivalente de f .

$$f(n) = \begin{cases} 0 & \text{si } n=0 \\ f(n-1) + n & \text{en otro caso} \end{cases}$$

Que en el lenguaje como CAML se vería de la siguiente forma:

```
let rec f = fun 0 -> 0
              | n -> f(n-1) + n;;
```

En SCHEME

```
(define f (lambda (n) (if (= 0 n) 0 (+ (f (- n 1)) n))))
```

En HASKELL (¡esta es una versión no recursiva, pero bastante elegante!):

```
f n = sum [1..n]
```

La ineficiencia de la recursión para resolver ciertos problemas es conocida; esto podría verse como un grave inconveniente para la programación funcional la cual hace uso extensivo de la misma, sin embargo, algoritmos iterativos pueden simularse a través de la recursión. Para mostrar esto, analicemos el típico algoritmo que calcula el n -ésimo término de la sucesión de Fibonacci, la cual se define como:

$$fib(n) = \begin{cases} 1 & \text{si } n=1 \vee n=2 \\ fib(n-1) + fib(n-2) & \text{en otro caso} \end{cases}$$

Un programa en CAML calcular $fib(n)$ sería:

```
let rec fib = fun 1 -> 1
              | 2 -> 1
              | n -> fib(n-1) + fib(n-2);;
```

Este programa, aunque bastante fiel a la definición matemática de la función, es demasiado ineficiente, pues su tiempo de ejecución $t(n)$ se

comporta asintóticamente como $fib(n)$ lo cual se nota como $t(n) = O(fib(n))$. Una mejor opción sería calcular $fib(n)$ de manera iterativa, es decir, generando $fib(1)$, $fib(2)$, $fib(3)$, etc, hasta llegar al $fib(n)$ (en cada paso se utilizan los dos últimos valores para generar el siguiente valor); el tiempo de ejecución de este algoritmo $t'(n)$ se comporta asintóticamente como n ($t'(n) = O(n)$), es decir el tiempo es una función lineal del argumento de la función, el cual es claramente mejor que el tiempo del algoritmo recursivo.

A continuación se muestra la implementación del algoritmo iterativo (realmente iterativo recursivo) en CAML:

```
let rec fibaux (n, cont, pen, ult) =
  if (cont >= n) then ult
  else fibaux (n, cont+1, ult, pen+ult);;
let rec fib n = fibaux (n, 2, 1, 1);;
```

II. ¿DÓNDE RESIDE LA POTENCIA DE LA PROGRAMACIÓN FUNCIONAL?

La potencia de la programación funcional depende de varias características que poseen los lenguajes funcionales; entre ellas: el manejo de funciones de alto orden, la declaración de tipos algebraicos, la inferencia de tipos, el emparejamiento de patrones y el manejo automático de la memoria dinámica.

Estas características no son exclusivas de los lenguajes funcionales; en el caso del emparejamiento de patrones esta facilidad fué tomada de Prolog. El manejo automático de la memoria dinámica, aunque tiene su origen en un lenguaje funcional, LISP, hoy en día lo poseen lenguajes imperativos como JAVA; sin embargo todas estas características han estado ligadas estrechamente al desarrollo de la programación funcional.

A continuación se describirán los elementos característicos más relevantes de la programación funcional.

2. Valga la pena aclarar que existe una solución mucho más sencilla: emplear la fórmula $f(n) = n(n+1)/2$. La fórmula recursiva se presenta con fines puramente ilustrativos.

A. Funciones de alto orden

El concepto "alto orden" se refiere a funciones que reciben como argumento o retornan funciones, es decir, las funciones pueden manipularse como datos; esta característica también es referida como "funciones como objetos de primera clase."

Por ejemplo, podríamos utilizar la siguiente definición de la derivada de una función f :

$$f'(x) = \lim_{h \rightarrow \infty} \frac{f(x+h) - f(x)}{h}$$

para construir una función *deriva* que nos aproxime la derivada de cualquier función

```
f:float → float
let deriva (f,x) = (f(x +. 0.001) -. f(x)) /. 0.001;;3
```

En este caso, f sería una función que es recibida como argumento por *deriva*; por lo tanto, *deriva* sería una función de alto orden. En el siguiente ejemplo:

```
let g x = x *. x -. 1 . 0 in deriva (g, 3. 0);;
```

Está calculándose una aproximación de la derivada de $g(x) = x^2 - 1$ calculada en $x = 3$.

Un concepto asociado a las funciones de alto orden es el de *currying*, el cual tiene su origen en el estudio matemático de funciones. En pocas palabras, este concepto sugiere que es suficiente estudiar funciones de un solo argumento. Por ejemplo, considere la función $f(x,y) = x+y$, la cual puede ser expresada como una función g tal que $g(x)$ es una función que al ser aplicada sobre y nos retorna $x+y$, es decir $(g(x))(y) = x+y$; implementado en CAML quedaría:

```
let g x = fun y -> x+y;;
```

El proceso de evaluación para $g\ 3\ 4$ es:

```
g 3 4 ==> (fun y -> 3+y) 4 ==> 3+4 ==> 7
```

En general, dada una función $f(x,y)$, el proceso de *currying* consiste en construir una función g tal que $(g(x))(y) = f(x,y)$. Podemos construir una función

en CAML que nos permita obtener la versión '*currificada*' de una función con dos argumentos:

```
let curry f = (fun x -> (fun y -> f(x,y) ) );;
```

Utilicemos la función *curry* aplicada sobre *deriva* para construir una función g' que corresponda a la derivada de $g(x) = x^2 - 1$:

```
let g x = x *. x -. 1.0;;
let g' = (curry deriva) g;;
```

El proceso de evaluación de $g'\ 3.0$ sería:

```
g' 3.0 ==> ((curry deriva)g) 3.0
==> ((fun x -> (fun y -> deriva(x,y))) g) 3.0
==> (fun y -> deriva (g,y)) 3.0
==> deriva (g, 3.0)
:
:
```

B. Tipos algebraicos

Los lenguajes funcionales con tipos como ML y Haskell dan la posibilidad de declarar tipos adicionales a los tipos primitivos (int, char, float, etc.). Los tipos declarados por el usuario se definen mediante constructores como la enumeración:

```
let dirección = norte | sur | oriente | occidente;;
```

...el producto cartesiano de tipos ya definidos o primitivos:

```
let racional = fraccion of int *int;;
```

y definiciones recursivas:

```
let lista = vacia | cons of int *lista;;
```

En el primer caso, se define un tipo que solo posee cuatro valores posibles; en el segundo ejemplo, el tipo racional se construye como el producto cartesiano del tipo primitivo *int*; valores como *fracción (3, 4)*, *fracción (-4, 2)*, etc.,

3. En CAML, los operadores aritméticos seguidos de un punto (+, -, *, /) se refieren a operadores sobre reales; el punto los distingue de los operadores respectivos sobre enteros[10].

pertencen a este tipo. *Fracción* es un constructor que no se ha declarado previamente.

En el último ejemplo se emplea el tipo *lista* dentro de su propia definición; concretamente se dice que una lista puede ser *vacía* o el '*cons*' de un entero con una lista ya existente; en este caso '*cons*' se puede entender como la operación de adicionar un entero a la cabeza de la lista. Ejemplos de elementos del tipo *lista* son:

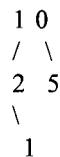
```
vacía
cons (5,vacía)
cons (3, cons (9, cons (1, vacía)))
```

Como puede observarse, el número de elementos pertenecientes al tipo *lista* es infinito; esto es, efecto directo del carácter recursivo de su definición. Precisamente esta característica, la recursividad es la que le da el nombre de algebraico al sistema de tipos y le permite al programador definir y manejar estructuras bastante complejas, sin necesidad de manipular apuntadores; esto elimina la gestión directa de la memoria dinámica y, por tanto obvia, una gran fuente de errores.

Otro ejemplo de tipo recursivo es el de árbol

```
type arbolbi = vacio | nodo of int*arbolbi*arbolbi;;
```

En este caso están representándose el tipo de árboles binarios con etiquetas enteras en sus nodos. El siguiente árbol:



Se representa mediante la siguiente expresión de tipo *arbolbi*:

```
nodo (10, nodo (2, vacio, nodo (1, vacio, vacio)),
nodo (5, vacio, vacio))
```

C. Emparejamiento de patrones

El emparejamiento de patrones (en inglés *Pattern Matching*), se refiere a la posibilidad que brindan algunos lenguajes funcionales de definir

funciones por casos; esto le da mayor capacidad expresiva al lenguaje permitiendo escribir el código más claro, sencillo y conciso.

Un ejemplo de definición de una función utilizando emparejamiento de patrones es la función *fib* recursiva del numeral I. Otro ejemplo es la siguiente definición de la función *Suma* que calcula el tamaño de una lista de enteros según la definición del tipo *lista* dada en el apartado B. del numeral II.

```
let rec Suma = fun vacía -> 0
                | cons (cabeza, resto) -> cabeza
                + Suma (resto);;
```

Aquí están definiéndose dos patrones posibles ("*vacía*" y "*cons (cabeza, resto)*") que pueden tomar el argumento enviado a *suma*; el compilador trata de emparejarlo con cada uno. En caso de conseguirlo, ejecuta el código a continuación de la flecha (->).

Por ejemplo, en caso de evaluar *Suma cons(5, cons(2, cons(8, vacía)))*, el compilador empareja el argumento con el segundo patrón ("*cons (cabeza, resto)*") haciendo *cabeza*=5 y *resto*=*cons(2, cons(8, vacía))*; la evaluación de la expresión continuaría así:

```
Suma cons (5, cons (2, cons (8, vacía) ) )
=> 5 + Suma cons (2, cons (8, vacía) )
=> 5 + 2 + Suma cons (8, vacía)
=> 5 + 2 + 8 + Suma vacía (en este caso se empareja el
                             primer patrón)
=> 5 + 2 + 8 + 0
```

D. Inferencia de tipos

Tradicionalmente, los sistemas de tipos de los lenguajes de programación se han dividido en dos: estrictos y no estrictos. En el primer caso, el programador debe declarar el tipo de cada una de las variables y de los argumentos de las funciones y procedimientos, y ceñirse de manera estricta a estas declaraciones; en el segundo caso, el programador no debe ceñirse necesariamente a las declaraciones⁴ y eventualmente las puede obviar, como cuando a una función declarada en C que recibe un entero, se le envía como argumento un número de punto flotante.

La experiencia ha demostrado que los sistemas de tipos estrictos son preferibles a los no estrictos, pues favorecen la depuración sencilla del código al eliminar en tiempo de compilación muchos errores potenciales; sin embargo pueden resultar muy engorrosos para el programador por su falta de flexibilidad.

Los sistemas de inferencia de tipos representan un punto intermedio entre los dos esquemas mencionados anteriormente. Por un lado, conservan el carácter estricto del sistema de tipos y por otro liberan al programador de declaraciones explícitas de los tipos de los argumentos de las funciones y de sus valores de retorno. Esto puede parecer contradictorio, pero la clave está en que el compilador hace el trabajo de asignación de tipos por el programador y, lo que es mejor, lo hace de la manera más general posible, es decir, evidenciando la genericidad y el polimorfismo cuando éstos tienen cabida dentro de una función.

Por ejemplo, al introducir la función *Suma* definida en el apartado C del numeral II.C, el intérprete de CAML responde con el siguiente mensaje:

Suma: lista -> int = <fun>

Lo cual está indicándonos que la función *Suma* recibe como argumento un dato del tipo *lista* y retorna como resultado un entero; esto es inferido por el intérprete de manera automática.

En el caso de una función como la siguiente:

let primero (x, y) = x;;

La cual de una pareja de datos nos retorna el primero, puede ser aplicada a parejas de diferentes tipos de datos; por tanto, el sistema de tipos debe inferir el tipo más general posible y éste es:

*primero : a' * b' -> a'*

Donde *a'* y *b'* se refieren a cualquier tipo, es decir, son variables de tipo que nos indican el carácter general de la función. El tipo inferido por el

4. En lenguajes como LISP ni siquiera existe el concepto de tipo, todos los datos son compatibles entre sí; éste es el caso extremo.

intérprete para la función *curry* definida en el apartado A del numeral II. es:

*curry : (a' * b' -> c') -> (a' -> (b' -> c'))*

el cual evidencia el alto orden de la función, pues ésta recibe como parámetro una función de tipo (*a' * b' -> c'*) y retorna una función (también de alto orden) de tipo (*a' -> (b' -> c')*).

PERSPECTIVAS Y CONCLUSIONES

Las características que hemos ilustrado de hasta ahora evidencian el gran potencial de los lenguajes funcionales como herramientas que les facilite a los programadores enfrentar la complejidad creciente del desarrollo de *software*; esto nos permite afirmar que en los próximos años los lenguajes funcionales tomarán un lugar en el área de *desarrollo de software a gran escala*, al lado de lenguajes tan tradicionales como C, C++, ADA [3,4]. La afirmación anterior se sustenta con los siguientes puntos:

- **Transparencia referencial.** Este concepto se refiere a la propiedad de los lenguajes funcionales que hace que la misma expresión siempre represente el mismo valor; esto permite probar matemáticamente la corrección de un programa. La posibilidad de escribir programas cuya corrección es probable en vez de gastar el tiempo pescando errores puede revolucionar el proceso de producción de *software*.
- **Fundamentación matemática.** Desde sus inicios, la programación funcional ha tenido un gran componente matemático, el cálculo Lambda, la lógica combinatoria, las teorías de tipos, los sistemas de reescritura, la teoría de dominios y la teoría de categorías son algunas de las áreas de la matemática que la fundamentan. Esto le da pilares suficientemente sólidos que le permiten ser la base del desarrollo de una verdadera ciencia de la programación.
- **Eficiencia de compiladores e intérpretes.** Uno de los inconvenientes que tradicionalmente se le ha achacado a los lenguajes funcionales es la ineficiencia de sus

intérpretes y compiladores. Hoy día, esto no representa un problema pues los avances investigativos han permitido la construcción de compiladores que generan código nativo que iguala en eficiencia el código generado por compiladores convencionales (C, Fortran, etcétera.).

- **Paralelismo Implícito.** El hecho que los lenguajes funcionales (puros) no permitan efectos laterales ni el uso de variables globales, hace que la evaluación de diferentes expresiones constituyan procesos independientes y, por tanto, que pueden ser ejecutados de manera simultánea. Esta característica puede ser explotada para programar computadores paralelos de manera natural algo que no se ha logrado de manera satisfactoria a través de la programación convencional.

BIBLIOGRAFÍA

1. CURRY H., FEYS R. *Combinatory Logic*,- North-Holland , 1958.
2. DYBVIK K. *The Scheme programming language*. Addison-Wesley 1996.
3. HUGHES J. *Why functional programming matters*. *The computer Journal*, vol. 32 N° 2. Abril 1989.
4. POUNTAIN, D. " *Functional programming comes of age*". *BYTE*, Agosto de 1994
5. SETHI R. *Programming languages*, Addison-Wesley 2 Ed ., 1996.

Páginas en internet

Preguntas frecuentes de comp. lang funcional

6. <http://www.cs.nott.ac.uk/Department/Staff/gmb/faq.html>

Programación funcional

7. <http://cm-bell-labs.com/cm/cs/who/wadler/guide.html>
8. <http://www.lpac.ac.uk/SEL-HPC/Articles/FuncArchive.html>
9. <http://carol.fwi.uva.nl/~jon/func.html>

Lenguajes

10. CAML: <http://pauillac.inria.fr/caml/index-eng.html>
11. Haskell: <http://www-i2.informatik.rwth-aachen.de/Forschung/FP/Haskell/>
12. Scheme: <http://ai.mit.edu>