El paradigma de programación Funcional

Luque Romero Guido, Univ.
gluque2@umsa.bo

Universidad Mayor de San Andres, Facultad de Ciencias Puras y Naturales, Programa de [Informática] (1)

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.
Palabras Clave: programación funcional, paradigma funcional, abstracción de datos, funciones de orden superior, evaluación perezosa, transparencia referencial, tipos de datos

Abstract
This article shows concepts underlying functional programming, as well as characteristics that make them a particular focus and novel of the programming that make it in a clear choice versus the imperative approach conventional in the area of software development.
Keywords: functional programming, functional paradigm, data abstraction, higher-order functions, evaluation lazy, referential transparency, data types

1. 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, e, 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, LlSP, 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, el modularidad, la mantenibilidad y la corrección. 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.

1.1 ¿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 sus cómputos exclusivamente a través de la aplicación de funciones. Un 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 computable'. 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 retome 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;
intsuma=O;
for (i=l; i<=n; i++)
suma=suma+ i;
retum suma;
}
```

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 recursiva-equivalente de *f*.

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

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

$$let\ rec\ f = fun\ 0 \rightarrow 0$$
$$| n -> f(n-1) + n;;$$

EnSCHEME

$$\Big(define\ f\ \Big(lambda(n)(if\ (n = 0)0)\big(+\big(f(-n\ 1)\big)n\big)\Big)\Big)$$

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 similarse a través de la recursión. Para mostrar esto, analicemos el típico algoritmo que calcula el n-ésimo término de la sucesión de Fibonnaci, la cual se define como:

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

Un programa en CAML calcular *fib* (n) seria:

```
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 {J),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 1a implementación del algoritmo iterativo (realmente iterativo recursivo) en CAML.:

letrec 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);;

1.2 ¿DÓNDE RESIDE LA POTENCIA DELA 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 fue 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.

A. Funciones de alto orden

El concepto "alto orden" se refiere a funciones que reciben como argumento o retoman funciones, es decir, las funciones pueden manipulase 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'(x)= Lim f(x+h)-f(x) h~oo h

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

f: float -> float

let deriva (f,x) = ([(x +. 0.00 l) -.f(x)) /. 0.00 It;

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 gx = x*. x -. 1 . Oin deriva (g, 3. O);;

Está calculándose una aproximación de la derivada de g(x) =x2 -l 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 funciónf(x,y) =x+y, la cual puede ser expresada como una función gtal que g (x) es una función que al ser aplicada sobre y nos retoma x+y, es decir (g(x))(y)= x+y; implementado en CAML quedaría:

let gx =fun y -> x+y ;;

E 1 proceso de evaluación para g 3 4 es:

g 34 ==> (funy -> 3+y) 4 ==> 3+4 ==> 7

En general, dada una funciónf(x,y), el proceso de currying consiste en construir una función g tal que g (x)(y) =j{x, y). Podemos construir una función en CAML que nos permita obtener la versión 'currificada' de una función con dos argumentos:

let curryf= (fun x -> ([uny-> 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 gx =x *. x-l. O;;

let g' = (curry deriva) g ;;

El proceso de evaluación de g' 3. Osería:

g' 3.0= => ((curry deriva)g) 3.0

==> (tfun x -> (fun y ->deriva(x,y))) g) 3.0

==> (fun y -> deriva (g,y)) 3. O

==> 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 I sur I oriente I occidente;;

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

let racional = fraccion of int *int;;

y definiciones recursivas:

let lista = vacia I cons ofint *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) r fracción (-4, 2), etc., pertenecen 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 vacia 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 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 = vado I nodo ofint*arbo1bi*arbolbi;;

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

1 O

/ \

2 5

\

1

Se representa mediante la siguiente expresión de tipo arbolbi:

nodo (10, nodo (2, vacio, nodo (1, vacio, vacioi),

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 1. 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 11.

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

Aquí están definiéndose dos patrones posibles C'vacia" 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, vacia))) , el compilador empareja el argumento con el segundo patrón ("cons (cabeza, resto)") haciendo cabeza=5 y resto=cons(2, cons(8, vacia)); la evaluación de la expresión continuaría así:

Suma cons (5, cons (2, cons (8, vacia)))

=> 5 + Suma cons (2, cons (8, vacia))

=> 5 + 2 + Suma cons (8,vacia)

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

=> 5 + 2 + 8 + O

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 declaraciones' 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 retomo. 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 e del numeral Il.C, el intérprete de eAML 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 retoma como resultado un entero; esto es inferido por el intérprete de manera automática.

En el caso de una función como la siguiente:

lel primero (x, y) = x;;

La cual de una pareja de datos nos retoma 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 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 retoma una función (también de alto orden) de tipo (a' -> (b' -> c')).

2. MATERIALES Y MÉTODOS/METODOLOGÍA

Los materiales que se usó para las comparaciones de los diferentes lenguajes son: una computadora 4 RAM (min.), también deben tener los diferentes programas ya instalados.
A continuación, se dará un vistazo al código de cada uno de los programas en los diferentes tipos de lenguajes:

Lenguajes no Funcionales:

	Python	Java	C#

Fibonacci	def fibonacci(num):
	  if num < 5:
	  s = "0 0 1 1 "
	  else:
	  num1 = 0
	  num2 = 0
	  num3 = 1
	  num4 = 1
	  series = 2
	  s = "0 0 1 1"
	  for i in range(4,num):
	  num1 = num2;
	  num2 = num3;
	  num3 = num4;
	  num4 = series;
	  s = s + ' '+str(series);
	  series = num1 + num2 + num3 +num4;
	  return s
		String s ="";
		  if (num < 5){
		  s = "0, 0, 1, 1, ";
		  }
		  else{
		  int num1 = 0;
		  int num2 = 0;
		  int num3 = 1;
		  int num4 = 1;
		  int series = 2;
		  s = "0, 0, 1, 1,";
		  out.println(s);
		  for (int i =5; i<= num;i++){
		  num1 = num2;
		  num2 = num3;
		  num3 = num4;
		  num4 = series;
		  s = series+ ", ";
		  out.println(s);
		  series = num1 + num2 + num3 +num4;
		  }
		  }	

Calculadora	def Suma(x, y):
	  return x + y
	
	def Resta(x, y):
	  return x - y
	
	def Multiplicar(x, y):
	  return x * y
	
	def Dividir(x, y):
	  return x / y
	def CalcuSuma(x, y): return Suma(x, y)
	def CalcuResta(x, y): return Resta(x, y)
	def CalcuMul(x, y): return Multiplicar(x, y)
	def CalcuDiv(x, y): return Dividir(x, y)
		private int suma(int a,int b){ return a + b;}
		private int resta(int a,int b){ return a - b;}
		private int mult(int a,int b){ return a * b;}
		private int div(int a,int b){ return a / b;}
		
		private int calcuSuma(int a,int b){ return suma(a,b);}
		private int calcuResta(int a,int b){ return resta(a,b);}
		private int calcuMul(int a,int b){ return mult(a,b);}
		private int calcuDiv(int a,int b){ return div(a,b);}	

Operaciones de Matrices	def suma():
	  for i in range(2):
	  for j in range(2):
	  S[i][j] += A[i][j] + B[i][j]
	
	def resta():
	  for i in range(2):
	  for j in range(2):
	  R[i][j] += A[i][j] - B[i][j]
	
	def multi():
	  for i in range(2):
	  for j in range(2):
	  res=0
	  for k in range(2):
	
	  res += A[i][k] * B[k][j]
	  M[i][j] = res
	def trasp():
	  for i in range(2):
	  for j in range(2):
	  T[i][j] = A[j][i]
		//suma
		for(int i=0;i<2;i++){
		  for(int j=0;j<2;j++){
		  S[i][j] = A[i][j] + B[i][j];
		  }
		}
		//resta
		for(int i=0;i<2;i++){
		  for(int j=0;j<2;j++){
		  R[i][j] = A[i][j] - B[i][j];
		  }
		}
		//multiplicacion
		for(int i=0;i<2;i++){
		  for(int j=0;j<2;j++){
		  int res =0;
		  for(int k=0;k<2;k++){
		  res += A[i][k] * B[k][j];
		  }
		  M[i][j] = res;
		  }
		}
		//transpuesta
		for(int i=0;i<2;i++){
		  for(int j=0;j<2;j++){
		  T[i][j] = A[j][i];
		  }
		}	

Factorial	def fact(n):
	  factorial_total = 1
	  while n > 1:
	  factorial_total *= n
	  n -= 1
	  return factorial_total
		private int factorial(int n){
		  int res = 1;
		  while (n > 1){
		  res *= n;
		  n -= 1;
		  }
		  return res;
		}	

Serie de Cubos	def serieC(num):
	  j = 1
	  s = ""
	  for i in range(1, num+1):
	  j = i*i*i
	  s = s +str(j)+ ", ";
	  return s
		String s ="";
		int series = 0;
		for (int i =1; i<= num;i++){
		  series = i*i*i;
		  s = series+ ", ";
		  out.println(s);
		}	

Se observa que python reduce las líneas de código a comparación de java según el algoritmo que se ejecute.

Leguajes Funcionales:

	Scala	F#

Fibonacci	def fibonacci_iterative(n: Int): String = {
	  var s: String = ""
	  if(n < 5){
	  s = "0, 0, 1, 1, "
	  s
	  } else {
	  var i = n
	  var num1 = 0
	  var num2 = 0
	  var num3 = 1
	  var num4 = 1
	  var new_acc = 2
	  s = "0, 0, 1, 1"
	  while(i > 4) {
	  i -= 1
	  num1 = num2
	  num2 = num3
	  num3 = num4
	  num4 = new_acc
	  s = s + ", "+ new_acc.toString
	  new_acc = num1 + num2 + num3 + num4
	  }
	  s
	  }
	  s
	}	

Calculadora	def sum(a: Int, b: Int): Int = a + b def res(a: Int, b: Int): Int = a - b def mult(a: Int, b: Int): Int = a * b def divide(a: Int, b: Int): Int = a / b def calcuSum(a: Int, b: Int)=sum(a, b) def calcuRes(a: Int, b: Int)=res(a, b) def calcuMul(a: Int, b: Int)=mult(a, b) def calcuDiv(a: Int, b: Int)=divide(a, b)	

Operaciones de Matrices	def sumaM=()=>{
	  i = 0;
	  while (i < 2) {
	  j = 0;
	  while (j < 2) {
	  MatrixS(i)(j) = Matrix1(i)(j) + Matrix2(i)(j)
	  j = j + 1;
	  }
	  i = i + 1;
	  }
	}
	  def restM=()=>{
	  i = 0;
	  while (i < 2) {
	  j = 0;
	  while (j < 2) {
	  MatrixR(i)(j) = Matrix1(i)(j) - Matrix2(i)(j)
	  j = j + 1;
	  }
	  i = i + 1;
	  }
	}
	
	def mulM=()=>{
	  i = 0;	

	while (i < 2) {
	j = 0;
	while (j < 2) {
	res = 0;
	k = 0;
	while (k < 2) {
	res = res + (Matrix1(i)(k) *
	Matrix2(k)(j));
	k = k + 1;
	}
	MatrixM(i)(j) = res;
	j = j + 1;
	}
	i = i + 1;
	}
	}
	
	def traspM=()=>{
	i = 0;
	while (i < 2) {
	j = 0;
	while (j < 2) {
	MatrixT(i)(j)=Matrix1(j)(i);
	j = j + 1;
	}
	i = i + 1;
	}
	}	

Factorial	def factorial=(n: Int)=>{
	res=1
	j = n
	while (j > 0) {
	res = res*(j)
	j = j - 1
	}
	res
	}	

Serie de Cubos	def serieCubos(n: Int): String = {
	var s: String = ""
	var num1 = 1
	var i = 1
	while(i <= n) {
	num1 = i*i*i
	s = s + num1+ ", "
	i += 1
	}
	s
	}	

Fuente: Adaptado de (Guido, 2021)

Para hacer que el código fuente de Scala sea un código ejecutable que se ejecute en una máquina virtual Java, debe compilarse en código de bytes Java. Al estar orientado a objetos, Scala utiliza una sintaxis de llaves que recuerda al lenguaje de programación c.
Scala tiene mucha flexibilidad en comparación con java. A continuación se muestran algunos ejemplos.

Las líneas se unen automáticamente si comienzan o terminan con un símbolo t, o si hay paréntesis o corchetes sin cerrar, no es necesario el punto y coma.
El impulso central detrás de Scala es hacer la vida más fácil y productiva para el programador. Esto se hace con tres técnicas principales: en primer lugar, agrega expresividad, fusiona estrechamente conceptos de programación funcional y orientada a objetos en un lenguaje, y luego protege la información existente ejecutándose en la máquina virtual Java e interoperando sin problemas con Java. En segundo lugar, reduce el estándar, para que los programadores puedan concentrarse en la lógica de sus problemas.

3. RESULTADOS Y DISCUSIÓN

- **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 les 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 constituya 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.

4. CONCLUSION

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, aliado de lenguajes tan tradicionales como e, e++, ADA.

REFERENCIAS

S.N recuperado el 15 de diciembre del 2021 de la página: http://www.cs.nott.ac.uk/Department/Staff/gmb/faq.html

Programación funcional

S.N recuperado el 10 de diciembre del 2021 de la página: http://cm-bell-Iabs.com/cm/cs/who/wad ler/ guide.html

S.N recuperado el 13 de diciembre del 2021 de la página: http://www.lpac.ac.uklSEL- HPC/Articles/ FuncArchive.html

S.N recuperado el 14 de diciembre del 2021 de la página: http://carol.fwi.uva.nl/-:jon/func.html Lenguajes

S.N recuperado el 15 de diciembre del 2021 de la página: CAML: http://pauillac.inria.fr/camllindex-eng. html

S.N recuperado el 15 de diciembre del 2021 de la página: Haskell: http://www-i2 informatik.rwthaachen.de/Forschung/FP/Haskell/

S.N recuperado el 15 de diciembre del 2021 de la página: Scheme: http://ai.mit.edu

Repositorio Git de los programas:

https://github.com/Gluquer/examen-Practico.git