Paradigmas de Programación

Paradigma Funcional

Historia

Sus orígenes provienen del Cálculo Lambda (o λ-cálculo), una teoría matemática elaborada por Alonzo Church como apoyo a sus estudios sobre computabilidad.

Los programas escritos en un lenguaje funcional están constituidos únicamente por definiciones de funciones, entendiendo éstas no como subprogramas clásicos de un lenguaje imperativo, sino como funciones puramente matemáticas, en las que se verifican ciertas propiedades como la transparencia referencial.

Otras características propias de estos lenguajes son la no existencia de asignaciones de variables y la falta de construcciones estructuradas como la secuencia o la iteración (lo que obliga en la práctica a que todas las repeticiones de instrucciones se lleven a cabo por medio de funciones recursivas).

¿Qué es la Programación Funcional?

C, Java, Pascal, Ada, etc.. son lenguajes imperativos. Son "imperativos" en el sentido de que consisten en una secuencia de comandos que son ejecutados uno tras otro estrictamente.
Un programa funcional es una expresión simple que es ejecutada por evaluación de la expresión. La cuestión está en QUÉ va a ser computado, no en CÓMO va a serlo.
Otro lenguaje muy conocido, casi funcional es el lenguaje de consultas estándar de bases de datos, SQL. Una consulta SQL es una expresión con proyecciones, selecciones y uniones. Una consulta dice qué relación se debe computar sin decir cómo debe computarse. Además, la consulta puede ser evaluada en cualquier orden que sea conveniente

Modelo Funcional

El modelo funcional, tiene como objetivo la utilización de funciones matemáticas puras sin efectos.
El esquema del modelo funcional es similar al de una calculadora. Se establece una sesión interactiva entre sistema y usuario: el usuario introduce una expresión inicial y el sistema la evalúa mediante un proceso de reducción. En este procesos se utilizan las definiciones de funciones realizadas por el programador hasta obtener un valor no reducible.

El valor que devuelve una función está únicamente determinado por el valor de sus argumentos consiguiendo que una misma expresión tenga siempre el mismo valor.
Es más sencillo demostrar la corrección de los programas ya que se cumplen propiedades matemáticas tradicionales como la propiedad conmutativa, asociativa, etc.
El programador se encarga de definir un conjunto de funciones sin preocuparse de los métodos de evaluación que posteriormente utilice el sistema. Estas funciones no tienen efectos laterales y no dependen de una arquitectura concreta. Este modelo promueve la utilización de una serie de características como las funciones de orden superior, los sistemas de inferencia de tipos, el polimorfismo, la evaluación perezosa, etc.

Funciones de orden superior

Un lenguaje utiliza funciones de orden superior cuando permite que las funciones sean tratadas como valores de 1ª clase, permitiendo que sean almacenadas en estructuras de datos, que sean pasadas como argumentos de funciones y que sean devueltas como resultados.
La utilización de funciones de orden superior proporciona una mayor flexibilidad al programador, siendo una de las características mas sobresalientes de los lenguajes funcionales.

```
quad :: Int -> Int
quad x = x * x
```

```
impar :: Int -> Bool
impar x
  | (x `mod` 2) == 1    = True
  | otherwise           = False

map :: (t -> u) -> [t] -> [u]
map f [] = []
map f (a:x) = f a : map f x


map quad [1,2,3,4] = [1,4,9,16]
map impar [1,2,3,4] = [True, False, True, False]
```

Sistemas de Inferencia de Tipos y Polimorfismo

Muchos lenguajes funcionales han adoptado un sistema de inferencia de tipos que consiste en:

- El programador no está obligado a declarar el tipo de las expresiones.
- El compilador contiene un algoritmo que infiere el tipo de las expresiones.
- Si el programador declara el tipo de alguna expresión, el sistema chequea que el tipo declarado coincide con el tipo inferido.

Los sistemas de inferencia de tipos permiten una mayor seguridad evitando errores de tipo en tiempo de ejecución y una mayor eficiencia, evitando realizar comprobaciones de tipos en tiempo de ejecución.
Los sistemas de inferencia de tipos aumentan su flexibilidad mediante la utilización de polimorfismo.

El polimorfismo permite que el tipo de una función dependa de un parámetro. Por ejemplo, si se define una función que calcule la longitud de una lista, una posible definición sería:

```
long ls = if vacia(L) then 0
          else 1 + long(cola (L))
long :: [x] → Integer
```

El sistema de inferencia de tipos inferiría el tipo: long::[x] → Integer, indicando que tiene como argumento una lista de elementos de un tipo a cualquiera y que devuelve un entero.
En un lenguaje sin polimorfismo sería necesario definir una función long para cada tipo de lista que necesitase. El polimorfismo permite una mayor reutilización de código ya que no es necesario repetir algoritmos para estructuras similares.

Evaluación Perezosa

Los lenguajes tradicionales, evalúan todos los argumentos de una función antes de conocer si estos serán utilizados.
Por ejemplo:

```
f (x:Integer; y:Integer):Integer
begin
  return (x+3);
end;

g (x:Integer):Integer
begin
(* bucle infinito *)
while true do
  x:=x
end;

-- Programa Principal
begin
  write (f(4,g(5)));
end;
```

Con el sistema de evaluación tradicional, el programador no devolvería nada, puesto que al intentar evaluar g(5) el sistema entraría en un bucle infinito. Dicha técnica de evaluación se conoce como evaluación impaciente porque evalúa todos los argumentos de una función antes de conocer si son necesarios.
Por otra parte, en ciertos lenguajes funcionales se utiliza evaluación perezosa que consiste en no evaluar un argumento hasta que no se necesita. En el ejemplo anterior, si se utilizase evaluación perezosa, el sistema escribiría 7.

¿Qué tienen de bueno los lenguajes funcionales?

Estos son algunas de las características más importantes de los lenguajes funcionales.

Examinemos algunos de los beneficios de la programación funcional:

1. **Programas cortos.** La brevedad de los programas funcionales hace que sean mucho más concisos que su copia imperativa.
2. **Facilidad de comprensión de los programas funcionales.** Deberíamos ser capaces de entender el programa sin ningún conocimiento previo del Haskell. No podemos decir lo mismo de un programa en C. Nos lleva bastante tiempo comprenderlo y, cuando lo hemos entendido, es muy fácil cometer un pequeño fallo y tener un programa incorrecto.
3. **No hay ficheros "core".** La mayoría de los lenguajes funcionales y Haskell en particular son fuertemente tipados y eliminan una gran cantidad de clases que se crean en tiempo de compilación con las que se pueden cometer errores. O sea, fuertemente tipados significa que no hay ficheros "core". No hay posibilidad de tratar un puntero como un entero o un entero como un puntero nulo.
4. **Reutilización de código.** Los tipos fuertes están, por supuesto, disponibles en muchos lenguajes imperativos como Ada o Pascal. Sin embargo el sistema de tipos de Haskell es mucho menos restrictivo que, por ejemplo el de Pascal porque usa polimorfismo. Por ejemplo, el algoritmo Quicksort se puede implementar de la misma manera en Haskell para listas de enteros, de caracteres, listas de listas... mientras que la versión en C es sólo para arrays de enteros.
5. **Plegado.** Los lenguajes funcionales no estrictos tienen otra característica, la evaluación perezosa. Los lenguajes funcionales no estrictos llevan exactamente esta clase de evaluación. Las estructuras de datos son evaluadas justo en el momento en el que se necesita una respuesta y puede que haya parte de estas estructuras que no se evalúen
6. **Abstracciones potentes.** Generalmente, los lenguajes funcionales ofrecen nuevas formas para encapsular abstracciones. Una abstracción permite definir un objeto cuyo trabajo interno está oculto. Por ejemplo, un procedimiento en C es una abstracción. Las abstracciones son la clave para construir programas con módulos y de fácil mantenimiento. Son tan importantes que la pregunta para todo nuevo lenguaje es: "¿De qué mecanismos de abstracción dispone?".
   Un mecanismo de abstracción muy potente que está disponible en los lenguajes funcionales son las funciones de alto orden. En Haskell una función es un "ciudadano de primera clase": pueden pasarse tranquilamente a otras funciones, ser devueltas como el resultado de otra función, ser incluidas en una estructura de datos, etc. Esto nos quiere decir que el buen uso de estas funciones de alto orden puede mejorar sustancialmente la estructura y modularidad de muchos programas.
7. **Manejo de direcciones de memoria.** Muchos programas sofisticados necesitan asignar memoria dinámica desde una pila. En C, esto se hace con una llamada a "malloc", seguida de un código para inicializar la memoria. El programador es el responsable de liberar memoria cuando ya no se necesite más. Esto produce, muchas veces, errores del tipo "dangling-pointer" (punteros colgados).
   Cada lenguaje funcional libera al programador del manejo de este almacenamiento. La memoria es asignado e inicializado implícitamente y es recogido por el recolector de basura. La tecnología de asignación del store y la recolección de basura está muy bien desarrollada y los costes son bastante insignificantes.

Cálculo lambda

El cálculo lambda es un sistema formal diseñado para investigar la definición de función, la noción de aplicación de funciones y la recursión. Fue introducido por Alonzo Church y Stephen Kleene en la década de 1930; Church usó el cálculo lambda en 1936 para resolver el Entscheidungsproblem[1]. Puede ser usado para definir de manera limpia y precisa qué es una "función computable".

Church resolvió negativamente el Entscheidungsproblem: probó que no había algoritmo que pudiese ser considerado como una "solución" al Entscheidungsproblem.

El cálculo lambda ha influenciado enormemente el diseño de lenguajes de programación funcionales, especialmente LISP.

Se puede considerar al cálculo lambda como el más pequeño lenguaje universal de programación. Consiste de una regla de transformación simple (substitución de variables) y un esquema simple para definir funciones.

El cálculo lambda es universal porque cualquier función computable puede ser expresada y evaluada a través de él. Por lo tanto, es equivalente a las máquinas de Turing. Sin embargo, el cálculo lambda no hace énfasis en el uso de reglas de transformación y no considera las máquinas reales que pueden implementarlo. Se trata de una propuesta más cercana al software que el hardware.

Church redujo todas las nociones del cálculo de sustitución. Normalmente, un matemático debe definir una función mediante una ecuación.

Por ejemplo, si una función f es definida por la ecuación f(x)=t, donde t es algún término que contiene a x, entonces la aplicación f(u) devuelve el valor t[u/x], donde t[u/x] es el término que resulta de sustituir u en cada aparición de x en t.
Por ejemplo, si f(x)=x*x, entonces f(3)=3*3=9.

Lambda Expresiones

Church propuso una forma especial (más compacta) de escribir estas funciones. En vez de decir "la función f donde f(x)=t", él simplemente escribió \x.t.
Para el ejemplo anterior: \x.x*x.

Un término de la forma \x.t se llama "lambda expresión".

La principal característica de lambda cálculo es su simplicidad ya que permite efectuar solo dos operaciones:

- Definir funciones de un solo argumento y con un cuerpo especifico, denotado por la siguiente terminología: λ**x.B**, en donde x determina el parámetro o argumento formal y B representa el cuerpo de la función, es decir f(x) = B.

Ejemplo 1:

Para la función
f: A $\rightarrow$ B
f(x) $\rightarrow$ 2x + 1
Podemos escribirla como una λ-expresión de la forma
λx . 2x+1

Ejemplo 2:

f(x, y) = (x + y)*2
λx $\rightarrow$ λy $\rightarrow$ (x + y)* 2
Se expresaría en λ-Calculo como

[1] El Entscheidungsproblem (en castellano: problema de decisión) es el reto en lógica simbólica de encontrar un algoritmo general que decida si una fórmula del cálculo de primer orden es un teorema. (Un teorema es una afirmación que puede ser demostrada como verdadera dentro de un marco lógico.)

λx. λy. * (+ x y) 2

- Aplicar alguna de las funciones definidas sobre un argumento real (A); lo que es conocido también con el nombre de reducción, y que no es otra cosa que sustituir las ocurrencias del argumento formal (x), que aparezcan en el cuerpo (B) de la función, con el argumento real(A), es decir: (λx.B) A.

Ejemplo 3:

(λ x. (x+5)) 3

lo que indica que en la expresión x+5, se debe sustituir el valor de x por 3.

Los dos mecanismos básicos presentados anteriormente se corresponden con los conceptos de **abstracción funcional** y **aplicación de función**; si le agregamos un conjunto de identificadores para representar **variables** se obtiene lo mínimo necesario para tener un lenguaje de programación funcional. Lambda calculo tiene el mismo poder computacional que cualquier lenguaje imperativo tradicional.

La sintaxis BNF para las λE es:

exp	::=	cons	constante predefinida
		var	identificador de variable
		(λ var . exp)	abstracción
		(exp exp)	aplicación

Reducción de Expresiones

La labor de un evaluador es calcular el resultado que se obtiene al simplificar una expresión utilizando las definiciones de las funciones involucradas.

Ej: doble :: Integer → Integer
doble x = x + x

```
5 * doble 3
5 * ( 3 + 3 )    { por el operador + }
5 * 6            { por el operador * }
30
```

Una expresión se reduce sustituyendo, en la parte derecha de la ecuación de la función, los Parámetros Formales por los que aparecen en la llamada (también llamados Parámetros Actuales o Parámetros).

Cada paso es una reducción.
Un redex es cada parte de la expresión que pueda reducirse.
Cuando una expresión no puede ser reducida mas se dice que esta en forma normal.

¿Qué ocurre si hay más de un redes? por ejemplo en doble (doble 3)

Se puede reducir la expresión desde dentro hacia fuera (primero los redex internos)
Otra estrategia consiste en reducir desde fuera hacia dentro (primero los redex externos)

El valor obtenido de la función siempre dependerá únicamente de los argumentos y siempre tendrá la consistencia de retornar el mismo valor para los mismos argumentos. Por lo tanto se tiene **transparencia referencial**.

Ordenes de evaluación

Es importante el orden en el que se aplican las reducciones, y dos de los mas interesantes son: Aplicativo y Normal.

Orden Aplicativo

Se reduce siempre el término MAS INTERNO (el más anidado en la expresión). En caso de que existan varios términos a reducir (con la misma profundidad) se selecciona el que aparece más a la izquierda de la expresión. Esto también se llama paso de parámetros por valor (call by value), ya que ante una aplicación de una función, se reducen primero los parámetros de la función.

A los evaluadores que utilizan este tipo de orden, se les llama estrictos o impacientes

```
doble (doble 3)          por la definición de doble
doble (3 + 3)            por el operador +
doble (6)                por la definición de doble
6 + 6                    por el operador +
12
```

Orden Normal

Consiste en seleccionar el término MÁS EXTERNO (el menos anidado), y en caso de conflicto el que aparezca más a la izquierda de la expresión.
Esta estrategia se conoce como "paso de parámetro por nombre o referencia" (call by name), ya que se pasan como parámetros de las funciones expresiones en vez de valores.
A los evaluadores que utilizan el orden normal se les llama "no estrictos".
Una de las características más interesantes es que este orden es normalizante.

```
doble (doble 3)          por la definición de doble
(doble 3) + (doble 3)    por la definición de doble
(3 + 3) + (doble 3)      por la definición de +
6 + (doble 3)            por la definición de doble
6 + (3 + 3)              por la definición de +
6 + 6                    por la definición de +
12
```

Evaluación PEREZOSA o LENTA (Lazy)

No se evalúa ningún elemento en ninguna función hasta que no sea necesario. Las listas se almacenan internamente en un formato no evaluado. La evaluación perezosa consiste en utilizar paso por nombre y recordar los valores de los argumentos ya calculados para evitar recalcularlos.
También se denomina *estrategia de pasos de parámetros por necesidad (call by need).*

Con una estrategia no estricta de la expresión doble (doble 3), la expresión (3 + 3) se calcula dos veces.

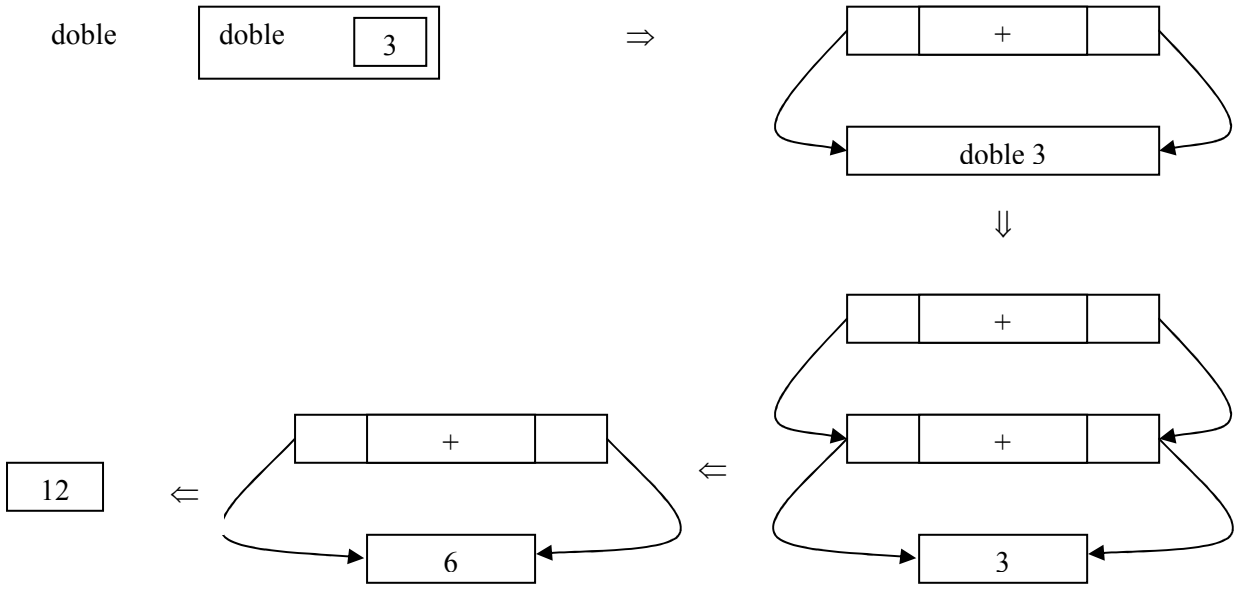
Este tipo de evaluación es útil para trabajar con listas infinitas

Ejemplo:

```
ones = 1 : ones
```
– Lista con un número infinito de 1s

El problema que tiene este tipo de evaluación es que algunas expresiones se reducen varias veces como por ejemplo, en la expresión `doble (doble 3)`; la expresión (3 + 3) se reduce dos veces, lo que no ocurre en el caso de evaluación impaciente.

```
doble (doble 3)                          por la definición de doble
a + a       donde a = doble 3            por la definición de doble
a + a donde a = b + b   donde b = 3      por el operador +
a + a donde a = 6                        por el operador +
12
```



Estructuras de datos infinitas y Evaluación Perezosa.

Un beneficio particular de la evaluación perezosa es que hace posible manipular estructura de datos infinitas. Por supuesto que no podemos construir o almacenar una estructura infinita completamente. La ventaja de la evaluación perezosa es que permite construir objetos infinitos pieza por pieza según sea necesario.

Como un ejemplo simple, consideremos la siguiente función que puede usarse para construir listas infinitas de valores enteros.

```
countFrom n = n : countFrom (n+1)
```

Si evaluamos la expresión "`countFrom 1`", podremos ver que se forma una lista de valores enteros desde 1 hasta que se intrrumpe la ejecución del programa. Evaluar esta expresión equivale a ejecutar un loop infinito que imprima la lista de valores en un lenguaje imperativo.

```
? countFrom 1
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,^C{Interrupted!}
(53 reductions, 160 cells)
?
```

Para aplicaciones prácticas, solamente nos interesará construir la lista hasta la posición que nos interese. Por ejemplo al usar "`countFrom`" junto con la función "`take`" podemos tomar solo los primeros 10 enteros y encontrar su suma.

```
? sum (take 10 (countFrom 1))
55
(62 reductions, 119 cells)
?
```

Transparencia Referencial

Principio de Transparencia Referencial

"Si en una expresión sintácticamente correcta se cambia una subexpresión por otra, también correcta, que denote el mismo objeto, la expresión resultante denotará el mismo objeto que la expresión inicial."

Según esto, el orden de reducción de las subexpresiones reducibles de una expresión no debe alterar el resultado.

Sea cual fuere la estrategia seguida, el resultado final en ambos tipos de evaluación será el mismo valor (en el ejemplo: 12).
Si aparecen varios redex podemos elegir cualquiera, sin embargo, la reducción de un redex equivocado, puede que no conduzca a la forma normal de una expresión:

```
infinito :: Integer
infinito = 1 + infinito
cero :: Integer → Integer
cero x = 0
```

Si en cada momento se elige el redex más interno ocurriría lo siguiente:

```
cero infinito                por la definición de infinito
cero (1 + infinito)          por la definición de infinito
cero (1 + (1 + infinito)     por la definición de infinito
...
```

la evaluación no terminaría nunca.

Si elegimos el redex más externo:

```
cero infinito                por la definición de cero
0
```

La selección de la estrategia utilizada para seleccionar el redex es crucial.

Con la evaluación impaciente podríamos efectuar reducciones que no son necesarias

```
cero (10 * 4)           por la definición de *
cero 40                 por la definición de cero
0
```