Paradigma Funcional
(Material complementario)

Ing. Lucas Spigariol - 2007

Conceptos generales

A diferencia de otros paradigmas donde un programa describe en detalle los pasos que la máquina debe ejecutar para realizar el ordenamiento, con la mayoría del código ocupándose de los detalles de bajo nivel de la manipulación de datos, un programa funcional resuelve el problema de ordenamiento en un **más alto nivel**, con una **mejora significativa en la brevedad y claridad**. Los programas son más fáciles de diseñar, de escribir y de mantener, pero dan al programador menos posibilidades de control sobre la máquina.

La escritura de sistemas de software grandes que el trabajo es difícil y cara, y su mantenimiento lo es aún más. Lenguajes de programación funcionales, pueden hacerlo más fácil y más barato.

Buena parte del tiempo de vida de un producto de software se utiliza en la especificación, el diseño y el mantenimiento, y no en la programación. Los lenguajes funcionales son magníficos para escribir especificaciones que pueden ser ejecutadas realmente (y por lo tanto, probado y eliminado errores). Tal especificación entonces es el primer prototipo del programa final.

Los programas funcionales son también relativamente fáciles de mantener, porque el código es más corto, más claro, y el control riguroso del efecto de lado elimina una enorme cantidad de interacciones imprevistas. [1]

Principales características

La **transparencia referencial** permite que el valor que devuelve una función esté únicamente determinado por el valor de sus argumentos consiguiendo que una misma expresión tenga siempre el mismo valor.

Permite aumentar su flexibilidad y realizar unidades de software genéricas mediante la utilización **de tipos de datos genéricos**, que es una forma de implementar el **polimorfismo** en el paradigma funcional.

La **evaluación perezosa o diferida** (*lazy evaluation*) en una característica de gran potencia, por la que se evalúa del programa solamente tanto como se requiere para conseguir la respuesta. En la invocación de funciones con sus correspondientes argumentos, la evaluación de las expresiones intervinientes se posterga hasta el momento en que realmente sean utilizadas.

Una herramienta para la formulación de algunas soluciones, principalmente en las más de bajo nivel, es la **recursividad,** basada en el principio matemático de inducción, que se ve

[1] Sitio oficial de Haskell (www.haskell.org)

expresada en el uso de tipos de datos recursivos, como las listas, y funciones recursivas que las operan.

Una característica fundamental es la posibilidad de tratar a las funciones como datos mediante la definición de **funciones de orden superior**, que permiten un gran nivel de abstracción y genericidad en las soluciones. el uso correcto de las funciones de orden superior puede mejorar substancialmente la estructura y la modularidad de muchos programas.

Los lenguajes funcionales relevan al programa de la compleja tarea de gestión de memoria. El almacenamiento se asigna y se inicializa implícitamente, y es recuperado automáticamente por un recolector de la basura (*garbage collector*). La tecnología para la asignación de memoria y de la recolección de la basura están actualmente bien desarrollados por lo que su costo de funcionamiento es leve.

Caso práctico: Planilla de cálculo

Cualquier persona que ha utilizado una planilla de cálculo tiene experiencia de la programación funcional. En una hoja de balance, uno especifica el valor de cada celda en términos de valores de otras celdas. El foco está en qué debe ser computado, no cómo debe serlo. Por ejemplo:

- **No se especifica el orden** en la cual las celdas deben ser calculadas, ya que es la planilla de cálculo quien computa las celdas en un orden que respete sus dependencias.
- **No se le dice** a la planilla de cálculo **cómo asignar su memoria**, se espera que presente un panorama al parecer infinito de celdas, y que asigne memoria solamente a esas celdas cuando realmente las utilice.
- En general, se especifica el valor de una celda **mediante una expresión cuyas piezas se puedan evaluar en cualquier orden**, **antes que por una secuencia de comandos que compute su valor.**

Una consecuencia interesante de la no especificación de orden de la planilla de cálculo, es que la noción de la asignación no es muy útil, ya que si no se sabe exactamente cuándo sucederá una asignación, no se puede hacer mucho uso de ella. Esto, lo une al paradigma con el que comparte la centralidad de la declaratividad y lo pone un contraste fuerte con programas en los lenguajes de paradigmas procedurales, que consisten esencialmente en una secuencia cuidadosamente especificada de asignaciones, o de objetos, en la cual el ordenar de las llamadas del método es crucial para el significado de un programa.

Este foco en el de alto nivel, de especificar el "qué" antes que en el bajo nivel del "cómo" es una característica el distinguir de lenguajes de programación funcionales.

Otro lenguaje con fuertes rasgos funcionales bien conocido es el lenguaje de consulta estándar de base de datos SQL. Una consulta del SQL es una expresión que implica las proyecciones, selecciones, ensambles y así sucesivamente. La pregunta dice qué relación debe ser computada, sin decir cómo debe ser computada. De hecho, la pregunta se puede evaluar en cualquier orden conveniente. Las puestas en práctica del SQL realizan a menudo la optimización extensa de la pregunta que (entre otras cosas) calcula hacia fuera la mejor orden para evaluar la expresión.

Caso práctico: QuickSort

Las planillas de cálculo y el SQL son ejemplos bastante especializadas. Los lenguajes de programación funcionales toman las mismas ideas y las mueven en el reino de la programación de uso general. Para tener una idea de como es un programa funcional, y la expresividad de sus lenguajes, se pueden ver los siguientes programas que realizan un ordenamiento rápido (quicksort).

El primer programa se escribe en un lenguaje del paradigma funcional, Haskell, y el otro en un lenguaje imperativo, C. [2]

Mientras que el programa en C describe en detalle los pasos que la máquina debe ejecutar para realizar el ordenamiento -con la mayoría del código ocupándose de los detalles de bajo nivel de la manipulación de datos- el programa de Haskell resuelve el problema de ordenamiento en un más alto nivel, con una mejora significativa en la brevedad y claridad.

Ejemplo de Quicksort en Haskell

```
qsort [ ]     = [ ]
qsort (x:xs) = qsort (filter (< x) xs) ++ [x] ++ qsort (filter (>= x) xs)
```

Ejemplo de Quicksort en C

```
void qsort(int a[], int lo, int hi) {
{
  int h, l, p, t;
  if (lo < hi) {
    l = lo;
    h = hi;
    p = a[hi];
    do {
      while ((l < h) && (a[l] <= p))
         l = l+1;
      while ((h > l) && (a[h] >= p))
         h = h-1;
      if (l < h) {
         t = a[l];
         a[l] = a[h];
         a[h] = t;
      }
    } while (l < h);
    t = a[l];
    a[l] = a[hi];
    a[hi] = t;
    qsort( a, lo, l-1 );
    qsort( a, l+1, hi );
  }
}
```

[2] Ejemplos de implementación extraídos del sitio oficial de Haskell (www.haskell.org)

Los programas funcionales tienden a ser mucho más concisos que las contrapartes imperativas. El quicksort es un caso extremo, pero en general los programas son más cortos (en una proporción de 2 a 10).

Los programas funcionales son a menudo más fáciles de entender. Se debe poder entender el programa sin mucho conocimiento anterior del Haskell o del quicksort. Lo mismo ciertamente no pueden ser dichos del programa de C. Toma un tiempo entender, y aún cuando se lo logra, es muy fácil cometer un pequeño error y terminar con un programa incorrecto.

Una explicación detallada del quicksort de Haskell:

> La primera línea se lee: "El resultado de ordenar una lista vacía ([ ]) es otra lista vacía". La segunda: "El resultado de ordenar una lista, cuyo primer elemento se denomina *x* y el resto se denomina *xs*, es la concatenación (++) de las listas que se obtienen de ordenar los elementos de *xs* que son menores que *x* y de ordenar los elementos de *xs* que son mayores o iguales a *x*, con *x* intercalado en el centro."

En Haskell, el programa no sólo ordenará listas de números enteros, sino también listas de números de punto flotante, listas de caracteres, listas de listas; de hecho, ordenará listas de cualquier cosa para las cuales tenga sentido tener las operaciones "menor" y "mayor". En cambio, la versión de C puede ordenar solamente un conjunto de números enteros. Se ve cómo el polimorfismo permite reutilizar el código.

El quicksort de C utiliza una técnica ingeniosa, por la que ordena el conjunto sin usar ningún almacenamiento adicional. Consecuentemente, funciona rápidamente y en una cantidad pequeña de memoria. En cambio, el programa de Haskell asigna bastante memoria adicional detrás de lo que se ve y funciona algo más lento que el otro programa.

En un lenguaje imperativo, el quicksort asume esta complejidad algorítmica para lograr una reducción en el tiempo de ejecución. En los usos donde se requiere la mejor performance a cualquier costo, o cuando la meta es ajustar minuciosamente un algoritmo de bajo nivel, un lenguaje imperativo como C sería probablemente una opción mejor que Haskell, precisamente porque proporciona un control más íntimo sobre la manera exacta de la cual se realiza el cómputo.

Por otra parte, son pocos los programas que requieren performance a cualquier costo. Después de todo, hace tiempo que se ha dejado de escribir programas de lenguaje assembler, excepto quizás para ciertas aplicaciones de más bajo nivel, porque las ventajas del tener un modelo de programación de soporte compensan con creces el costo de un tiempo de ejecución más modesto.

Con el paradigma funcional sucede algo similar: Los lenguajes funcionales tienden hacia un modelo de programación de alto nivel, con una ganancia en el diseño, desarrollo y mantenimiento y una pérdida relativa de control sobre la máquina, pero que para muchos programas es un precio perfectamente aceptable para el buen resultado que se obtiene.

Lenguaje Haskell

Haskell es un **lenguaje funcional puro**, de propósito general, que incluye muchas de las últimas innovaciones en el desarrollo de los lenguajes de programación funcional, como son las funciones de orden superior, evaluación perezosa (*lazy evaluation*), tipado polimórficos, tipos definidos por el usuario, encaje de patrones (*pattern matching*), y definiciones de listas por comprensión.

Su nombre proviene de **Haskell Brooks Curry**, quien trabajó en la lógica matemática que sirve como fundamento de los lenguajes funcionales. **Haskell** se basa en el ***cálculo lambda***, por lo tanto el signo de lambda (**λ**) es utilizado como ícono.

Hay un amplio número de compiladores y de intérpretes disponibles, la mayoría libres. Uno de ellos, de simple manejo y amplitud de prestaciones es el intérprete **Hugs** (basado en el

Gofer, un sistema experimental de programación funcional, presentado en 1991), que ofrece una gran compatibilidad con el estándar **Haskell**. En realidad, el nombre **Hugs** fue en principio escogido como un mnemónico de "*Haskell Users' Gofer Sistem*"

En este trabajo, los ejemplos se desarrollan utilizando la versión que se denomina *WinHugs 98.*[3]

Listas por comprehensión

La notación de listas por comprehensión permite declarar de forma concisa una gran cantidad de iteraciones sobre listas. Esta notación está adaptada de la teoría de conjuntos. Sin embargo, se trabaja con listas, no con conjuntos. El formato básico de la definición de una lista por comprensión es una expresión genérica, que define a cada elemento de la lista, y una serie de cualificadores, que determinan las características, posibilidades y restricciones para cada uno de ellos.

Ejemplo:

[expresion | cualificador1, cualificador2 … cualificadorN]

Los cualificadores son de dos tipos:

- **Generadores:** Un cualificador de la forma **patron <- lista** es utilizado para extraer de la **lista,** en el mismo orden en que están, cada elemento que unifique con el **patrón**.

Ejemplo:

[x*x | x <- [1..10]]

[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

El cualificador generador **x <- [1..10]** hace que la variable **x** asuma cada uno de los valores de la lista **[1..10]**. La expresión **x * x** indica que el contenido de la lista va a ser el resultado de evaluar la expresión para cada uno de los valores **x** cualificados.

- **Filtros:** Es una expresión de tipo booleana que se utiliza combinada con los generadores, que actúa como restricción sobre los valores generados, de manera que sólo van a ser tenidos en cuenta para integrar la lista aquellos que satisfagan la expresión.

Ejemplo:

[x*x | x <- [1..10] , even x]

[4, 16, 36, 64, 100]

Se comporta como en el ejemplo anterior, con la diferencia que la expresión **x*x** solo se evalúa sobre los elementos generados que además verifiquen el cualificador **even x**, es decir, aquellos que sean pares.

Cuando aparecen varios cualificadores hay que tener en cuenta que las variables generadas por los cualificadores posteriores varían más rápidamente que las generadas por los cualificadores anteriores

[3] Se puede ampliar y descargar el software gratuitamente en www.haskell.org/hugs

Ejemplo:

[(x, y) | x<- [1..3], y <- [1..2]]

[(1,1), (1,2), (2,1), (2,2), (3,1), (3,2)]

Representa una lista de tuplas de dos de la forma **(x,y)**, donde **x** varía entre **1** y **3** e **y** lo hace entre **1** y **2**. En otras palabras , genera todas las combinaciones.

Por otra parte, los cualificadores posteriores pueden utilizar los valores generados por los anteriores

Ejemplo:

[(x, y) | x <- [1..3], y <- [1..x]]

[(1,1), (2,1), (2,2), (3,1), (3,2), (3,3)]

Representa una lista de tuplas de dos de la forma **(x,y)**, donde **x** varía entre **1** y **3** e **y** lo hace entre **1** y el valor de **x**. Por lo tanto, para **x** = **1**, **y** vale sólo **1;** para **x** = 2, **y** asume los valores **1** y **2;** y por último, para **x** = **3**, **y** vale **1**, **2** y **3**.

El orden en que se indican los cualificadores es importante. Puede tener efecto directo en la eficiencia. Las variables definidas en cualificadores posteriores ocultan las variables definidas por cualificadores anteriores. No es una buena costumbre reutilizar los nombres de las variables, ya que dificulta la comprensión de los programas, aunque las expresiones sean válidas.

Ejemplo:

[x | x <- [[1,2], [3,4]], x <- x]

[1, 2, 3, 4]

Inferencia de Tipos

El sistema de inferencia de tipos consiste en que el sistema contiene un mecanismo que infiere el tipo de dato de las expresiones, por lo que **no es obligatoria la declaración el prototipo de las funciones**.

Cuando el programador declara el tipo de alguna expresión, el sistema chequea que el tipo declarado coincide con el tipo inferido. De esta manera, la explicitación de los prototipos **permite una mayor seguridad** evitando errores de tipo en tiempo de ejecución y una mayor eficiencia, evitando realizar comprobaciones de tipos en tiempo de ejecución.

Polimorfismo

Remitiendo a las ideas básicas del polimorfimo, que consiste en construir piezas de software genéricas que trabajen indistintamente con diferentes tipos de entidades y que las puedan considerar como intercambiables, se puede afirmar que dicha situación se aplica en el paradigma funcional, al definir funciones cuyos argumentos pueden ser indistintamente, de uno u otro tipo de dato.

Los sistemas de inferencia de tipos aumentan su flexibilidad mediante la utilización de polimorfismo, que permite que el tipo de una función dependa de un parámetro. El polimorfismo tiene como gran beneficio una mayor reutilización de código ya que no es necesario repetir algoritmos para estructuras similares.

Variables de tipos de datos

Hay muchas funciones en las que las ecuaciones y expresiones que determinan su funcionalidad son las mismas independientemente del tipo de dato que reciben o devuelven. En estos casos, la utilización **de tipos de datos genéricos**, mediante variables de tipos de datos, permite implementar el polimorfismo para aumentar la flexibilidad y genericidad de las funciones, definiéndolas una sola vez y utilizándolas para diferentes tipos de datos.

En la declaración del prototipo de una función, en vez de especificar un tipo de dato en particular, se utiliza **una variable de tipo de dato**. En el momento de la invocación, esta variable infiere su valor a partir del tipo de dato de los argumentos. La utilización de ese argumento que se haga en el interior de la función debe ser lo suficientemente genérica para que funcione correctamente en todos los casos, ya que puede ser en algunas ocasiones de un tipo de dato y en otras ocasiones de otro.

Funciones de orden superior

La programación funcional incorpora el concepto de función como **objeto de primera clase**, lo que significa que **las funciones son tratadas como datos** y en consecuencia pueden ser pasadas como parámetros, calculadas, devueltas como resultados, incluidas en una estructura de datos o utilizadas en cálculos más complejos con otros datos.

Los lenguajes funcionales ofrecen generalmente maneras potentes de encapsular abstracciones. Las abstracciones son la clave de la construcción modular y de los programas mantenibles. Un mecanismo poderoso de abstracción disponible en los lenguajes funcionales son las funciones de orden superior.

Las funciones que reciben a otras funciones como parámetros se llaman funciones de orden superior. En otras palabras **una función es de orden superior cuando existe al menos uno de sus argumentos cuyo tipo de dato es función**.

Aplicación Parcial

Todas las funciones **devuelven otra función,** cuando son **evaluadas con una menor cantidad de argumentos** de los que tiene definidos.

Los tipos de datos en el prototipo de una función, están implícitamente asociados del final hacia el principio.

Ejemplo:

Una función de definida de la siguiente manera

```
funcion :: Tipo1 -> Tipo2 -> ... -> TipoN-1 -> TipoN
```

Es interpretada por defecto como si tuviese paréntesis:

```
funcion :: Tipo1 -> (Tipo2 -> (... -> (TipoN-1 -> TipoN)))
```

En una función de "n" argumentos, al invocarse con los argumentos 1 al "k", la función que se retorna tiene su prototipo formado por los tipos de datos "k +1"hasta "n". En general, la función que devuelve tiene los argumentos que no hayan sido invocados.

Ejemplo:

Al invocar a la función anterior con menos argumentos de los que se supone que recibe, de la forma

```
funcion argumento1
```

se obtiene como resultado una nueva función, cuyo tipo de dato es

Tipo2 -> ... -> TipoN-1 -> TipoN

La función resultante se comporta como función en el lugar donde se la ubique. Para que tenga sentido, será evaluada en un contexto donde se le completen los demás argumentos para que devuelva un valor que no sea otra función.

La utilidad principal de este proceso de **evaluación parcial de funciones es** permite crear nuevas funciones a partir de las funciones existentes.

Ejemplo:

La función de suma (+) recibe dos valores enteros como argumento y devuelve un entero. Su tipo de dato es

(+) :: Int->Int->Int

que explicitando la precedencia con paréntesis es equivalente a

(+) :: Int->(Int->Int)

Por lo tanto, al invocar

5 +

se obtiene una función cuyo tipo es **Int->Int**, o sea, que recibe un entero y devuelve otro entero. De hecho, esta expresión denota una nueva función, que en caso de aplicarse sobre un entero devuelve dicho entero más 5:

(5 +) 3

8

El mismo caso deja de ser trivial al utilizarlo en un nuevo contexto, como siendo argumento de **map**. La invocación:

map (5 +) [1, 4, 6]

[6, 9, 11]

Ejemplos:

(1+) es la función sucesor que devuelve su argumento más 1.

(1.0/) es la función inverso.

(/2) es la función mitad.

(:[]) es la función que convierte un elemento simple en una lista con un único elemento que lo contiene

Existe un caso especial importante. Una expresión de la forma:

(-e) es interpretada como **negate e**, no como la aplicación parcial de **"-"** que resta el valor de **e** de su argumento.

Su uso está relacionado con el concepto de **currificación**[4], que plantea la utilización de argumentos individuales en vez de estructuras de tipo tuplas en el prototipo de la función. De esta manera, es condición necesaria para que se pueda dar la aplicación parcial. Solo puede evaluarse parcialmente una función de varios parámetros si está currificada.

Ejemplo

Una definición sencilla de una función

[4] En honor a Haskell Curry.

```
multiplicar :: ( Int, Int ) -> Int
multiplicar (x, y) = x * y
```

Así definida, **multiplicar** no esta currificada, ya que recibe un sólo argumento, que es una tupla. La versión currificada de la misma función es la siguiente, con dos argumentos simples:

```
multiplicar :: Int -> Int -> Int
multiplicar x y  = x * y
```

La aplicación parcial de **multiplicar** sólo es posible con la segunda definición.

Invocando

```
multiplicar 2
```

Se obtiene una nueva función que calcula el doble de un número entero

Expresiones lambda

Además de las definiciones de función con nombre, es posible definir y utilizar funciones sin necesidad de darles un nombre explícitamente, mediante las expresiones lambda.

Una expresión lambda tiene una estructura similar a la de una función. De hecho, las ecuaciones típicas que componen una función son formas abreviadas de las expresiones lambda, y todo el desarrollo matemático denominada cálculo lambda está en los fundamentos mismos del paradigma funcional.

En general, una expresión lambda está constituida por una serie de variables, que hace las veces del dominio de la función, y una expresión, que equivale a la imagen de la función.

Ejemplo

Una función lambda tiene la siguiente forma

```
\ variable1 variable2 .. variableN -> expresion1
```

La expresión denota una función que toma un número de parámetros (uno por cada variable, produciendo el resultado especificado por expresion1.

Ejemplo

Una función como

```
inc x = x+1
```

es equivalente a la expresión lambda

```
\x -> x+1
```

Al hacer la invocación

```
(\x -> x+1) 6
```

el resultado es

```
7
```

Ejemplo

```
\x y -> x + y
```

Toma dos argumentos enteros y devuelve su suma. Es una expresión equivalente al operador (+):

```
(\x y -> x + y ) 2 3
5
```

Las expresiones lambda son utilizadas frecuentemente en conjunto con las funciones de orden superior, actuando como argumentos.

Ejemplo

Si se quiere obtener una lista con el cuadrado de los primeros números naturales, una posibilidad, invocando a la función de orden superior, es:

```
map  cuadrado [1..5]
[1, 4, 9, 16, 25]
```

Esta solución requiere de la definición de la función cuadrado, que puede estar definida así:

```
cuadrado x = x * x
```

Para obtener la misma funcionalidad, la alternativa utilizando una expresión lambda es invocar:

```
map  ( \x -> x * x ) [1..5]
```