Programación Funcional

Herrera, Maximiliano; Jungblut, Nicolás Eduardo; Samblancat, Matías Leonel; Taguchi, Diana

Universidad Argentina de la Empresa

Abstract

El presente documento consiste de una investigación sobre el paradigma: "programación funcional", partiendo de sus orígenes para conocer y evaluar la problemática que permite resolver, su evolución, y las principales características que lo convierten en una importante opción ante un desarrollo de software.

Asimismo se presentan las desventajas de este paradigma y su contraste con el paradigma orientado a objetos.

Sumado a lo anterior, se proponen ejemplos utilizando el lenguaje Heskell.

Finalmente se presenta un caso de éxito: "FACEBOOK", que mediante la aplicación de chat, hace uso de la programación funcional.

Palabras Clave

Programación Funcional, Paradigma de desarrollo, Heskell.

Objetivos

El propio nombre nos da una noción de la característica fundamental de este paradigma: se basa fuertemente en un concepto de función algo diferente al que estamos acostumbrados, más cercano al matemático que su contraparte en los lenguajes imperativos. No hay variables, por lo que estas funciones sólo tratarán con sus valores de entrada y con constantes predefinidas; no tienen más posibilidad de acción.

Alcances

Definir el concepto del paradigma de programación funcional, así como sus principales características. Asimismo se presentará ejemplos del paradigma mediante el lenguaje de programación Haskell.

De igual modo, se expondrá uno de los casos de éxito del lenguaje funcional.

Introducción

Muchas operaciones matemáticas se realizan mecánicamente mediante un "algoritmo" y otras son más difíciles de automatizar, por ejemplo, la demostración de un teorema. La invención del Cálculo por parte de Newton y Leibniz mostró a los matemáticos de la época como una notación adecuada podía hacer que operaciones complicadas se simplificaran, y surgió la idea de que con una notación adecuada, toda la matemática se puede hacer mecánica, y por lo tanto podría concebirse una máquina que hiciera todo el trabajo. Esto originó diversas investigaciones las cuales crearon un nuevo paradigma llamado programación funcional, basado fundamentalmente en la utilización de funciones aritméticas.

Marco histórico

Para definir una forma aritmética, dos personas llegaron independientemente, por vías muy diferentes, quienes aportaron importantes investigaciones. Uno fue Alan Turing, quien ofreció un modelo teórico de una "máquina de cómputo programable" (la «máquina de Turing»). El otro fue Alonso Church, quien creó un modelo lógico para definir "decidible"[1] (el «cálculo lambda»[2], muy inspirado en la «lógica combinatoria» de Haskell Curry[3]).

El modelo de Turing es una máquina programable. Hasta entonces, cuando se

[1] **Decidible**: Que en un sistema lógico, es posible demostrar si es verdadero o falso.

[2] **Cálculo lambda**: es un sistema formal diseñado para investigar la definición de función, la noción de aplicación de funciones y la recursión.

[3] **Haskell Curry**: 12 de septiembre de 1900 - 1 de septiembre de 1982) fue un matemático y lógico estadounidense.

pensaba en una máquina, era solo de una tarea concreta. Turing dio el modelo teórico de "la máquina que puede hacer lo que cualquier otra pudiera hacer". Esta fue llamada la "máquina de Turing"[4] la cual se basa en cambios de estado (parte de un estado inicial en el que tiene un número y luego pasa a otro estado que corresponde a otro número).

Church utiliza el cálculo lambda para describir las relaciones funcionales entre los datos y el resultado. Todas las estructuras con las que trabaja el cálculo lambda son funciones. Matemáticamente hablando, una función es estática ("inmutable"), y esa es la diferencia más importante con el modelo de Turing: en el cálculo lambda, no hay cambios de estado.

//el calculo lambda permite definir que funciones son computables, es decir, que características debe tener una función para ser computable.

Church y Turing llegaron a la misma conclusión: **la aritmética NO es decidible**. Turing convirtió el problema de decidibilidad en otro equivalente, pero ambos demostraron que una función lambda, para cualesquiera dos funciones lambda, no son equivalentes.

Adicionalmente, demostraron que el modelo de Turing y el de Church son equivalentes, y por lo tanto toda función Turing-computable se puede expresar en el cálculo lambda, y viceversa. De ambos modelos surgieron lenguajes de programación, de hecho, el propio cálculo lambda es un lenguaje de programación.

La historia de la programación funcional comienza en 1960, cuando Peter Landin[5] y Christopher Strachey[6] identificaron la principal importancia del cálculo de lambda para modelar lenguajes de programación y tomaron las operaciones de semántica propuestas por Landin (1964) y la notación semántica de Strachey (1964).

En la década del 70, Rod Burstall[7] y John Darlington realizaron trabajos en transformación con el objetivo de crear el primer lenguaje de programación funcional creando los patrones de reconocimiento de Burstall y Darlington (1977). Por esos momentos David Turner desarrolló SASL, un lenguaje de programación funcional puro de alto nivel derivado del conjunto de aplicaciones ISWIM de Landin (1966), el cual incorporó las ideas de Darlington en reconocimiento de patrones en lenguajes de programación ejecutables.

En los final de los 70, Gerry Sussman[8] y Guy Steele[9] desarrollaron el lenguaje de programación Scheme que consiste en dos principales dialectos que evolucionaron del lenguaje de programación LISP. Esto dio lugar a que publicaran investigaciones que fueron conocidas como "lambda papers" en los cuales desarrollaron sus ideas sobre el uso del cálculo lambda.

Al mismo tiempo Robin Milner[10] inventó ML como meta lenguaje[11] para demostrar el

[4] **Máquina de Turing**: es un dispositivo que manipula símbolos sobre una tira de cinta de acuerdo a una tabla de reglas. A pesar de su simplicidad, una máquina de Turing puede ser adaptada para simular la lógica de cualquier algoritmo de computador y es particularmente útil en la explicación de las funciones de un CPU dentro de un computador.

[5] **Peter Landin**: (5 de junio 1930, Sheffield – 3 de Junio 2009), fue un cientifico en ciencias de la computacion

[6] **Christopher Strachey**: (1916–1975) fue un cientifico britanico especializado en ciencias de la computación

[7] **Rod Burstall**: (nacido en 1934 en Liverpool, Inglaterra) es un cientifico británico especializado en ciencias de la computación. Es uno de los 4 fundadores del laboratorio de ciencias de la computación de la universidad de Edimburgo

[8] **Gerry Sussman:** (nacido el 8 de Febrero de 1947) es profesor de ingenieria eléctrica en MIT. Ha realizado aportes para el desarrollo de lenguajes de programación

[9] **Guy Steele:** Nacido en Missouri, graduado en Harvard y doctorado en ciencias de la computación en MIT, ha colaborado en desarrollar LISP.

[10] **Robin Milner:** (Plymouth, 13 de enero de 1934 - Cambridge, 20 de marzo de 2010). Fue un científico en ciencias de computación

[11] **Meta Lenguaje:** Meta Lenguaje o ML describe a todo lenguaje de programación diseñado para desarrollar tácticas de demostración.

teorema LCF[12] en Edinburgo. El tipo de polimorfismo de Milner para ML posee una particularidad de influenciar (Milner, 1978; Damas and Milner, 1982). Ambos esquemas and ML fueron lenguajes estrictos y, sin embargo ellos contenían mejoras imperativas. Ellos no promovieron mucho el estilo del lenguaje funcional, y en particular el uso de las funciones.
En 1978 John Backus defendió a la programación funcional como un paradigma orientado a las organizaciones. Backus, mediante el equipo de desarrollo que construyó Fortran, invento Backus Naur Form (BNF). Unos años después, Strachey en colaboración con Dana Scott definieron la notación semántica para fundamentos matemáticos que finalmente se adoptó en los futuros lenguajes, la cual se continúa utilizando hasta el día de hoy.

Desarrollo

Que es un lenguaje funcional

Programación funcional es un paradigma que puede ser aplicada en varios lenguajes. Tal como su nombre lo indica, se enfoca en la aplicación de funciones.
A diferencia de los lenguajes imperativos[13], a los cuales se corresponden aquellos lenguajes más tradicionales como C y Java, la programación funcional consiste enteramente de funciones: el programa en sí es una función que recibe en sus argumentos la entrada y devuelve el resultado en su salida.

En general la función principal se define en términos de otras funciones, que a su vez están definidas en términos de otras funciones hasta llegar al nivel más bajo, donde las funciones son primitivas del lenguaje.
Al no utilizar sentencias, no existen asignaciones, por lo que una vez que las variables asumen un valor, no cambian durante la ejecución. A su vez, una llamada a una función no tiene efectos laterales, ya que estas no pueden modificar estados, haciendo que el orden de ejecución sea irrelevante.
Dado que las expresiones pueden ser evaluadas en cualquier momento, se pueden reemplazar libremente las variables por sus valores o viceversa, haciendo que los programas sean "referencialmente transparentes", es decir que el resultado de evaluar una expresión compuesta depende únicamente del resultado de evaluar las subexpresiones que la componen; no depende del historial del programa en ejecución ni del orden de evaluación de las subexpresiones que la componen.

Principios

Se definen los siguientes principios de la programación funcional:

- *Evitar estados mutables*:
  La inmutabilidad de los valores propone dos principales ventajas, una de ellas es facilitar la programación multi-hilos (multithreaded programming), es decir, si múltiples hilos modifican un mismo valor, entonces se debe sincronizar el acceso al mismo, lo que puede generar errores. Si el valor es inmutable, ya no habría inconvenientes de sincronización o concurrencia. Otro beneficio de este principio es que al no modificarse los valores, facilita el entendimiento y el test del código.
  Si bien este primer principio propone evitar la inmutabilidad de los valores, no siempre se podrá

[12] **LFC:** Es una demostración automática de teoremas interactivo desarrollado en la Universidad de Edimburgo y la Universidad de Stanford por Robin Milner y otros
[13] En los **lenguajes imperativos** se desarrollan programas que describen una serie de acciones que utilizan incidentalmente valores, utilizando un conjunto de sentencias (tales asignaciones a variables, loops, condicionales) y expresiones, tal como en los cálculos, se obtiene el valor que se asigna a una variable en una instrucción de asignación y para la condición en la que depende un condicional.

eludir, ya que todo programa tienen entradas y salidas, sin embargo la programación funcional fomenta la utilización de la mutabilidad de los valores sólo en casos extremadamente necesarios.

- *Funciones como valores de primer clase*:
  Los valores de primera clase son aquellos objetos o valores primitivos que pueden ser pasados a métodos o bien ser asignados a una variable.
  Los lenguajes de Programación Funcional consideran las funciones como valores de primer clase, ya que pueden pasarse como parámetro en otros métodos, ser devueltas como resultado o bien asignarle como valor de una variable.

- *Lambdas y Closures:*
  Un closures se forma cuando el cuerpo de una función hace referencia a una variable libre. Se entiende como variable libre a las funciones que no están declaras con anterioridad, es decir que se declaran solamente en el momento de uso.
  Las funciones anónimas o lambdas[14] no se encuentran definidas a nivel de clase, y no poseen nombres. Son funciones que se almacenan en variables y puede ser invocada en cualquier parte donde se encuentre la referencia a las mismas.

- *Funciones de orden superior*:
  Se llaman funciones de orden superior a aquellas funciones que toman otras funciones como argumento o las devuelven como resultado.
  Las funciones de orden superior son una herramienta valiosa a la hora de diseñar comportamiento de abstracción y composición.

- *Recursión:*
  Como se comentó anteriormente, uno de los principios de la programación funcional es evitar estados mutables, es decir que las iteraciones con contadores no son permitida, por lo que la programación funcional hace uso de la recursividad.

- *Evaluación Perezosa vs.1m Evaluación Voraz:*
  Con la Evaluación Perezosa (Lazy Evaluation), los lenguajes funcionales solo calculan las expresiones cuando realmente se necesiten calcular, oponiéndose al estilo tradicional de "Evaluación Voraz" (Eager Evaluation), en donde las expresiones se calculan ni bien sus parámetros están disponibles.

- *Programación Declarativa vs. Programación Imperativa*:
  Finalmente, la programación funcional es declarativa, como las matemáticas, donde sus propiedades y relaciones se encuentran definidas. Y es en tiempo de ejecución donde se conoce cómo calcular los valores finales.

  A continuación se expone cómo resulta un fragmento de programación declarativa:

```
public double declarativeFactorial(int n)
{
    if(n == 1) return 1;
    else return n * declarativeFactorial(n-1);
}
```

  En cambio, en la programación orientada a objetos es imperativa, es decir que se definen los pasos que se deben seguir.

[14] **Funciones anónimas o lambdas** : deben el nombre a la teoría matemática del cálculo Lambda.

```
public double imperativeFactorial(int n)
{
    double result = 1;
    for (int i = 2; i<= n; i++){
       result*= i;
    }
    return result;
}
```

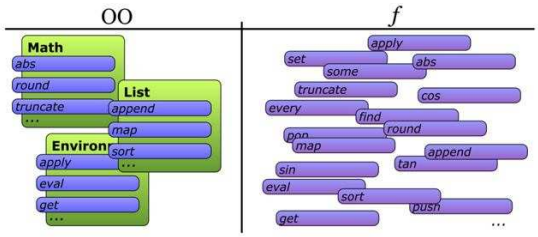
- *Sintaxis*
  Como se mencionó anteriormente, los lenguajes de programación funcional permiten desarrollo con una sintaxis minimalista, y aunque este es un factor que por lo general depende del desarrollador, los programas realizados bajo este paradigma tienen que ser entendido más fácilmente. Esto es una gran ventaja frente a los lenguajes de programación orientada a objetos, cuyos programas suelen tener una sintaxis más extensa. Además, haciendo que puedan existir estructuras en común entre programas diferentes, logra que sea más fácil para un desarrollador pasar de un proyecto a otro, o que varios desarrolladores participen de un mismo proyecto.

- *Modelado*
  El problema aparece al momento de elegir de qué forma representar la arquitectura: el Lenguaje de Modelado Unificado (UML por sus siglas en ingles), por la forma en que está pensado, puede generar una tendencia al diseño clásico de POO. Es por eso que en programación funcional no se suelen utilizar medios tan visuales para modelar los sistemas, si no que se acude a la notación matemática y lógica.

- *Manejo de memoria:*
  Muchos programas sofisticados necesitan asignar la memoria dinámicamente desde la pila. En C, esto se realiza con la función “malloc”, seguida por la porción de código para inicializar el segmento a ser asignado. El programador es responsable de retornar el segmento cuando este ya no se necesita más, lo cual puede conducir a tener errores de "dangling-pointer".
  Todo lenguaje funcional desliga al programador de esta gestión de espacio. El espacio de direcciones es asignado e inicializado implícitamente, y recuperado automáticamente por el recolector de basura. La tecnología de asignación de espacios y recolección de basura actualmente está muy desarrollada, y los costos de performance son mucho menores.

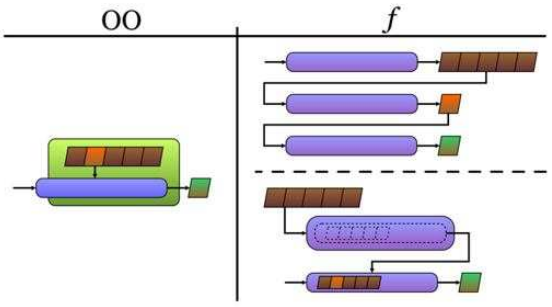
Desventajas de la Programación Funcional

-Modularización:
Los lenguajes funcionales suelen definir modularización como la capacidad de encapsular una unidad de trabajo en una función. El problema está en que en la mayoría de los casos las funciones están definidas de forma global haciendo que estén disponibles desde cualquier parte. Comparándolo con la modularización en un lenguaje orientado a objetos es fácil ver la diferencia: en estos últimos la mínima unidad de encapsulamiento es la clase. Al poder heredar de otras clases, es posible organizar las funciones en grupos, combinando clases a gusto, posibilitando de esa forma al desarrollador de que sólo utilice las porciones de librerías que necesitan.



-Abstracción:
La abstracción es la posibilidad de brindar interfaces claras, simples y de alto nivel, ocultando el funcionamiento interno de la

implementación. En la programación funcional esto se logra o bien mediante el uso de varias funciones que manejen bloques de datos desconocidos, que son manejados por el usuario o también mediante el uso de "Closures"[15] para capturar los datos en una función que luego serán pasados a otra función como una unidad.
En los lenguajes orientados a objetos es posible abstraer los datos junto con las funciones, permitiendo estructuras mucho más claras, y que los datos queden agrupados junto con las funciones que los procesan. Con este nivel de encapsulamiento es mucho más fácil reemplazar el comportamiento interno de las clases por otro que cumpla con el mismo objetivo pero de forma diferente, sin que los usuarios que utilizan la clase se enteren.



Ejemplos de lenguajes de programación funcional: Haskell

Anteriormente estuvimos hablando de Haskell, el padre de los lenguajes de la programación funcional.
Haskell es un lenguaje funcional puro, standard, moderno. Sus principales características son la evaluación perezosa (lazy) y las funciones de alto nivel que posee. También tiene un sistema innovador que soporta una forma sistemática de sobrecarga de métodos.
Este lenguaje tiene una sintaxis expresiva y una gran variabilidad de tipos de datos, incluyendo enteros (integer) y racionales, puntos flotantes, además de tipos de booleanos.
Existen varios compiladores e intérpretes disponibles y todos de ellos son gratuitos.

Pureza
Algunos lenguajes de programación permiten utilizar expresiones para producir acciones además de retornar valores. Estas acciones son llamadas "side-effects" para remarcar que el valor retornado es la salida más importante de una función. Los lenguajes que prohíben "side-effects" son llamados puros. Haskell es un lenguaje puro, lo que significa que el resultado de cualquier llamada a una función es completamente determinada por sus argumentos. Las pseudo-funciones como *rand()* en C, que retornan diferentes resultados en cada llamada, son imposibles de escribir en Haskell. Por sobre todo, las funciones de Haskell no pueden tener "side-effects", lo que significa que no pueden afectar ningún cambio al "mundo real", como cambiar archivos, escribir en la pantalla, imprimir, enviar datos a través de una red, y otras cosas más. Estas dos restricciones juntas significan que cualquier llamada a una función puede ser reemplazada por el resultado de una llamada previa con los mismos parámetros, y el lenguaje garantiza que todos estos ordenamientos no van a cambiar el resultado del programa.
Una ventaja de esto se da en la compilación. En lenguajes como C el optimizador y compilador intenta adivinar que funciones no tienen "side-effects" y no dependen de variables mutables globales. Si este proceso es incorrecto, la optimización puede cambiar la semántica del programa, y por lo tanto para evitar cualquier inconveniente los optimizadores conservan

[15] Se llama "Closure" o "Clausura" a una función que es evaluada en un entorno conteniendo una o más variables dependientes de otro entorno. Cuando es llamada, la función puede acceder a estas variables. En algunos lenguajes, una clausura puede aparecer cuando una función está definida dentro de otra función, y la función más interna refiere a las variables locales de la función externa. En tiempo de ejecución, cuando la función externa se ejecuta, se forma una clausura, consistiendo en el código de la función interna y referencias a todas las variables de la función externa que son requeridas por la clausura.

una gran parte del código o requieren que el programador indique que expresiones son puras para ser procesadas de manera independiente. Si comparamos este proceso con un compilador de Haskell, resulta que este es un conjunto de transformaciones matemáticas puras. El resultado es un nivel de optimización más elevado, con computación matemática pura que puede ser más fácil de dividir en varios hilos los cuales pueden ser ejecutados en paralelo cosa que es muy importante actualmente en CPUS de más de un núcleo. Adicionalmente, la computación pura es menos propensa a errores y fácil de verificar, lo que hace al lenguaje Haskell mucho más robusto y veloz en el desarrollo de programas.

Evaluación perezosa (lazy evolution)
Debido a que la computación pura es transparente para referenciar puede ser mejorada en cualquier momento y continuar devolviendo el mismo resultado. Esto hace posible posponer el cálculo de valores hasta que se necesiten, es decir, para calcular perezosamente. La evaluación diferida evita cálculos innecesarios y permite, por ejemplo, definir y utilizar estructuras de datos infinitas.

Recursividad
La recursión es muy utilizada en la programación funcional y muchas veces es la única manera de iterar. El proceso de compilación en Haskell optimiza fuertemente los fragmentos de código recursivos para optimizar el consumo de memoria.

Quicksort en Haskell
Una de las primeras características de la sintaxis de Haskell es el uso de paréntesis para agrupar, y no aplicarlos en funciones. La utilización de una función f con un parámetro “x” se codifica f x y no f(x). También puede escribirse en forma separada como (f x)

```
quicksort :: Ord a => [a] -> [a]
quicksort []     = []
quicksort (p:xs) = (quicksort lesser) ++ [p] ++ (quicksort greater)
    where
        lesser  = filter (< p) xs
        greater = filter (>= p) xs
```

Los paréntesis indican que debe hacer una agrupación en el lado derecho de la ecuación.
Los símbolos [] representan el array vacío, [p] para un array unitario que contiene un elemento p, ++ es un operador que concatena el array, y las dos llamadas a los operadores “filter” obtienen los elementos de “xs”
En Haskell se pueden definir funciones utilizando lo que se denomina “pattern-matching”. En este ejemplo, la primera con el patrón [] para un array vacío y el segundo con el patrón (p:xs) manteniendo en un array no vacío el elemento “p” en la cabecera (usado como un elemento pivot) y la cola xs.
La primera línea de arriba es la firma del tipo de la función indica que quicksort transforma una lista de elementos de algún tipo a (generalmente por "alfa") en una lista del mismo tipo, para un tipo de que es una instancia de la clase de tipos Ord (lo que significa que las operaciones de comparación se definen para eso, por lo que los elementos de tipo A puede ser comparado con uno del otro).
A continuación mostramos como sería el algoritmo en C para mostrar su complejidad asociada:

```
// To sort array a[] of size n: qsort(a,0,n-1)

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);

    a[hi] = a[l];
    a[l] = p;

    qsort( a, lo, l-1 );
    qsort( a, l+1, hi );
  }
}
```

En el ejemplo anterior puede verse que los programas funcionales tienden a ser mucho más concisos, generalmente por un factor de dos a diez, que sus contrapartes imperativas. El quicksort anterior puede ser escrito incluso escrito de forma más concisa con la ayuda de listas por comprensión:

```
qsort (p:xs) = qsort [x | x<-xs, x<p] ++ [p] ++ qsort [x | x<-xs, x>=p]
```

No hay volcados de memoria

Muchos lenguajes funciones, y Haskell en particular, son fuertemente tipados, eliminando una gran cantidad de errores de compilación en tiempo de ejecución. En particular el tipado es tan fuerte lo cual significa que no habrá volcados de memoria. Simplemente no es posible tratar un integer con un puntero, y por lo tanto tener un puntero nulo.

Reutilización de código

Por supuesto, el tipado está disponible en muchos lenguajes imperativos, como Ada o Pascal. Sin embargo, el sistema de tipos de Haskell es mucho menos restrictivo que, por ejemplo, de Pascal, ya que utiliza polimorfismo.

Por ejemplo, el programa quicksort que mostramos anteriormente no sólo ordena listas de números enteros, sino también las listas de números de punto flotante, listas de caracteres, listas de listas y, de hecho, se ordenará la lista de cualquier cosa por la cual tiene sentido tener operadores "menor que "y" mayor que “. En contraste, la versión C sólo puede ordenar un arreglo de enteros, y nada más. El polimorfismo mejora la reutilización.

Potente Abstracción

En general, los lenguajes funcionales ofrecen poderosas maneras de encapsular la abstracción. Una abstracción permite definir un objeto cuyo comportamiento interno es oculto. La abstracción es la clave para desarrollar en forma modular programas mantenibles. Un buen mecanismo de abstracción que proveen los lenguajes funcionales son las funciones de alto nivel. En Haskell, una función puede ser pasada a otra función , ser retornada como resultado de una función , guardada en una estructura de datos y muchas cosas más. Esta característica es denominada función de alto nivel y mejora sustancialmente la estructura y la forma de modular muchos programas.

Caso de Éxito

Los lenguajes de programación funcional han sido utilizados en muchas aplicaciones. Un notable caso de éxito fue realizado por la compañía **FACEBOOK** para desarrollar su chat, el cual está integrado con la interfaz web del sitio y tiene una tasa de uso superior al billón de mensajes por usuario por día, lo cual implica un pico de tráfico superior a 1GB. El proyecto inició en Enero del 2007, con un prototipo en el evento Hackathon, y poco más de un año después, en Abril del 2008, el servicio fue implementado. Los lenguajes utilizados fueron Earlang (otro lenguaje de PF) como principal, Jabber[16] para el backend del chat y su interface, y Haskell para un parser de

[16] Servicio de mensajería instantánea basado en el protocolo XMPP, un protocolo basado en XML de código y estándar abiertos.

código PHP (principal lenguaje en el que está desarrollado Facebook).

Las razones que llevaron a los ingenieros de Facebook a elegir Earlang fueron:

- Facilidad para manejar paralelismo en procesadores multinúcleo
- Los errores graves, como segfaults, no matan el proceso del Sistema Operativo
- En el logueo de errores se tenían todos los parámetros de las llamadas a las funciones, con lo cual el rastreo apuntaba al bug casi directamente
- "Hot Code Swapping", lo cual significa implementar mejoras en la aplicación en tiempo de ejecución, manteniendo de esa forma todos los estados actuales y que los usuarios no vean afectado su funcionamiento. Esto es una ventaja en ambos sentidos, ya que no solo es fácil implementar código nuevo, sino que también es igual de fácil hacer un rollback al código anterior en caso de algún error.
- Simplifica la modelación de interacciones concurrentes, a través del Modelo Actor

El Modelo Actor se trata de un modelo matemático para programación concurrente que trata a las primitivas universales como “actores”. Estos actores reciben mensajes a partir de los cuales pueden tomar decisiones locales, crear más actores o responder al mensaje. Al igual que en el modelo de Objetos se considera a todo un Actor, pero la diferencia está en que las acciones no se realizan de modo secuencial sino que a través de comunicaciones asincrónicas. El actor recibirá los mensajes como si se tratase de una casilla de correo, y luego implementara una lógica para leerlos y decidir qué hacer de acorde a todos los mensajes recibidos.

Según los comentarios de los Christopher Piro y Eugene Letuchy, los Ingenieros de Facebook a cargo del proyecto, de entre los problemas más importantes que se encontraron solo uno estuvo relacionado con una de las características inherentes de la programación funcional: la tipificación polimórfica. El problema en cuestión fue causado por la falta de checkeo de tipos, por lo que una variable asumía un tipo que no le correspondía, cosa que no podría haber sucedido en un lenguaje de programación tradicional como C# o PHP.
La utilización de un lenguaje de programación funcional también acarreo otro tipo de problemas: primero y principal, la falta de recursos educacionales, una comunidad limitada al momento de desarrollarlo (principios del 2007) y la falta de personal capacitado por el poco interés académico y profesional en el tema.
Por el otro lado, los problemas institucionales dentro de la misma empresa, en donde el equipo encargado de estos desarrollos fue quedando aislado del resto de los equipos, sumado a la imposibilidad de reutilizar herramientas desarrolladas en el pasado pensadas para PHP y C++.
A continuación se listan otros desarrollos que se han realizado en lenguajes funcionales:

- Amoco realizó un experimento en el que se re-codifica en Miranda, un lenguaje funcional lazzy, una fracción sustancial de código de su principal sistema de simulación de repositorios de aceite lo cual es una aplicación crítica. El programa resulto ser mucho más corto, y su producción reveló un número de errores en el software existente. Amoco posteriormente transcribió el programa funcional en C++ con resultados alentadores.

- Un Investigador de la corporación MITRE está utilizando Haskell para hacer prototipos de sus aplicaciones de procesamiento digital de señales
  - Investigadores de la Universidad de Durham utilizaron Miranda, y más tarde Haskell, en un proyecto de siete

años para construir LOLITA, un programa 30000 líneas para la comprensión de lenguaje natural[17].

- Query es un lenguaje query del sistema de base de datos relacional orientado a objetos O2. O2Query es el lenguaje más sofisticado en este campo que se encuentra disponible comercialmente y es un lenguaje funcional.

- ICAD Inc comercializa un software de renderizado CAD para ingenieros aeronáuticos y mecánicos. El lenguaje en el cual los ingenieros realizan el diseño es funcional, y utiliza evaluación lazy para evitar re computarizar partes del diseño que no son visibles en primer plano de la pantalla. Esto produjo resultados sustancialmente performantes.

- El compilador Haskell está escrito en Haskell, el cual tiene 100000 líneas de código.

- Pugs, la implementación Perl6 es escrita en Haskell

Conclusión

Hemos presentado en el este documento el paradigma de programación funcional y sus características principales.
Si bien los lenguajes funcionales han estado en desarrollo durante varios años, no es hasta recientemente que se ha empezado a utilizar con más frecuencia como herramientas que facilite a los programadores enfrentar la creciente complejidad del desarrollo de software. En la Argentina, el crecimiento de este paradigma en el ámbito laboral es lento, sin embargo cada vez más empresas la están adoptando. Sin embargo, en el ámbito académico, aún no ha sido admitida en sus programas y por lo tanto existe poco personal calificado.

En nuestra experiencia como desarrolladores, hemos notado que con la aplicación de este paradigma se reducen notablemente las cantidades de errores debido a dos características principales: la inexistencia de variables y a la sintaxis minimalista. Esto induce al programador a codificar de manera modular, obteniéndose un desarrollo más claro y compacto, lo cual hace más eficiente el mantenimiento de la solución.

En el presente documento hemos analizado ciertas ventajas y desventajas de la programación funcional, y vemos un gran potencial estos lenguajes. La característica que hace potente a un lenguaje son las librerías, que permiten extender las funcionalidades que el mismo posee. Si bien existen lenguajes como Haskell que tienen varios años, creemos que aún todavía falta mucho desarrollo de librerías, y por lo tanto existen limitaciones frente a otros lenguajes de programación, como por ejemplo interfaz gráfica.

A pesar de que muchos de los más acérrimos defensores e impulsores de la Programación Funcional lo afirmen, creemos que este paradigma no viene a reemplazar a la Programación Orientada a Objetos, sino que viene a complementarlo. Tal como reza el dicho “la herramienta indicada para el trabajo indicado”.

Referencias

[1] Allen, E. (2010, Abril 21). Comparison of Object-oriented and Functional. Retrieved Octubre 2012, from http://www.cs.rpi.edu/research/pdf/10-04.pdf

[2] Gibbons, J. (2010, Setiembre 3). Patterns in Functional Programming. Consultado Octubre 2012, from http://patternsinfp.wordpress.com/2010/09/03/functional-programming/

[3] Haskell Wiki. (n.d.). Consultado Octubre 2012, from http://www.haskell.org/haskellwiki/Haskell

[4] How does functional programming affect the structure of your code? (2008, Setiembre 22).

[17] Lenguaje natural: lenguaje humano utilizado en la comunicación ya sea por señas, hablada o escrita. Es propia del intelecto humano.

Consultado Octubre 2012, from http://lorgonblog.wordpress.com/2008/09/22/how-does-functional-programming-affect-the-structure-of-your-code/

[5] Hughes, J. (1984). Why Functional Programming Matters. Consultado Octubre 2012, from http://www.cse.chalmers.se/~rjmh/Papers/whyfp.pdf

[6] Narayanan, A., & Hansen, P. A. (n.d.). Does Functional Programming Really Matter? Consultado Octubre 2012, from http://www.cs.utexas.edu/~arvindn/hughes/

[7] Piro, C., & Letuchy, E. (2009, Setiembre 4). Functional Programming at Facebook. Consultado Setiembre 2012, from http://vimeo.com/6699769

[8] Piro, C., & Letuchy, E. (2009, Setiembre 4). Functional Programming at Facebook. Consultado Octubre 2012, from http://cufp.galois.com/2009/slides/PiroLetuchy.pdf

[9] Rivadera, G. R. (2008). La Programación Funcional: Un Poderoso Paradigma. Consultado Octubre 2012, from http://www.ucasal.net/templates/unid-academicas/ingenieria/apps/3-p63-Rivadera.pdf

[10] University Of Oxford. (n.d.). Jeremy Gibbons. Consultado from http://www.cs.ox.ac.uk/jeremy.gibbons/

[11] University of St Andrews. (n.d.). Haskell Brooks Curry. Consultado from http://www-history.mcs.st-andrews.ac.uk/Biographies/Curry.html

[12] Vermeersch, R. (2009, Enero 12). Concurrency in Erlang & Scala: The Actor Model. Consultado Octubre 2012, from http://ruben.savanne.be/articles/concurrency-in-erlang-scala

[13] Wampler, D. (2011). Functional Programming for Java Developers. O'Reilly Media.