ISSN 1870-4069

Análisis de las propiedades de corte aplicables sobre objetos funcionales

Jesús Juárez-De Felipe, Ulises Juárez-Martínez, María Antonieta Abud-Figueroa, José Luis Sánchez-Cervantes, Lizbeth Rodríguez-Mazahua

Instituto Tecnológico de Orizaba, División de Estudios de Posgrado e Investigación, Orizaba, Veracruz, México

jjuarezdefelipe@acm.org, ujuarez@ito-depi.edu.mx, mabud@ito-depi.edu.mx, jsanchezc@ito.depi.edu.mx, lrodriguez@itorizaba.edu.mx

Resumen. En lenguajes de programación como Scala y Java el modelo de programación orientado a objetos se ve fortalecido con el paradigma de programación funcional, así surge un nuevo paradigma basado en los objetos funcionales en donde cada función es un objeto de primera clase. Por lo que aparece la interrogante acerca de cómo programar aspectos en este tipo de paradigma, así como también de cómo aplicar patrones de diseño orientados a aspectos y cuáles son las ventajas y desventajas que se obtienen del empleo de la orientación a aspectos con objetos funcionales. La aportacion principal de este trabajo es describir cómo aplicar la programación funcional en los lenguajes Java 8 y Scala, también cómo aplicar la programación orientada a aspectos sobre los objetos funcionales y mediante algunos ejemplos se muestra hasta donde es posible usar el lenguaje AspectJ sobre los objetos funcionales.

Palabras clave: AspectJ, Java, Scala, objetos funcionales, programación orientada a aspectos.

Analysis of Cross-cutting Properties Applicable on Functional Objects

Abstract. In programming languages such as Scala and Java the object-oriented programming model is strengthened with the functional programming paradigm, so a new paradigm emerges based on the functional objects where each function is a first class object. So the question arises about how to program aspects in this type of paradigm, as well as how to apply aspect oriented design patterns and what are the advantages and disadvantages which are obtained from the use aspects orientation with functional objects. The main contribution of this work is to describe how to apply functional programming in Java 8 and Scala languages, how to apply the aspect oriented programming within functional objects and by some examples it is show to where it is possible to use the AspectJ language within functional objects.

Keywords: AspectJ, aspect oriented programming, functional objects, Java.

1. Introducción

El paradigma de programación objeto funcional surge de la combinación del paradigma de programación orientado a objetos junto con el paradigma de programación funcional, por lo que en este nuevo paradigma cada función se considera un objeto. Con la aparición de los objetos funcionales surge la interrogante sobre cómo encapsular adecuadamente los requerimientos no funcionales. Propiedades funcionales como las funciones de orden superior permiten igualar algunos conceptos de la programación orientada a objetos, por lo que se hace necesario revisar cómo se programan aspectos en este tipo de lenguajes híbridos, cómo se aplican los patrones de diseño [1, 2] orientados a aspectos (por ejemplo el patrón Objeto Trabajador [3] que convierte aplicaciones secuenciales en aplicaciones concurrentes) y cómo se obtienen ventajas (o desventajas) especialmente de un lenguaje como AspectJ que fue originalmente diseñado para Java.

Actualmente existen trabajos en los cuales se estudia la orientación a aspectos junto con lenguajes que se basan en el paradigma objeto funcional como lo son Scala y Java, por ejemplo en [4] se planteó la creación de ScalaPipe, que es un generador de aplicaciones de *streaming* para plataformas heterogéneas, mediante el uso de una colección de lenguajes de dominio específico incrustados en el lenguaje de programación Scala. En [5] se describió un marco de trabajo de programación orientada a aspectos totalmente funcional en el lenguaje Scala. En [6] se presentó el desarrollo de una biblioteca orientada a aspectos codificada en AspectJ, que pretende imitar el estándar OpenMP para la programación multi-núcleo en Java. Sin embargo, aún es necesario integrar adecuadamente todo este soporte para desarrollar soluciones eficientes para la industria.

La contribución principal de este trabajo consiste en mostrar cómo aplicar la programación orientada a aspectos usando el lenguaje AspectJ sobre los objetos funcionales en los lenguajes Java 8 y Scala, usando la sintaxis propia de AspectJ para Java y el sistema de anotaciones de AspectJ para Scala. Este artículo está organizado de la siguiente forma: en la Sección 2 se describe la aplicación del enfoque funcional desde un enfoque general. La Sección 3 muestra los objetos funcionales en el lenguaje Java 8 y cómo aplicar la programación orientada a aspectos. La Sección 4 muestra la implementación de la programación orientada a aspectos a los objetos funcionales en el lenguaje Scala. La Sección 5 muestra los resultados obtenidos. La Sección 6 da pie a la discusión de ideas. Finalmente en la Sección 7 se presentan las conclusiones y el trabajo a futuro.

2. Aplicación del paradigma funcional

El paradigma de programación funcional presenta varias ventajas [7], la más importante de ellas es que las funciones no presentan efecto de borde, esto quiere decir que no se modifica el estado interno de la función, no tiene variables globales o estáticas que sean modificadas y no presentan datos en la pantalla o realizan operaciones sobre archivos. Así al ser llamadas las funciones aún de manera concurrente estas no guardan ningún estado y los resultados que devuelvan las funciones serán siempre los mismos.

Esta característica de las funciones hace que el proceso de depuración de los programas sea mucho más rápido, ya que los programas que trabajan mediante funciones son la mayoría de las veces más confiables, además de que se presenta una mayor facilidad para la ejecución de los programas de manera concurrente y distribuida. Como desventaja del paradigma funcional se observó que en ocasiones es necesario que los métodos guarden alguna variable, por lo que en estos casos no conviene el uso de funciones, ya que se perderían las ventajas que el paradigma funcional aporta al aparecer el efecto de borde. Otra desventaja en el caso del lenguaje Java con el uso de objetos funcionales es la necesidad de utilizar la interfaz funcional para la declaración de los métodos y no declarar el comportamiento de manera directa.

Actualmente dos de los lenguajes orientados a objetos que han implementado el paradigma de programación funcional son Java y Scala. Comparando estos lenguajes de programación se observó que ambos cuentan con mecanismos para el control de concurrencia como son hilos, futuros y actores usando el *toolkit* Akka [13], aunque los últimos dos puntos Scala también los maneja de manera nativa. Además de que ambos lenguajes cuentan también con el manejo de funciones anónimas, funciones de orden superior y funciones de primera clase.

La ventaja que presenta Scala es que cuenta con un soporte para los mecanismos concurrentes antes mencionados más maduro, además de contar con soporte nativo para elementos importantes como los actores. Java tiene la ventaja de que AspectJ fue diseñado para él, por lo que su soporte está completo y se tienen todas las ventajas tanto del corte estático como del corte dinámico. Todas estas ventajas no se obtienen al trabajar con AspectJ mediante anotaciones.

3. Java 8

En Java los objetos funcionales se implementaron a partir de la versión 8 del lenguaje mediante expresiones lambda, y una forma de utilizarlas es mediante una interfaz funcional que como requisito sólo debe contener un único método abstracto. En el ejemplo que se muestra a continuación (Código 1) se emplea una lambda para realizar la operación de suma e imprimir el resultado, en la línea número 1 se observa una interfaz funcional llamada suma, que posee sólo un único método (línea 2), y en el método *main* de la clase se crea el objeto funcional, se le implementa su comportamiento (líneas 6 a 11) y por último se hace uso de él en la línea 12.

Para aplicar la programación orientada a aspectos sobre el lenguaje Java se utilizó el lenguaje AspectJ y para la compilación y ejecución de los códigos se utilizó el *plugin* de AspectJ para Eclipse [8].

```
1  interface Suma{
2      public int m(int x, int y);
3  }
4  public class Calculadora {
5      public static void main(String[] args) {
6          Suma a = new Suma() {
7              public int m(int x, int y) {
```

```
8               return x + y;
9               }
11        };
12        System.out.println(a.m(20, 10));
13    }
14  }
```

Código 1. Creación y uso de un objeto funcional en Java.

Para ver todos los puntos de unión con los que cada clase cuenta, Java proporciona la herramienta *javap*, que permite ver todo el *bytecode* que contiene cada archivo con extensión *.class* y así saber todos los posibles puntos de unión en dónde aplicar cortes para colocar avisos. Otra manera de ver todos los puntos de unión es directamente aplicando un aspecto sobre la clase interesada y usar la primitiva de AspectJ *whitin* [9] sobre toda la clase e imprimir todos los puntos de unión que encuentre. Los puntos de unión que AspectJ es capaz de detectar sobre el ejemplo anterior (Código 1) para la clase calculadora y que pertenecen al objeto funcional son:

1. initialization(paq.Suma())
2. call(int paq.Suma.sumar(int, int))
3. execution(int paq.Main.1.sumar(int, int))

En el ejemplo del codigo 2 se muestra cómo realizar cortes a todos los puntos de unión arriba mencionados.

El corte 1 se aplicó sobre el constructor de la interfaz suma (código 1 línea 6) y sólo muestra un mensaje, el corte 2 se realizó sobre la llamada al método sumar de la interfaz suma y permite capturar los valores que se le mandan al método y modificarlos. Para el corte 3 se tuvo que usar el comodín * dado que a la clase se le agregó por defecto un número 1, pero en la sintaxis de AspectJ para las firmas de los métodos los números no son válidos, este aviso captura el valor de retorno de la función. Analizando el bytecode con la herramienta javap se observó que el comportamiento del objeto funcional queda dentro de una clase interna en la clase calculadora (Código 1), pero no es posible acceder a ella mediante un aspecto, por lo que AspectJ no tiene el soporte adecuado para este tipo de construcciones.

```
1   public aspect CalculadoraAspect {
2       pointcut corte1():
3           initialization(paq.Suma.new(..));
4       pointcut corte2(int x, int y):
5           call(* paq.Suma.sumar(int, int)) && args(x, y);
6       pointcut corte3():
7           execution(int paq.Main.*.sumar(int, int));
8       before () :corte1(){
9           System.out.println("Suma.new");
10        }
11      int around(int x, int y): corte2(x, y) {
```

```
12            return proceed(x+1, y+2);
13        }
14          after() returning(Object r) :corte3(){
15            System.out.println("Retorno: "+r.toString());
16          }
17    }
```

Código 2. Aplicación de cortes al objeto funcional del Código 1.

De igual manera para la interfaz suma AspectJ no es capaz de detectar ningún punto de unión, aunque esta clase no es de gran importancia, ya que sólo contiene un método abstracto. Hay dos maneras de implementar las lambdas de forma más reducida. Con un estilo funcional:

```
Suma b = (x, y) -> x + y;
```

Y con un estilo más orientado a objetos:

```
Sum c= (x,y) -> {return x+y};
```

Los puntos de unión que AspectJ detecta para esta función son los siguientes:

1. call(int paq.Suma.sumar(int, int)),
2. execution(int paq.Main.lambda$0(int, int)).

El primer punto de unión es similar al primer ejemplo manejado (Código 1), pero el segundo es diferente, hace referencia a la ejecución de un método en una clase interna, está contiene el comportamiento de la función pero AspectJ no es capaz de detectarla.

Un problema que se presenta al trabajar expresiones lambda es cuando se vuelve a implementar el comportamiento de la misma interfaz funcional.

```
Suma a = (x, y) -> x + y;
Suma b = (x, y) -> x * y;
```

Los puntos de unión de la clase en donde se encuentran los dos objetos de la interfaz suma son:

1. call(int paq.Suma.sumar(int, int))
2. execution(int paq.Main.lambda$0(int, int))
3. call(int paq.Suma.sumar(int, int))
4. execution(int paq.Main.lambda$1(int, int))

Por lo que una manera de solucionarlo es validar la lambda que aparece primero en el código y esa es la numero cero, la siguiente es la numero 1 y así sucesivamente, para después verificar en el aspecto que el número de lambda sea el que se quiere cortar.

En el corte 5 se atrapa al objeto con la primitiva *target* para ver su clase, convertirla en cadena y validar que sea la lambda que se busca, en el corte 6 se valida directamente en el corte al metodo de la lambda en la cual se tiene interés.

```
1  public aspect MainAspect {
2      pointcut corte5(Object a):
3          call(int paq.Suma.sumar(int, int)) && target(a);
4      before(Object a): corte5(a) {
5          if(a.getClass().toString().contains("Lambda$1"))
6              System.out.println("Lambda 0");;
7          }
8        }
9      pointcut corte6():
10         execution(int paq.Main.lambda*$0(int, int));
11     before(): corte6() {
12         System.out.println("lambda 0 ");
13       }
14   }
```

Código 3. Aplicación de cortes para identificar una expresión lambda en particular.

4. Scala

Los objetos funcionales en Scala se trabajan mediante diferentes formas, ya que en Scala todo es un objeto [10] sólo se debe de cumplir el requisito de que ese objeto no presente efecto de borde. Scala cuenta con los *traits Function1* al *trait Function22* para definir funciones desde un argumento hasta veintidós, todas las funciones heredan de estos *traits* ya sea que se utilice la palabra reservada *extends* seguido del *trait* que corresponda a su número de párametros junto con sus valores de entrada y de retorno, o si no se indica la herencia de manera explícita a algún *trait*, Scala lo hace de manera autómatica.

El siguiente ejemplo muestra la aplicación de los objetos funcionales, en donde se emplea un objeto *singleton* que será el objeto funcional con su método *apply*, que es el método que de manera predeterminada utilizará la función para implementar su comportamiento.

```
1  object Test{
2      def main(args: Array[String]){
3          println(Suma(20, 30));
4      }
5  }
6 object Suma extends Function2[Int, Int, Int] { 7 def
apply(x: Int, y:Int): Int = x + y
8  }
```

Código 4. Objeto funcional en el lenguaje Scala.

Para la ejecución de los ejemplos se utilizó para la compilación y ejecución la herramienta de construcción SBT [11]. Usando las anotaciones de AspectJ [12] se

aplica la programación orientada a aspectos a programas escritos en el lenguaje Scala.

Al ser Scala un lenguaje que es interpretado por la máquina virtual de Java, todos los archivos con extensión *.scala* al compilarse dan como resultado uno o más archivos con extensión *.class* por lo que también se obtienen los puntos de unión que las clases contienen con los mismos procedimientos que se mostraron en la sección anterior Java 8. Al compilar el Código 4 se crean cuatro archivos con extensión *.class*, las clases que llevan en su nombre el símbolo "$" son las que toma la máquina virtual de Java para ejecutarlas y a estos archivos son a los que se deben de aplicar los aspectos.

Dado que el lenguaje Scala está hecho para reducir el código que los programadores escriben, siempre hay código que no se muestra a simple vista, por eso para aplicar los cortes es necesario revisar el *bytecode* de Java y ver a cuál instrucción corresponde cada uno de los puntos de unión que muestra AspectJ para conocer los nombres, escribir correctamente las firmas y realizar los cortes. Los puntos de unión que AspectJ es capaz de detectar para la función suma del Código 4 y su correspondiente nombre en *bytecode* son mostrados en la Tabla 1.

Tabla 1. Puntos de unión que AspectJ.

Punto de unión	Nombre en el bytecode
get(Suma. paq.Suma..MODULE$)	paq/Suma$.MODULE$:Lpaq/Suma$
call(int paq.Suma..apply$mcIII$sp(int, int))	paq.Suma$.apply$mcIII$sp:(II)I

Ahora una vez que se conocen los puntos de unión existentes en el código y cuál es su nombre completo en *bytecode* es posible generar las firmas colocando los correspondientes nombres de las clases y cambiando por comodines los símbolos que AspectJ no admite. Un ejemplo de un corte para los puntos de unión arriba mencionados se muestra a continuación:

```
1  @Aspect
2  class TestAspect {
3     @Before("get(paq.Suma$ paq.Suma$.MODULE*)")
4       def corte7(joinPoint: JoinPoint) = {
5           println("MODULE")
6       }
7     @Around("call(* paq.Suma$.apply$mcIII$sp(..)) && args(x, y)")
8       def corte8(joinPoint: ProceedingJoinPoint, x:Int,
        y:Int):Any = {
9         var a:Array[Object]=new Array[Object](2)
10        val z = new Integer(x+1)
11        val z2 = new Integer(y+2)
12        a(0)=z
13        a(1)=z2
```

```
14        joinPoint.proceed(a)
15
16      @AfterReturning(
17        pointcut = "call(* paq.Suma$.apply$mcIII$sp(..))",
18        returning= "result")
19          def corte9(joinPoint:JoinPoint, result:Object) {
20            println("Retorno : " + result);
21      }
22  }
```

Código 5. Aplicación de cortes al objeto funcional del Código 4.

El corte 7 es para el campo *module*, que es un objeto de tipo suma, el corte 8 sirve para cambiar el valor que recibe la función, se capturan los valores pero para mandarlos la función sólo recibe un arreglo de objetos por lo que es necesario crear el arreglo y llenarlo con los valores nuevos. El corte 9 captura el valor de retorno de la función y lo muestra. También AspectJ detecta puntos de unión en la clase *suma$* y estos están relacionados con la incialización de la clase suma, además de las llamadas y ejecución de los métodos que realizan las operaciones de la función. Otra manera de crear los objetos funcionales en el lenguaje Scala es mediante las funciones anónimas como se muestra a continuación.

```
1    object Test{
2        def main(args: Array[String]){
3  val Sum = (z:Int , y:Int) => z + y 4
println(Sum.apply(30, 40))
5            println(Sum(30, 40))
6        }
7    }
```

Código 6. Función anónima en el lenguaje Scala.

Al compilar el código se observa que se generan para la clase *test* del Código 6 tres archivos con extensión *.class*, dos de ellos tienen en su nombre el símbolo "$", uno contiene toda la implementación de la clase *test* del código y el otro archivo es una clase interna y contiene la implementación de la función anónima.

El problema que surge es con respecto a esta última clase, en donde AspectJ no es capaz de detectar ningún punto de unión, aunque al revisar el *bytecode* de esa clase se observa que contiene varios métodos y campos. Los puntos de unión que AspectJ encuentra para la clase test del Código 6 son los siguientes:

1. call(principal.Main.anonfun.1())
2. principal/Main$$anonfun$1.<init>:()V
3. call(int scala.Function2.apply$mcIII$sp(int, int))
4. scala/Function2.apply$mcIIsp:(II)I

En donde el primero es la llamada al constructor de la clase interna, este tipo de clases se generan automáticamente una por cada función anónima que se utilice. El segundo punto de unión es la llamada al método *apply*, a la firma de este método

internamente se le agregan los valores de entrada y de retorno, en este caso tres enteros (III).

5. Resultados

En las distintas pruebas que se realizaron mostradas en las secciones 3 y 4, se comprobó que AspectJ sí es capaz de realizar cortes sobre los objetos funcionales aunque con ciertas limitaciones. Existen algunas clases sobre las que AspectJ no cuenta con el soporte adecuado, por lo que aunque existan en el *bytecode* y sean ejecutadas por la máquina virtual son invisibles para los aspectos.

Estas clases que no son detectadas, son en su mayoría clases internas que se crean a tiempo de compilación y que no aparecen directamente en el código, el problema es que en varios de los casos éstas son las que contienen la implementación de los objetos funcionales, por lo que en esta situación sólo es posible aplicar cortes sobre las llamadas a los constructores y métodos de la clase externa pero no acceder a la clase interna.

En Java uno de los resultados que se obtuvo es que AspectJ no reconoce a las clases que se generan a tiempo de compilación y son las encargadas de manejar el comportamiento de las lambdas, éstas son clases generadas mediante la instrucción de *bytecode invoke dynamic*, esta instrucción sirve para la creación de clases en tiempo de compilación.

En Scala los resultados mostraron que este lenguaje al tener una sintaxis más corta para el programador, internamente genera una gran cantidad de código para crear todos los mecanismos que Scala necesita para funcionar, por lo que es necesario revisar el *bytecode* porque muchas de las líneas que en el código fuente son una, en el *bytecode* se generan varias y es necesario conocer exactamente en cuál se va a aplicar el corte, además de que internamente los nombres de algunos métodos presentan algunos cambios por lo que es necesario conocer su nombre real para crear la firma de ese método y realizar un corte.

6. Discusión

La programación orientada a aspectos presenta varias ventajas, entre ellas se encuentran el permitir una mayor reutilización del código, hacer que los programas sean más modulares y también más adaptables a los cambios, ya que no es necesario modificar el código fuente para cambiar el comportamiento de un programa, además permitir una mejor modularización del código. Todas estas ventajas que proporciona la programación orientada a aspectos se suman a las que proporcionan los objetos funcionales, por lo que es necesario un soporte adecuado de la programación orientada a aspectos en el lenguaje AspectJ para los objetos funcionales en los lenguajes Java y Scala.

7. Conclusiones

La programación con objetos funcionales tiene la ventaja de ofrecer una mejor modularización. Así como también al tener las funciones un estado inmutable pueden

ser llamadas de manera concurrente y al no tener las funciones variables adentro de ellas que pudieran alterarse cada vez que estas son llamadas, siempre devolverán los mismos resultados.

La programación orientada a aspectos con objetos funcionales no tiene todavía un gran soporte por parte del lenguaje AspectJ, pero con lo que se tiene hasta este momento es posible realizar varias tareas como interceptar los valores que se le envían a los objetos funcionales y cambiarlos o capturar los valores que las funciones retornan. Como trabajo futuro se aplicará la programación orientada a aspectos a las clases que son generadas automáticamente y que manejan el comportamiento de los objetos funcionales y que en este momento AspectJ no cuenta con el soporte adecuado, por lo tanto no es capaz de identificar puntos de unión dentro de ellas. Todo esto permitirá plantear herramientas y/o extensiones al lenguaje AspectJ para que sea posible aplicar correctamente aspectos a los objetos funcionales en los lenguajes Java 8 y Scala.

Agradecimientos. Los autores agradecen al Tecnológico Nacional de México por apoyar este trabajo. Además, este trabajo de investigación fue patrocinado por el Consejo Nacional de Ciencia y Tecnología (CONACYT), así como por la Secretaría de Educación Pública (SEP) a través de PRODEP.

Referencias

1. Hunt, J.: Scala Design Patterns: Patterns for Practical Reuse and Design. Springer Publishing Company, Incorporated (2013)
2. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns; Elements of Reusable Object-Oriented Software. Addison Wesley (1994)
3. Laddad, R.: AspectJ in Action Practical Aspect-Oriented Programming. Greenwich, CT: Manning Publications Co. (2003)
4. Wingbermuehle, J. G., Chamberlain, R. D., Cytron, R. K.: ScalaPipe: A Streaming Application Generator. Field-Programmable Custom Computing Machines (FCCM). In: IEEE 20th Annual International Symposium, pp. 244–244 (2012)
5. Spiewak, D., Zhao, T.: Method Proxy-Based AOP in Scala. Journal Of Object Technology, Vol. 8, No. 7 (2009)
6. Medeiros, B., Sobral, J. L.: Implementing an OpenMP-like standard with AspectJ. In: Proceedings of the 3rd workshop on Modularity in systems software (MISS '13), ACM, New York, NY, USA, pp. 1–6 (2013)
7. Wampler, D.: Functional Programming for Java Developers. O'Reilly (2011)
8. AspectJ Development Tools. Online, http://www.eclipse.org/ajdt (2016)
9. Miles, R.: AspectJ Cookbook. Sebastopol, CA, O'Reilly (2004)
10. Odersky, M., Spoon, L., Venners, B.: Programming in Scala. Artima (2007)
11. The interactive build tool. Online, http://www.scala-sbt.org (2016)

12. An Annotation Based Development Style. Online, https://eclipse.org/aspectj/doc/next/adk15notebook/ataspectj.html (2016)
13. Akka. Online, http://akka.io (2016)