Programación Funcional y Herencia de Clases: Tipos de Datos Algebraicos Extensibles en Scala

Agustín Jesús Ríos

Universidad Nacional de Rosario
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Rosario, República Argentina

Licenciatura en Ciencias de la Computación

Abril 2014

Director: Guido Macchi

A mi viejo... A vos arquetipo del nunca aflojar, leal y gamba...

"A language that doesn't affect the way you think about programming is not worth knowing."

Alan Perlis

Resumen

Cuando se desarrolla un software muchas veces se tiene que lidiar con tipos de datos recursivamente definidos y operaciones sobre ellos, por ejemplo en un árbol de sintaxis abstracta o en la manipulación simbólica de expresiones. Éstos son comunes tanto en la programación orientada a objetos como en la programación funcional. En el primer paradigma se representan como una jerarquía de clases relacionadas con una clase base común, siguiendo el patrón de diseño *Composite* [3], en el segundo como tipos de datos algebraicos.
El desafío en estas estructuras, que se encuadra dentro del *expression problem*, es cómo extender el tipo de datos y las operaciones sobre él sin modificar el código existente y reutilizando las operaciones previamente definidas.
En este trabajo se presentan los tipos de datos algebraicos extensibles en Scala. Se mostrará cómo se pueden codificar en el lenguaje y cómo extenderlos sin modificar el código existente y reutilizándolo para definir las extensiones de las operaciones. También se presenta una implementación de los tipos de datos abiertos y las funciones abiertas, utilizando herencia de clases y tipos de datos algebraicos extensibles.
Scala unifica los tipos de datos algebraicos con las jerarquías de clases, haciendo posible la utilización de la herencia en un entorno funcional. Esta característica permite la extensión del tipo de datos como así también de las operaciones sobre el mismo.
Un punto importante es que se puede usar *pattern matching* para definir todas las operaciones, inclusive en las extensiones.

Índice general

Agradecimientos

Quiero agradecer muy especialmente a mis padres Rubén y Cecilia. A mi vieja por incentivarme a que siga la facultad, porque, como decía ella, eso me iba a dar las herramientas para mi futuro. A mi viejo, por alentarme a que continúe, por ayudarme en todo y por estar siempre listo para salir al cruce de cualquier problema que pudiera tener. Sin su empuje no hubiese podido terminar la carrera. Gracias por hacerme la persona que soy.
A mi familia: Luciano, Nadia, Alejo y Mónica.
A mi abuela mumu, por estar siempre pendiente.
A mi novia, que me bancó en todas, gracias Vicki por estar apoyándome siempre.
A Germán, mi gran amigo de la facu, por todas las madrugadas que compartimos estudiando en tu departamento. A Andrés, mi otro gran amigo de la carrera. Gracias por compartir el largo camino conmigo. Sin ustedes no se hubiera disfrutado tanto. Una de las grandes cosas que me llevo de la facultad es su amistad.
A mis amigos de siempre, que de una u otra manera estuvieron, en especial a Ivan, que me ahorró una pequeña fortuna en impresiones.
A Julián por los momentos compartidos al comienzo de la carrera. A Ale por transitar conmigo el último tramo de esta etapa. A Mariana, Juanjo y Majo que también estuvieron en este tiempo. Gracias a todos por los momentos vividos.
A Guido, por aceptar dirigir mi tesis, después de una lectura de un minuto del paper que le mostré.

Capítulo 1

Introducción

Todo software evoluciona con el tiempo, entonces es esencial que el software sea extensible. Un software es extensible si puede ser adaptado a cambios en la especificación del mismo. Construir software que sea fácil de extender es uno de los principios fundamentales de la Ingeniería del Software, pero el desarrollo de software extensible propone muchos desafíos de diseño e implementación, especialmente si la extensión no fue anticipada.
Por oro lado, la reutilización del código existente del sistema es esencial para reducir los costos de producción del sistema como así también el tiempo de desarrollo de nuevos requerimientos. Por lo tanto cuando se desarrolla un software se busca un diseño extensible y que permita la reutilización de código.
El *expression problem* es un caso particular del problema de extensión de un programa, refiere a la situación donde se necesita extender un tipo de datos recursivamente definido y las operaciones sobre él, con dos restricciones:

- Sin recompilar el código existente.
- Resolviendo los tipos estáticamente (es decir, sin casts).

El término fue acuñado por Philip Wadler en un post en la lista de mail de *Java Genericity* [18]. El problema entonces es cómo construir software extensible, que evolucione con el tiempo y que sea fácil de extender, lidiando con especificaciones de datos recursivos y operaciones sobre ellos.
Esta situación se presenta tanto en el paradigma funcional como en el paradigma orientado a objetos, cada uno de ellos propone un enfoque distinto de la representación, pero en ambos casos es difícil la extensión en alguna de las dos dimensiones (datos u operaciones).
Vamos a agregar una restricción más a las dos anteriores: queremos reutilizar el código existente para definir las operaciones sobre las extensiones del tipo.
Un punto a tener en cuenta cuando extendemos el tipo de datos y las operaciones, es que las extensiones pueden ser lineales, es decir, se hace una extensión del tipo que luego se vuelve a extender, o independientes, es decir, distintos grupos de programadores hacen sus propias extensiones de algunas de las dos dimensiones. El

desafío en estas últimas es combinar las extensiones para utilizar ambas. Veremos que con los tipos de datos algebraicos extensibles y algunas construcciones de Scala, la combinación es bastante directa.
Muchas soluciones del *expression problem* fueron propuestas:
En su trabajo, Togersen [17] presentó varias soluciones al problema, todas ellas basadas en clases genéricas, en el contexto de lenguajes orientados a objetos como Java y C#. Togersen propuso dos tipos de soluciones, una basadas en la descomposición orientada a objetos y otras basadas en la descomposición funcional, utilizando el patrón de diseños *Visitor* para implementar las operaciones separadas del tipo de datos. En ninguna de las soluciones propuestas es posible combinar extensiones independientes del tipo.
El patrón *Visitor Extensible* fue propuesto por Krishnamurti, Felleisen y Friedman [5], es una extensión del patrón de diseño Visitor que permite añadir nuevos tipos de datos y nuevas operaciones. Es una combinación de la descomposición funcional y las descomposición orientada a objetos, que elimina las debilidades de extensión de cada una de ellas. Este nuevo patrón está basado en *type casts*, lo que hace que no puedan resolverse los tipos estáticamente.
Los tipos de datos abiertos y funciones abiertas fueron propuestas por Andres Löh y Ralf Hinze [7]. En su trabajo proponen una extensión del compilador de Haskell para que la definición de los constructores de un tipo de datos abierto y las ecuaciones de las funciones abiertas puedan aparecer esparcidos en el código del programa. De esta manera es posible ir declarando nuevos constructores de un tipo a medida que se vayan necesitando y se pueden agregar ecuaciones a una función sobre ese tipo para que maneje el nuevo caso. En la actualidad no existe ningún compilador de Haskell que soporte la extensión propuesta, si se implementó como una extensión del compilador del lenguaje Ocaml.
Swierstra [16] presentó una solución al *expression problem* en Haskell, sin ninguna extensión al sistema de tipos o al compilador. En su trabajo se describe una técnica para definir tipos de datos y funciones sobre el tipo de manera que sean extensibles ambas entidades.
Zenger y Odersky [19] presentaron los tipos algebraicos de datos extensibles con defaults. En su trabajo utilizaron tipos algebraicos del lenguaje Pizza, un lenguaje experimental que es un superconjunto de Java, ya sin continuidad de desarrollo, para implementar operaciones extensibles con un caso por default. Mostraron que es posible implementar los tipos algebraicos extensibles en un lenguaje orientado a objetos como Java, usando un patrón de diseño basado en un Visitor extensible con casos por defaults. En su trabajo las operaciones no están definidas con un *pattern matching* natural y no pueden combinarse extensiones independientes del tipo, solo pueden combinarse extensiones lineales.
En este trabajo se presentan dos soluciones al *expression problem* basadas en los tipos algebraicos de datos extensibles con defaults, propuestos por Zenger y Odersky. A diferencia de la propuesta original, en la implementación de tipos extensibles aquí presentada se pueden combinar extensiones independientes del tipo y no se utilizará el patrón de diseño *Visitor*, sino que se utilizará el *pattern matching* provisto

por el lenguaje, consiguiendo así una codificación mas limpia y funcional.
También se mostrará una implementación de los tipos de datos abiertos y funciones abiertas, propuestas por Löh y Hinze, que no requiere la modificación del compilador de Scala.
Las soluciones presentadas se implementan en el lenguaje de programación Scala, aprovechando su combinación de paradigmas y las características de su sistema de tipos. Scala es un lenguaje funcional orientado a objetos, que crea una combinación de ambos paradigmas (unifica y generaliza conceptos de los dos estilos) [8, 11], eso hace que la solución presentada como tipos de datos algebraicos extensibles en Scala pueda utilizarse en un estilo puramente funcional.
El resto de este trabajo se organiza como sigue:

- En el Capítulo 2 se hace una introducción mas detallada al *expression problem* a partir de un ejemplo concreto, y se muestra como surge en los paradigmas de programación funcional y orientado a objetos.

- En el capítulo 3 se muestra una implementación de los tipos algebraico de datos extensibles con defaults en Scala, basada en el trabajo de Zenger y Odersky [19]. Se mostrará como extender el tipo de datos, agregando nuevos casos, y dos maneras de extender las operaciones sobre el tipo, una utilizando funciones de alto orden y otra utilizando herencia de clases. También se mostrará como puede solucionarse el *expression problem* con esta técnica.

- En el capítulo 4 se implementan los tipos de datos abiertos y las funciones abiertas, propuestas en [7], utilizando los tipos de datos algebraicos extensibles descriptos en el capítulo 3. Se verá como puede extenderse tanto el tipo de datos como las funciones sobre él.

- En el capítulo 5 se muestra como definir el operador *fold* del tipo de dato de manera que soporte las extensiones del mismo y de esta forma poder definir funciones extensibles sobre el tipo de datos con la función *fold*. El enfoque estará basado en la técnica presentada en [16].

- El Capítulo 6 presenta las conclusiones de este trabajo.

- El apéndice A hace una introducción de las principales características del lenguaje de programación Scala.

- En el apéndice B se demuestran las propiedades algebraicas pertinentes del Capítulo 5.

- En el apéndice C se describe como organizar el código mostrado en este trabajo para compilarlo y ejecutarlo.

Capítulo 2

El *Expression Problem*

Vamos a ver cómo surge el *expression problem* al representar una típica estructura de programación. Para ello vamos a recurrir a un sencillo ejemplo: representar expresiones de un lenguaje simple y las operaciones sobre él.
El lenguaje por ahora consta de números enteros y de la suma de dos enteros. Queremos implementar un evaluador para ese lenguaje. Más adelante extenderemos el lenguaje agregando nuevas construcciones y veremos cómo podemos extender el evaluador. También agregaremos una nueva operación sobre el lenguaje que nos devuelva una representación String de una expresión del mismo. Debe ser posible realizar tales extensiones sin modificar ni recompilar el código existente, ya que pudo ser escrito por otra persona y puede existir en una librería cerrada.
A fines de mostrar el problema, el evaluador va a ser muy sencillo, particularmente no va a manejar errores.
La gramática que describe nuestro pequeño lenguaje es la siguiente:

⟨expr⟩ ::= ⟨num⟩ | ⟨plus⟩
⟨plus⟩ ::= ⟨expr⟩ **'+'** ⟨expr⟩
⟨num⟩ ::= **'número entero'**

2.1. Representación Orientada a Objetos

La representación orientada a objetos clásica es la que describe el patrón de diseño *Interpreter* [3]. Se crea una jerarquía de clases relacionadas con una clase base abstracta común. Cada clase que se extiende de la clase base representa una de las construcciones de nuestro lenguaje. En la clase base se especifican las operaciones sobre la estructura. Cada subclase concreta implementa cada una de las operaciones:

```
abstract class Term{
 def eval(): Int
```

```
}

class Num(value: Int) extends Term {
 def eval(): Int = value
}

class Plus(left: Term, right: Term) extends Term {
 def eval(): Int = left.eval() + right.eval()
}
```

En esta descomposición agregar nuevas construcciones al lenguaje, es decir, extender el tipo de datos, es directo, por ejemplo si queremos agregar una expresión que represente la multiplicación de dos expresiones del lenguaje, simplemente se crea una nueva clase Mul que extiende de la clase base y se implementa la operación eval() en ella:

```
class Mul(left: Term, right: Term) extends Term {
 def eval(): Int = left.eval() * right.eval()
}
```

Para evaluar una expresión usando esta representación, simplemente se llama a eval() sobre la expresión:

```
val term = new Plus(new Plus(new Num(3), new Num(2)), new Num(2))
term.eval()
```

El problema surge cuando queremos agregar nuevas operaciones sobre el tipo, ya que tenemos que agregar un nuevo método a la clase base Term y modificar cada una de las clases que heredan de ella para implementar ese método. Por ejemplo si queremos agregar un método que devuelve una representación String de las expresiones, tendríamos que modificar la clase abstracta Term:

```
abstract class Term{
 def eval(): Int
 def termtoString(): String
}
```

y luego implementar ese método en cada subclase de Term.

2.2. Representación Funcional

En un lenguaje funcional, la descomposición de este problema se haría clásicamente declarando un tipo algebraico que represente las expresiones de nuestro lenguaje y luego definiendo operaciones sobre ese tipo algebraico, es decir, la definición del

tipo de datos y las operaciones sobre él están estrictamente separadas, por ejemplo en Haskell se declara:

```
data Term = Num Int | Plus Term Term
```

Como Scala también es un lenguaje funcional, vamos a usar su notación para codificar esto, en Scala la declaración anterior se expresa:

```
abstract class Term
case class Num(value: Int) extends Term
case class Plus(left: Term, right: Term) extends Term
```

Y el evaluador se expresa como una función que toma como parámetro un valor de tipo Term. Estas funciones se definen usando pattern matching:

```
def eval(t:Term): Int = t match {
 case Num(v) => v
 case Plus(l,r) => eval(l) + eval(r)
}
```

Para evaluar una expresión se utiliza la función eval() y se pasa la expresión como parámetro:

```
val term = Plus(Plus(Num(3), Num(2)), Num(2))
eval(term)
```

En Scala cuando se declara un tipo algebraico con **case class** no es necesario utilizar la instrucción **new**.
El inconveniente encontrado al agregar operaciones en la representación orientada a objetos no lo tenemos en esta codificación, ya que agregar nuevas funcionalidades es directo, simplemente se declara una nueva función, por ejemplo la operación que devuelve la representación String de la expresión se codifica:

```
def termtoString(t: Term): String = t match {
 case Num(v) => v.toString
 case Plus(l,r) => "("+termtoString(l)+ "+" +termtoString(r)+")"
}
```

El problema que se presenta en esta representación es en la extensión en la otra dimensión. Para agregar nuevas construcciones al lenguaje tenemos que modificar la declaración del tipo algebraico y todas las funciones definidas hasta el momento:

```
abstract class Term
```

```
case class Num(value: Int) extends Term
case class Plus(left: Term, right: Term) extends Term
case class Mul(left: Term, right: Term) extends Term
```

Esta descomposición funcional también puede codificarse en un lenguaje orientado a objetos utilizando el patrón de diseño *Visitor* [3], donde cada operación es representada por una clase Visitor separada, que contiene métodos visit() por cada tipo de datos de la estructura de clases. La idea de este patrón es codificar el pattern matching de los lenguajes funcionales usando un "double dispatch" y así separar las operaciones del tipo de datos [2]. Esta codificación sufre del mismo inconveniente expresado anteriormente, es fácil agregar nuevas operaciones, simplemente se crea un nuevo visitor, pero para agregar nuevas construcciones al lenguaje hay que modificar todos los visitor definidos para que manejen el nuevo caso:

```
abstract class Term {
  def accept(v: Visitor)
}

abstract class Visitor{
 def visitNum(term: Num)
 def visitPlus(term: Plus)
}

class Num(val value: Int) extends Term {
 def accept(v: Visitor) = v.visitNum(this)
}

class Plus(val left: Term, val right: Term) extends Term {
 def accept(v: Visitor) = v.visitPlus(this)
}

class EvalVisitor extends Visitor{
 var result: Option[Int] = None
 def visitNum(term: Num) = this.result = Some(term.value)
 def visitPlus(term: Plus) = {
  val l = new EvalVisitor
  val r = new EvalVisitor
  term.left.accept(l)
  term.right.accept(r)
  this.result=Some(l.result.get + r.result.get)
 }
}
```

Para evaluar una expresión, se crea una instancia concreta del visitor específico que codifica la operación de evaluación y se pasa al método accept() de la expresión:

```
val term = new Plus(new Plus(new Num(3), new Num(2)), new Num(2))
```

```
val eval = new EvalVisitor
term.accept(eval)
```

Si queremos agregar un nuevo caso al tipo de datos, podemos heredar una nueva clase de la clase abstracta Term:

```
class Mul(left: Term, right: Term) extends Term {
 def accept(v: Visitor) = v.visitMult(this)
}
```

Pero tenemos que modificar la clase abstracta Visitor para agregar el nuevo caso y cada una de las subclases concretas de visitor:

```
abstract class Visitor{
 def visitNum(term: Num)
 def visitPlus(term: Plus)
 def visitMult(term: Mult)
}
```

Otro problema que tiene el diseño con el patrón visitor a comparación del pattern matching es que causa una sobrecarga de código, y lo torna más complicado [1].

2.3. Solución utilizando Herencia de Clases

En la programación orientada a objetos una forma de extender la funcionalidad de una clase es utilizando herencia de clases. Con este mecanismo podemos extender una clase, agregando nuevos métodos, sin necesidad de recompilar el código existente. Vamos a extender el ejemplo de la sección 2.1, agregando el método termtoString(). Para ello vamos a hacer una pequeña modificación, la clase abstracta Term vamos a expresarla como un *Trait*. En Scala, un *Trait* es como una *interface* de Java, pero permite la implementación de métodos en su definición. Entonces el código queda:

```
trait Term{
 def eval(): Int
}

class Num(value: Int) extends Term {
 def eval(): Int = value
}

class Plus(left: Term, right: Term) extends Term {
 def eval(): Int = left.eval() + right.eval()
}
```

[1] Martin Odersky, diseñador de Scala, escribió un artículo al respecto llamado "In Defense of Pattern Matching", http://www.artima.com/weblogs/viewpost.jsp?thread=166742

```
class Mul(left: Term, right: Term) extends Term {
 def eval(): Int = left.eval() ∗ right.eval()
}
```

Y definimos un nuevo trait que extienda de Term, en el cual se define la operación que queremos incluir.

```
trait TermP extends Term {
 def termtoString() : String
}
```

Ahora declaramos nuevas clases que extienden de las anteriores y que implementan el trait TermP:

```
class PlusP(left: Term, right: Term) extends Plus(left,right) with TermP {
 def termtoString(): String = left.termtoString() + "+" + right.termtoString()
}

class NumP(value: Int) extends Num(value) with TermP {
 def termtoString(): String = value.toString()
}

class MulP(left: Term, right: Term) extends Mul(left,right) with TermP {
 def termtoString(): String = left.termtoString() + "∗" + right.termtoString()
}
```

Al parecer logramos solucionar el problema, pero lamentablemente el código anterior no compila, ya que hay un error de tipos. La llamada left.termtoString() en las clases PlusP y MulP falla, ya que el parámetro left (y right) de los constructores es de tipo Term (como lo especifica el patrón *interpreter*), y ese tipo no tiene un método termtoString.
Una solución al problema de tipos anterior es declarar los parámetros left y right con tipo TermP, como se muestra en el siguiente código:

```
class PlusP(left: TermP, right: TermP) extends Plus(left,right) with TermP {
 def termtoString(): String = left.termtoString() + "+" + right.termtoString()
}

class NumP(value: Int) extends Num(value) with TermP {
 def termtoString(): String = value.toString()
}

class MulP(left: TermP, right: TermP) extends Mul(left,right) with TermP {
 def termtoString(): String = left.termtoString + "∗" + right.termtoString()
}
```

Con esta modificación logramos que el código compile, pero nos encontramos con otro problema. La nueva signatura de los constructores de las clases heredadas, especifica que los argumentos ya no pueden ser de tipo Term, sino que tienen que ser

de un tipo mas específico, TermP. Esto implica que una instancia de tipo Term no puede ser utilizada para crear instancias de TermP, lo que dificulta la reutilización de código, ya que cualquier constante preexistente o método que devolvía un valor de tipo Term no puede ser reutilizado. Por ejemplo, si teníamos una función para calcular una parte compleja de una ecuación usando nuestro simple lenguaje y por otro lado teníamos otra función para calcular otra parte de la misma ecuación y luego queremos representar la suma de ambas, ya no podemos reutilizarlas para nuestra nueva representación con TermP, porque ambas funciones devuelven objetos de tipo Term que era el tipo base con el que trabajábamos antes de la extensión, es decir, no podemos definir:

```
val suma = PlusP(CalcularSumaIzquierda(), CalcularSumaDerecha() )
```

La forma de evitar estos problemas y reutilizar el código preexistente con la nueva representación es recurrir a *downcasting* para transformar instancias de tipo Term en instancias de tipo TermP , pero esto viola una de las restricciones impuestas en el capítulo 1.
El problema que surge al tratar de extender una estructura de clases definida recursivamente como Term utilizando herencia de clases es bien conocido en el diseño orientado a objetos, por ejemplo al utilizar el patrón *composite* en [3] explícitamente se sugiere que en ciertas circunstancias se tendrá que recurrir a la comprobación de tipos en tiempo de ejecución.

Capítulo 3

Tipos de datos Algebraicos Extensibles con Defaults

Zenger y Odersky introdujeron los tipos de datos algebraicos extensibles con defaults [19]. En su trabajo utilizan los tipos algebraicos del lenguaje *Pizza* [12], un superconjunto de Java. La extensión de las operaciones sobre el tipo se logra con subclases. También implementan esta idea en un lenguaje como Java utilizando sólo objetos y subtipado (subclases). La codificación toma la forma de un nuevo patrón de diseño para visitors extensibles con casos por default.
Vamos a llevar la idea de tipos algebraicos extensibles al lenguaje Scala y se mostrará que puede utilizarse en un diseño funcional. Se utilizará la herencia de clases para extender el tipo de dato como así también las operaciones sobre el tipo. Vamos a ver que la extensión de las operaciones también se puede lograr sin utilizar herencia.

3.1. Extendiendo tipos de datos algebraicos en Scala

Los tipos algebraicos de datos en Scala se declaran con **case class**. Las clases declaradas con el modificador **case** se denominan *case classes*, y se utilizan para definir las distintas formas del tipo algebraico. El modificador **case** tiene varios efectos sobre la clase. Por un lado, permite la construcción de instancias del tipo sin la necesidad de utilizar **new**. Por el otro, permite el *pattern matching* sobre sus constructores [2].
A pesar de los efectos del modificador **case**, las case classes no dejan de ser clases, que pueden instanciarse y extenderse. Esto último es muy útil ya que nos permite extender el tipo utilizando la herencia de clases.
Definimos el tipo algebraico básico que representa a nuestro pequeño lenguaje:

```
abstract class Term
case class Num(value: Int) extends Term
case class Plus(left: Term, right: Term) extends Term
```

La declaración anterior introduce el tipo Term y dos constructores, Num que toma como parámetro un número entero y Plus que toma como parámetros dos expresiones de tipo Term. Si queremos ampliar la definición del lenguaje, agregando una nueva expresión, podemos extender el tipo para que refleje ese cambio usando subclases de la siguiente manera:

```
abstract class ExtendedTerm1 extends Term

case class Mul(left: Term, right: Term) extends ExtendedTerm1
```

Como el tipo base de Mul sigue siendo Term, cualquier función que espere un parámetro de tipo Term, va a aceptar una construcción Mul y nos permite hacer pattern matching sobre todos los casos:

```
def imprimirCaso(t:Term): Unit = t match {
 case Num(n) => println("Es Num!")
 case Plus(_,_) => println("Es Sum!")
 case Mul(_,_) => println("Es Mul!")
}
```

Entonces el problema ahora es como extender las funciones definidas originalmente sobre el tipo algebraico base para que manejen el nuevo caso, utilizando pattern matching. La solución directa es modificar todas las funciones, pero queremos hacerlo sin recompilar el código existente.
Antes de mostrar como se extienden las operaciones, vamos a ver que hay dos tipos de extensiones que se pueden hacer sobre los tipos algebraicos.
Se llama *extensión lineal*, cuando se extiende nuevamente un tipo algebraico que ya habíamos extendido, por ejemplo:

```
abstract class ExtendedTerm2 extends ExtendedTerm1

case class Diff(left: Term, right: Term) extends ExtendedTerm2
```

En este caso, se toma la extensión del tipo original, ExtendedTerm1, y se vuelve a extender, agregando el caso Diff.
La *extensión independiente* se da cuando se hacen varias extensiones del mismo tipo algebraico independientemente una de otras:

```
abstract class ExtendedTerm1 extends Term
case class Mul(left: Term, right: Term) extends ExtendedTerm1

abstract class ExtendedTerm2 extends Term
case class Diff(left: Term, right: Term) extends ExtendedTerm2
```

En este último caso, las extensiones ExtendedTerm1 y ExtendedTerm2 son independientes, ambas extienden el tipo base Term agregando nuevos constructores.
La extensión independiente es más interesante, ya que permite hacer extensiones del mismo tipo a diferentes grupos de programadores y luego combinarlas, más adelante se mostrará como se pueden combinar esas extensiones. Los esquemas de herencia de ambas extensiones se muestran en las figuras 3.1 y 3.2.

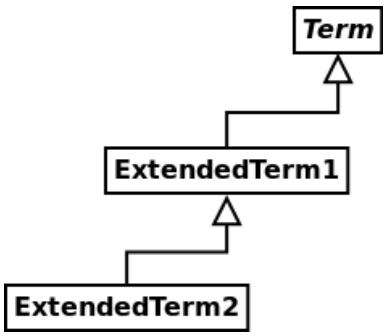

Figura 3.1: Extensión Lineal

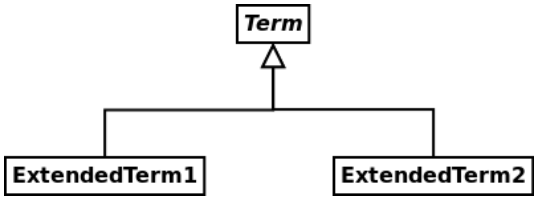

Figura 3.2: Extensión Independiente

3.2. Extendiendo las Operaciones con Funciones de Alto Orden

En esta sección vamos a mostrar como pueden extenderse las operaciones definidas para el tipo algebraico original, para que manejen los casos de las extensiones. Vamos a extenderlas usando funciones de alto orden, en la sección siguiente vamos a ver como pueden extenderse usando herencia. También mostraremos como pueden combinarse las extensiones independientes en operaciones que manejen todos los casos.
La estrategia fundamental es agregar un caso *default* al pattern matching de las operaciones originales, para que manejen las posibles futuras extensiones del tipo y parametrizar la función con otra función que se encargue de manejar ese caso por default.

3.2.1. Extensiones Lineales

Vamos a extender las operaciones cuando la extensión es lineal. Primero se declara el tipo algebraico original y las operaciones sobre él:

```
abstract class Term

case class Num(value: Int) extends Term
case class Plus(left: Term, right: Term) extends Term

def eval(f: Term => Int)(t: Term): Int = t match {
 case Num(v) => v
 case Plus(l,r) => eval(f)(l) + eval(f)(r)
 case default => f(default)
}

def termtoString(f: Term => String)(t: Term):String= t match {
 case Num(v) => v.toString
 case Plus(l,r) => "("+termtoString(f)(l)+"+"+termtoString(f)(r)+")"
 case default => f(default)
}
```

Si queremos usar el tipo Term y sus operaciones, podemos parametrizar las funciones con alguna función trivial, ya que por ahora nunca se va a entrar en el caso default, porque todos los casos del tipo están cubiertos en el pattern matching (Num y Plus). La función trivial puede ser aquella que lance una excepción o podemos cambiar la signatura de las operaciones y hacer que devuelvan un valor Option, por ejemplo en eval: **def** eval(f: Term => Option[Int])(t:Term): Option[Int], entonces se parametriza eval con la función que siempre devuelve None: x => None.
Para utilizar las funciones sobre el tipo podemos declarar:

```
val evaluador = eval(x => throw new Exception)(_)
val toString = termtoString(x => throw new Exception)(_)
```

Ahora hacemos una extensión sobre el tipo original agregando el constructor Mul y definimos las funciones que van a manejar ese caso, también incluimos un caso default en ellas para poder manejar futuras extensiones de este tipo.

```
abstract class ExtendedTerm1 extends Term
case class Mul(left: Term, right: Term) extends ExtendedTerm1

def evalExtended1(f: Term => Int)(t: Term): Int = t match{
 case Mul(l,r) => evalExtended1(f)(l) * evalExtended1(f)(r)
 case default => f(default)
}
```

```
def termtoStringExtended1(f: Term => String)(t: Term): String = t match{
 case Mul(l,r) => "("+termtoStringExtended1(f)(l)+"*"+termtoStringExtended1(f)(r)+")"
 case default => f(default)
}
```

Las funciones evalExtended1() y termtoStringExtended1() manejan sólo el caso de la extensión, los demás se manejarán con las funciones eval() y termtoString().
Hacemos un última extensión lineal, agregando un nuevo caso, Diff:

```
abstract class ExtendedTerm2 extends ExtendedTerm1
case class Diff(left: Term, right: Term) extends ExtendedTerm2

def evalExtended2(f: Term => Int)(t: Term): Int = t match{
 case Diff(l,r) => evalExtended2(f)(l) - evalExtended2(f)(r)
 case default => f(default)
}

def termtoStringExtended2(f: Term => String)(t: Term): String = t match{
 case Diff(l,r) => "("+termtoStringExtended2(f)(l)+ "-"
                   +termtoStringExtended2(f)(r)+")"

 case default => f(default)
}
```

Ahora vamos a ver como combinamos las distintas operaciones en una sola que maneje todos los casos. La idea general es componer todas las funciones en una sola, hay varias formar de hacer esto. Podemos declarar explícitamente una función que sea la composición de todas las anteriores:

```
def evalGeneral(t:Term):Int = eval(evalExtended1(evalExtended2(evalGeneral)))(t)

def toStringGeneral(t:Term): String =
  termtoString(termtoStringExtended1(termtoStringExtended2(toStringGeneral)))(t)
```

O podemos usar una característica de Scala llamada *typecase* [2], que es básicamente hacer pattern matching sobre el tipo de la variable y llamar a la función correspondiente para ese tipo:

```
def enlacePorTipos(t:Term): String = t match {
 case(t: ExtendedTerm2) => termtoStringExtended2(enlacePorTipos)(t)
 case(t: ExtendedTerm1) => termtoStringExtended1(enlacePorTipos)(t)
 case(t: Term) => termtoString(enlacePorTipos)(t)
}
```

En el caso de la función enlacePorTipos(), tenemos que tener cuidado en el orden de las cláusulas **case**, ya que ExtendedTerm1 es de tipo Term, por las reglas de tipado de la

herencia de clases, entonces si el caso **case**(t: Term) aparece primero en la definición, siempre se entrará en ese caso. Tenemos que poner primero los casos que aparecen más abajo en la estructura de herencia.
Las funciones evalGeneral() y toStringGeneral() así declaradas manejan todos los casos del tipo algebraico, es decir, los originales más todas las extensiones. También podemos declarar nuevas operaciones sobre nuestro tipo extendido, parametrizando nuevamente las funciones con un caso default para poder manejar futuras extensiones. Por ejemplo definimos una función que simplifica expresiones de nuestro lenguaje si se multiplica por cero o por uno, vemos que el pattern matching es natural sobre todos los casos:

```
def simplificar(f: Term => Term)(t:Term): Term = t match {
 case Mul(l, r) => (l,r) match {
  case (Num(0), _) => Num(0)
  case (Num(1), _) => r
  case (_, Num(0)) => Num(0)
  case (_, Num(1)) => l
  case _ => Mul(simplificar(f)(l), simplificar(f)(r))
 }
 case Plus(l,r) => Plus(simplificar(f)(l), simplificar(f)(r))
 case Diff(l,r) => Diff(simplificar(f)(l), simplificar(f)(r))
 case Num(n) => Num(n)
 case default => f(default)
}
```

3.2.2. Extensiones Independientes

Si en vez de tener extensiones lineales, tenemos extensiones independientes, es decir, si tenemos la extensión anterior de Term, ExtendedTerm1, y hacemos una nueva extensión ExtendedTerm2 que no herede de ExtendedTerm1, sino de Term:

```
abstract class ExtendedTerm2 extends Term
case class Diff(left: Term, right: Term) extends ExtendedTerm2

def evalExtended2(f: Term => Int)(t: Term): Int = t match{
 case Diff(l,r) => evalExtended2(f)(l) - evalExtended2(f)(r)
 case default => f(default)
}

def termtoStringExtended2(f: Term => String)(t: Term): String = t match{
 case Diff(l,r) => "("+termtoStringExtended2(f)(l)+ "-"
                   +termtoStringExtended2(f)(r)+")"

 case default => f(default)
}
```

La composición de las operaciones definidas es exactamente igual que para las extensiones lineales, es decir, no hay diferencia en la composición cuando la extensión es lineal o independiente. Esto nos permite combinar extensiones ortogonales de forma transparente.

```
def evalGeneral(t:Term):Int = eval(evalExtended1(evalExtended2(evalGeneral)))(t)

def toStringGeneral(t:Term): String =
  termtoString(termtoStringExtended1(termtoStringExtended2(toStringGeneral)))(t)
```

3.3. Extendiendo las Operaciones con Herencia de Clases

La extensión de operaciones mostrada en la sección anterior nos habilita a realizar una codificación funcional de la solución, ya que se define un tipo de datos algebraico y se declaran funciones sobre ese tipo, sin usar clases ni herencia para la implementación de las operaciones.
En esta sección vamos a empaquetar las operaciones en clases y vamos a extender las operaciones usando herencia de clases. Veremos que aunque utilicemos herencia, no perderemos la codificación funcional de la solución.

3.3.1. Extensiones Lineales

Tenemos el mismo tipo de dato algebraico, pero vamos a declarar una clase para definir las operaciones. Todas las funciones sobre el tipo van a estar empaquetadas dentro de una clase:

```
abstract class Term

case class Num(value: Int) extends Term
case class Plus(left: Term, right: Term) extends Term

class opBase {
 def eval(t:Term): Int = t match {
  case Num(v) => v
  case Plus(l,r) => eval(l) + eval(r)
 }
}
```

Declaramos la clase opBase, donde se definen las operaciones sobre el tipo algebraico, en este caso sólo definimos la función eval pero podemos declarar más de una función si es necesario. Entonces todas las operaciones sobre nuestro tipo algebraico Term van a estar empaquetadas en la clase opBase.

Ahora hacemos una extensión del tipo algebraico:

```
abstract class ExtendedTerm1 extends Term

case class Mul(left: Term, right: Term) extends ExtendedTerm1

class opExtended1 extends opBase {
 override def eval(t: Term): Int = t match{
  case Mul(l,r) => eval(l) * eval(r)
  case default => super.eval(default)
 }

 def termtoString(t: Term): String = t match{
  case Num(v) => v.toString
  case Plus(l,r) => "("+termtoString(l)+"+"+termtoString(r)+")"
  case Mul(l,r) => "("+termtoString(l)+"*"+termtoString(r)+")"
 }
}
```

La extensión del tipo es igual a las que hicimos en la sección anterior, pero para extender las operaciones usamos herencia de clases. La clase opExtended1 se hereda de opBase y se sobreescriben las operaciones, además agregamos una nueva operación termtoString. En el pattern matching de las operaciones extendidas también se agrega un caso default, pero esta vez se llama al método de la clase base, que es el que va a manejar los casos del tipo Term original. Al igual que antes, las extensiones de las operaciones van a manejar sólo los casos introducidos en la extensión del tipo, los demás se manejarán por los métodos de la clase base, con la llamada **super**.eval(default). Esa llamada a **super**, se refiere al método opBase.eval(), que está definido recursivamente. Se podría pensar que una vez ejecutada esa llamada el método no retorna y falla, por ejemplo en la siguiente ejecución: si llamamos a opExtended1.eval() con el parámetro Plus(Mul(Num(2), Num(3)), Mul(Num(4), Num(5))), como la operación no está definida para el caso Plus, va a entrar al caso default, donde se va a ejecutar la llamada **super** con el mismo parámetro, esa llamada va a ejecutar el método opBase.eval(), en este método se va a entrar al caso Plus(l,r) y se va a hacer el llamado recursivo, esta vez de la forma:

```
eval(Mul(Num(2), Num(3))) + "+" + eval(Mul(Num(4), Num(5)))
```

pero opBase.eval() no maneja el caso Mul, entonces habría un error de pattern matching. Sorpresivamente este no es el caso, ya que la llamada recursiva en realidad ejecuta opExtended1.eval(). La explicación para este comportamiento está en como Scala define los métodos recursivos (sigue el estilo de Java). Una llamada recursiva en la definición de un método de una clase de la forma:

```
def f(){
```

```
 // code
 f();
}
```

es en realidad una abreviatura sintáctica para:

```
def f(){
 // code
 this.f();
}
```

La expresión **this** denota un valor que es una referencia al objeto por el cual el método fue invocado. En el momento de la creación de un objeto con la expresión **new** C(), **this** se referencia al valor del objeto **new** C(). El tipo de **this** es la clase C donde la palabra **this** ocurre. En tiempo de ejecución, la clase del objeto referenciado puede ser la clase C o cualquier subclase de C. En general, **this** tiene el mismo tipo que el objeto al cual referencia [9, 4].
La referencia **super**.m refiere estáticamente al método m en el supertipo inmediato del objeto. Se evalúa al método m' en el supertipo que es igual a m o sobreescribe m (explícitamente con **override**) [9, 4]. Es decir que la expresión **super**.m sirve para acceder a un método de la superclase que fue sobreescrito, ejecutando el código original en vez del código con el cual se sobreescribió el método.
Veamos un ejemplo para aclarar un poco las cosas:

```
class Base {

 def rec(lista : List[Int]) : Unit = {
  println("*****Entro a Base.rec()");
  lista match {
   case Nil =>
   case x::xs => {
    println("*****Base.rec() − elemento: " + x)
    println("*****Hago el llamado recursivo desde Base")
    println("*****Tipo de this en Base: "+this.getClass().getName())
    rec(xs)}
  }
 }
}
class Extended extends Base{

 override def rec(lista : List[Int]) : Unit = {
  println("Entro a Extended.rec()")
  lista match {
   case Nil =>
   case x::xs => {
    println("Extended.rec() − elemento: " + x)
    println("Hago el llamado recursivo desde Extended")
    super.rec(xs)}
```

```
  }
 }
}
object Main {

 def main(args: Array[String]) : Unit = {
  val objeto : Base = new Extended()
  println("El tipo del objeto es: "+objeto.getClass().getName())
  println("LLAMO A EXTENDED.REC() DESDE MAIN: ")
  objeto.rec(List(1,2,3,4))
 }
}
```

En el código anterior declaramos una clase llamada Base con un único método rec() definido recursivamente, que toma una lista de enteros y va extrayendo y mostrando por pantalla cada elemento de la lista, además mostramos el tipo de la referencia **this** en tiempo de ejecución usando *reflection*. Luego declaramos otra clase Extended que hereda de Base y sobreescribe el método rec(), que hace un llamado recursivo, pero al método rec() de la clase Base con **super**.rec(). En el método main declaramos un objeto, mostramos su tipo en tiempo de ejecución y llamamos a rec().
La salida de la ejecución es la siguiente:

El tipo del objeto es: Extended
LLAMO A EXTENDED.REC() DESDE MAIN:
Entro a Extended.rec()
Extended.rec() - elemento: 1
Hago el llamado recursivo desde Extended
******Entro a Base.rec()*
******Base.rec() - elemento: 2*
******Hago el llamado recursivo desde Base*
******Tipo de this en Base: Extended*
Entro a Extended.rec()
Extended.rec() - elemento: 3
Hago el llamado recursivo desde Extended
******Entro a Base.rec()*
******Base.rec() - elemento: 4*
******Hago el llamado recursivo desde Base*
******Tipo de this en Base: Extended*
Entro a Extended.rec()

En la salida del programa se puede ver que la referencia **this** de la clase Base tiene el mismo tipo que el objeto que hace la llamada, en este caso Extended. Entonces en el llamado recursivo de Base.rec(), expresado como **this**.rec(), en realidad se llama al método Extended.rec() ya que **this** es de tipo Extended.
Si volvemos a extender el tipo algebraico, el esquema es el mismo:

```
abstract class ExtendedTerm2 extends ExtendedTerm1

case class Diff(left: Term, right: Term) extends ExtendedTerm2

trait Simplificacion{
 def simplificar(t:Term): Term = t match{
  case Mul(l, r) => (l,r) match {
   case (Num(0), _) => Num(0)
   case (Num(1), _) => r
   case (_, Num(0)) => Num(0)
   case (_, Num(1)) => l
   case _ => Mul(simplificar(l), simplificar(r))
  }
  case Plus(l,r) => Plus(simplificar(l), simplificar(r))
  case Diff(l,r) => Diff(simplificar(l), simplificar(r))
  case Num(n) => Num(n)
 }
}

class opExtended2 extends opExtended1 with Simplificacion {
 override def eval(t: Term): Int = t match {
  case Diff(l,r) => eval(l) - eval(r)
  case default => super.eval(default)
 }

 override def termtoString(t: Term): String = t match {
  case Diff(l,r) => "("+termtoString(l)+"-"+termtoString(r)+")"
  case default => super.termtoString(default)
 }
}
```

Como la extensión es lineal, opExtended2 se hereda de opExtended1. En este caso la llamada **super**.eval(default) de opExtended2 llama al método de la superclase opExtended1, que a su vez puede llamar a métodos de la clase opBase con otra llamada a **super**. Las llamadas a métodos de la clase base con **super** sigue el esquema de herencia de las clases, ver figura 3.1. El esquema de herencia mostrado es simple y lineal por el tipo de extensión que se hizo.
En esta extensión también agregamos una nueva operación sobre el tipo: simplificar, pero esta vez la agregamos utilizando un **trait** con una característica de Scala llamada *mixin composition* [11]. Un trait puede ser mezclado con muchas clases y una clase puede mezclarse con muchos traits, inclusive puede mezclarse un trait con un objeto concreto, además un trait puede tener implementaciones concretas de métodos, como en el caso de Simplificacion que ya tiene el método definido. Entonces para agregar operaciones sobre nuestro tipo, podemos agregar métodos cuando extendemos la clase de las operaciones o podemos declarar un nuevo trait con todas las operaciones que queremos agregar y lo mezclamos con la clase donde se definen las operaciones,

en este caso opExtended2.
Inclusive podemos mezclar un **trait** con un objeto concreto en su declaración, por ejemplo si no mezclamos Simplificacion con la clase opExtended2 lo podemos mezclar con una instancia concreta de la clase, como se muestra a continuación:

```
val term1 = Plus(Num(2), Num(3))
val term2 = Plus(Diff(Num(3), Num(2)), Mul(Plus(Num(1), Num(0)), Num(1)))

trait Simplificacion{

 def simplificar(t:Term): Term = t match{
  case Mul(l, r) => (l,r) match {
   case (Num(0), _) => Num(0)
   case (Num(1), _) => r
   case (_, Num(0)) => Num(0)
   case (_, Num(1)) => l
   case _ => Mul(simplificar(l), simplificar(r))
  }
  case Plus(l,r) => Plus(simplificar(l), simplificar(r))
  case Diff(l,r) => Diff(simplificar(l), simplificar(r))
  case Num(n) => Num(n)
 }
}

val op = new opExtended2 with Simplificacion

op.termtoString(term2)
op.simplificar(term2)
```

El objeto op va a tener todos los métodos definidos en opExtended2 más todos los definidos en el trait Simplificacion. El problema es que con esta declaración perdemos la capacidad de extender la función simplificar.
Al estar codificando en un estilo funcional, donde las operaciones sobre el tipo están separadas, podemos también definir nuevas funciones en una nueva clase independiente de las anteriores, de esta manera tendremos dos clases distintas donde se definen todas las funciones sobre el tipo:

```
class SimplificacionSum {
 def simplificarSuma(t:Term): Term = t match {
  case Plus(l,r) => (l,r) match {
   case (Num(0), _) => r
   case (_, Num(0)) => l
   case _ => Plus(simplificarSuma(l), simplificarSuma(r))
  }
  case Mul(l,r) => Mul(simplificarSuma(l), simplificarSuma(r))
  case Diff(l,r) => Diff(simplificarSuma(l), simplificarSuma(r))
  case Num(n) => Num(n)
 }
```

```
}

val newop = new SimplificacionSum
```

3.3.2. Extensiones Independientes

El problema con las extensiones independientes es que las llamadas a **super** ahora no siguen la estructura de herencia, y eso hace que pueda haber casos sin tratar en el pattern matching. En estas extensiones, nos van a quedar dos clases opExtended1 y opExtended2 que heredan de opBase, pero independientes entre si, entonces las llamadas a **super** de cualquiera de ellas van a llamar a métodos de la clase opBase. Por ejemplo, si llamamos a opExtended2.eval() y nos encontramos con el caso de Mul(l,r), como ese caso no está especificado en el pattern matching de esta operación, va a caer en el caso por default y se va a hacer la llamada **super**.eval(default), esa llamada va a ser manejada por la operación opBase.eval(), que tampoco especifica un caso del pattern matching para el constructor Mul. Entonces todo el pattern matching va a fallar y la operación completa fallará.
Una forma de solucionar esto utilizando herencia, es hacer, al igual que en las extensiones lineales, que opExtended2 herede de opExtended1, pero como las extensiones son independientes, cuando se define opExtended2 puede no haber conocimiento aún de la extensión opExtended1, más aún, opExtended1 puede no existir. Solucionaremos este problema recurriendo a otra característica de los traits de Scala, vamos a utilizar *modular mixin composition* para lograr *modificaciones apiladas* [11, 13]. El código que lo implementa queda:

```
abstract class Term

case class Num(value: Int) extends Term
case class Plus(left: Term, right: Term) extends Term

abstract class Operaciones{
 def eval(t: Term): Int
 def termtoString(t: Term): String
}

class opBase extends Operaciones{

 def eval(t:Term): Int = t match {
  case Num(v) => v
  case Plus(l,r) => eval(l) + eval(r)
 }

 def termtoString(t: Term):String= t match {
  case Num(v) => v.toString
  case Plus(l,r) => "("+termtoString(l)+"+"+termtoString(r)+")"
```

```
 }
}
```

En esta nueva implementación, Operaciones es una clase abstracta que define la interfaz de las operaciones sobre el tipo y la clase concreta opBase extiende de ella e implementa las operaciones. Hacemos la primer extensión:

```
abstract class ExtendedTerm1 extends Term

case class Mul(left: Term, right: Term) extends ExtendedTerm1

trait opExtended1 extends Operaciones {
 abstract override def eval(t: Term): Int = t match{
  case Mul(l,r) => eval(l) * eval(r)
  case default => super.eval(default)
 }

 abstract override def termtoString(t: Term): String = t match{
  case Mul(l,r) => "("+termtoString(l)+"*"+termtoString(r)+")"
  case default => super.termtoString(default)
 }
}
```

Ahora las operaciones sobre la extensión están declaradas como un trait, opExtended1, que hereda de Operaciones y no como una clase. Esto va a hacer posible que se puedan mezclar los traits para que las llamadas a **super** se encadenen correctamente y manejen todos los casos del tipo. El trait opExtended1 tiene una sutileza en su implementación. En los métodos eval() y termtoString() hay una llamada a **super**, pero esa llamada hace referencia a un método abstracto de la clase Operaciones. Ese llamado sería ilegal para clases normales, pero está permitido en un trait, siempre y cuando ese trait se mezcle con una clase u otro trait que implemente esos métodos abstractos, ya que una llamada a **super** en un trait se resuelve dinámicamente.
Hagamos otra extensión independiente al tipo:

```
abstract class ExtendedTerm2 extends Term

case class Diff(left: Term, right: Term) extends ExtendedTerm2

trait opExtended2 extends Operaciones {
 abstract override def eval(t: Term): Int = t match {
  case Diff(l,r) => eval(l) - eval(r)
  case default => super.eval(default)
 }
 abstract override def termtoString(t: Term): String = t match {
  case Diff(l,r) => "("+termtoString(l)+"-"+termtoString(r)+")"
  case default => super.termtoString(default)
```

```
  }
 }
```

En este caso ExtendedTerm2 hereda de Term y no de ExtendedTerm1. Nuevamente opExtended2 es un trait que hereda de Operaciones.
Ahora falta mezclar los trait de las dos extensiones de manera que las llamadas a **super** se encadenen correctamente para que manejen los casos de todas las extensiones. Esto se logra mezclando los traits como se muestra en la declaración de op:

```
val term2 = Plus(Diff(Num(3), Num(2)), Mul(Plus(Num(1), Num(0)), Num(1)))

trait Simplificacion{

 def simplificar(t:Term): Term = t match{
  case Mul(l, r) => (l,r) match {
   case (Num(0), _) => Num(0)
   case (Num(1), _) => r
   case (_, Num(0)) => Num(0)
   case (_, Num(1)) => l
   case _ => Mul(simplificar(l), simplificar(r))
  }
  case Plus(l,r) => Plus(simplificar(l), simplificar(r))
  case Diff(l,r) => Diff(simplificar(l), simplificar(r))
  case Num(n) => Num(n)
 }
}
val op = new opBase with opExtended1 with opExtended2 with Simplificacion
```

El orden en que se mezcla la clase opBase con los traits es importante. Cuando se llama a un método de una clase mezclada con traits, el método en el trait más a la derecha es el que se llama primero. Si ese método hace una llamada a **super**, se invoca al método del siguiente trait hacia la izquierda. Estas reglas están dadas por la *linearización de clases* [13, 11]. En el caso anterior, la llamada **super** en opExtended2 va a llamar a métodos de opExtended1 y la llamada **super** de opExtended1 va a llamar a opBase.
En este capítulo se mostraron los tipos de datos algebraicos extensibles y dos formas de extender las operaciones para que soporten los nuevos casos del tipo. De esta manera, si tenemos un módulo con un tipo algebraico y funciones sobre ese tipo, y queremos agregar funcionalidades al mismo, extendiendo el tipo de datos, podemos programar otro módulo que extienda el tipo de datos original y todas las operaciones sobre él, definiendo *nuevas* funciones que reutilizan a las originales. En el capítulo siguiente vamos a mostrar una forma de codificar las operaciones para que podamos utilizar las *mismas* funciones para operar sobre las extensiones del tipo original.
Cabe destacar que la implementación de tipos algebraicos extensibles con defaults presentada en este capítulo permite realizar extensiones independientes del tipo de dato y luego combinarlas para utilizarlas juntas, esto no era posible en la propuesta

original hecha por Zenger y Odersky en [19].

3.4. Limitaciones

Una de las limitaciones del la solución propuesta en este capítulo es que requiere que se extiendan demasiadas funciones al agregar un nuevo caso al tipo. En particular, cualquier función que en su interior haga un llamado a otra función definida para el tipo original debe ser redefinida, declarando una nueva función. Por ejemplo la función evalSimpl(), para el tipo original, que primero simplifica todas las sumas de la expresión y luego la evalúa pude ser declarada como la composición de otras dos funciones:

```
class Simpl {
 def simplificarSuma(t:Term): Term = t match {
  case Plus(l,r) => (l,r) match {
   case (Num(0), _) => r
   case (_, Num(0)) => l
  case _ => Plus(simplificarSuma(l), simplificarSuma(r))
  }
  case Num(n) => Num(n)
 }
}


val op = new opBase
val simplOp = new Simpl

def evalSimpl = op.eval _ compose newop.simplificarSuma
```

La función evalSimpl() no está definida utilizando *pattern matching*, pero al extender el tipo de datos agregando nuevos casos y definiendo las funciones extendidas para que manejen esos casos, necesariamente debemos declarar una nueva función evalSimpl() que sea la composición de las funciones extendidas y no de las originales.
Otra limitación de la solución es que al utilizar herencia de clases para agregar nuevos casos, sacrificamos precisión en los tipos. Para una función que acepta como parámetro un tipo Term, no se puede verificar en tiempo de compilación que el *pattern matching* sea exhaustivo, así es posible pasar como parámetro un caso para el cuál la función no está definida, produciendo un error en tiempo de ejecución.

3.5. ¿Funciones o Métodos?

Hasta aquí venimos utilizando las palabras "función" y "método" como sinónimos. Aquel que provenga de la programación orientada a objetos dirá que los métodos sólo pueden definirse dentro de una clase. Las personas que provienen de la programación funcional objetarán que las funciones deben ser *first class values*. Vamos a

formalizar un poco las ideas. Scala es un lenguaje funcional y orientado a objetos por lo tanto provee ambas construcciones, cada una con su respectiva semántica. Al ser un lenguaje funcional, Scala provee funciones como valores de primera clase, es decir, las funciones pueden ser tratadas como un valor mas del lenguaje, en particular, pueden pasarse como parámetros a otras funciones y pueden ser devueltas como valores de retorno de otras funciones. Para esto el lenguaje implementa las mismas como objetos con una determinada interfaz. Un *function type* en Scala es de la forma $(T_1,\ldots,T_n) => U$ y representa el conjunto de *funciones* que toman argumentos con tipos $T_1,\ldots,T_n$ y devuelven un valor de tipo U [9]. Estos tipos funcionales son una abreviatura para tipos de clases que implementan una determinada interfaz, específicamente la función *N-aria* (T1,...,Tn) => U es una abreviatura para el tipo de clase que implementa el *trait*:

```
trait FunctionN [−T1 , . . . , −Tn , +R ] {
 def apply(x 1 : T1 , . . . , x n : Tn ): R
 override def toString = "<function>"
}
```

Donde N es el número de argumentos que toma la función (Function1, Function2, etc). Entonces, en Scala, una función es una instancia de una clase particular que implementa un método apply, es decir, es un objeto.
Por otro lado, un *method type* se denota (Ps)U, donde (Ps) es una secuencia de nombres de parámetros y sus respectivos tipos (p_1: T_1, ..., p_n: T_n), para algún $n \geq 0$ y U es el tipo de valor de retorno del método. Este tipo representa un método que toma parámetros p_1, ..., p_n con tipos T_1, ...,T_n y devuelve un resultado de tipo U. Los *method type* no existen como valores, es decir, un valor no puede tener ese tipo [9]. Veamos algunos ejemplos para aclarar lo anterior:

```
def sucm(n:Int) = n+1
```

sucm: (n: Int)Int

Scala tipa sucm como un método, ahora definimos la función anónima equivalente al método y la asignamos a un valor:

```
val sucf = (n:Int) => n+1
```

sucf: Int => Int = <function1>

Llamamos al método y a la función con el argumento 3 y obtenemos los mismos resultados:

```
sucm(3)
```

res0: Int = 4

```
sucf(3)
```

res1: Int = 4

Definimos otra función, pero esta vez como un objeto, heredando explícitamente del trait Function1[Int,Int]

```
object suco extends Function1[Int,Int]{
    def apply(n:Int) = n+1
}
```

suco.type = <function1>

```
suco(3)
```

res2: Int = 4

En Scala las expresiones Function1[Int,Int] y Int => Int son sintácticamente equivalentes, el objeto suco lo podríamos haber definido:

```
object suco extends (Int => Int){
    def apply(n:Int) = n+1
}
```

Ya vimos que las funciones pueden asignarse a valores como lo hicimos con sucf, también pueden pasarse como argumentos a otras funciones o a métodos:

```
val g = (f: Int => Int, n: Int) => f(n)
```

g: (Int => Int, Int) => Int = <function2>

```
g(sucf, 3)
```

res3: Int = 4

```
def mg(f: Int => Int, n :Int) = f(n)
```

mg: (f: Int => Int, n: Int)Int

```
mg(sucf,3)
```

res4: Int = 4

Los *method type* no existen como valor, por lo tanto no pueden pasarse como parámetros a una función o método ni pueden asignarse a un valor. Pero Scala permite ese comportamiento empleando una *Eta-expansion*. Esta expansión transforma una expresión con *method type* a una expresión equivalente con *function type* [9]. La *expansión-Eta* la aplica el compilador automáticamente cuando lo que se espera es una

función y en su lugar se encuentra un método, siempre y cuando los tipos coincidan, por ejemplo, para asignar un método a un valor, tenemos que agregar al nombre del método el carácter "_" y Scala aplicará la expansión-Eta, tipando el valor como una función:

```
val m = sucm _
```

m: Int => Int = <function1>

Y si pasamos un método a una función (o método) que espera como parámetro una función, Scala transformará el método pasado en una función equivalente, siempre y cuando los tipos coincidan:

```
g(sucm,3)
```

res5: Int = 4

Se puede pensar que dado el método m, la función anónima equivalente obtenida luego de la expansión-Eta es x => m(x), donde los tipos de los parámetros son los mismo que los tipos de los parámetros del método. Así en Scala podemos usar métodos y funciones en forma intercambiable, ya que si definimos un método y queremos que se comporte como una función, se aplicará una expansión-Eta.
Al implementar las funciones como objetos, en Scala podemos definir funciones extensibles directamente, sin definirlas primero como métodos, para ello definimos una clase que represente la función y la extendemos del trait FunctionN correspondiente, por ejemplo:

```
abstract class Term
case class Num(value: Int) extends Term
case class Plus(left: Term, right: Term) extends Term

class evalFun extends Function1[Term, Int] {
 def apply(t: Term): Int = t match {
  case Num(v) => v
  case Plus(l,r) => apply(l) + apply(r)
 }
}

val eval = new evalFun
println(eval(Plus(Num(2), Num(3))))

case class Mul(left: Term, right: Term) extends Term

class evalFunExtended extends evalFun {
 override def apply(t: Term): Int = t match {
  case Mul(l,r) => apply(l) * apply(r)
  case default => super.apply(t)
 }
```

```
}
val term2 = Plus(Plus(Num(3), Num(2)), Mul(Plus(Num(1), Num(0)), Num(2)))
val evalExt = new evalFunExtended
println(evalExt(term2))

val h = (g: Term => Int) => (t: Term) => g(t)
println(h(evalExt)(term2))
```

La salida del script anterior en el intérprete de Scala es:
5
7
7

eval y evalExt son funciones que pueden ser pasadas como parámetros a otras funciones como h (que está currificada).
Algunos ejemplos más de definición de funciones como clases:

```
class fobj extends Function2[Term=>Int,Term,Int] {
 def apply(f:Term=>Int, t:Term):Int = f(t)
}

class fopar extends Function1[Term=>Int,Term=>Int] {
 def apply(f:Term=>Int):Term=>Int = (x:Term) => f(x)
}
```

O equivalentemente:

```
class fobj2 extends ((Term=>Int, Term) => Int){
 def apply(f:Term=>Int, t:Term):Int = f(t)
}

class fopar2 extends ((Term=>Int) => (Term=>Int)) {
 def apply(f:Term=>Int):Term=>Int = (x:Term) => f(x)
}
```

Al definir funciones como clases que implementan el **trait** FunctionN, agregamos una sobrecarga al código, y es una manera de definir funciones que se aleja del estilo de los lenguajes funcionales clásicos, pero ganamos en extensibilidad de las mismas.

Capítulo 4

Tipos de Datos Abiertos y Funciones Abiertas

En el capítulo anterior se describió como puede extenderse un tipo de datos algebraico y las funciones definidas sobre él. Para ello es necesario definir *nuevas* funciones que manejan los nuevos casos del tipo, las mismas reutilizan el código que maneja los casos originales. Si tenemos un programa y queremos ampliar su comportamiento, podemos extender el tipo de datos algebraico y definir nuevas funciones sobre el tipo, sólo codificando el comportamiento para los nuevos casos. Con este enfoque es necesario definir nuevas funciones para cada una de las funciones originales del programa.
En este capítulo vamos a mostrar una manera de organizar el código y de definir funciones para que sea posible adaptarlas a las extensiones del tipo, y así no tener que redefinir todas las funciones originales. De esta manera será posible reutilizar las *mismas* funciones para manejar las nuevas extensiones del tipo de datos, "añadiendo" el código necesario para los nuevos casos del tipo.

4.1. Tipos de Datos Abiertos y Funciones Abiertas en Haskell

Los tipos de datos abiertos y las funciones abiertas fueron propuestos por Löh y Hinze como una extensión ligera de Haskell para resolver el *expression problem* [7]. Los constructores de un tipo de datos abierto y las ecuaciones de una función abierta pueden aparecer dispersas en el código y en distintos módulos, es decir, se pueden ir declarando constructores del tipo de datos a medida que se vayan necesitando en el programa y por cada nuevo caso que se declara, se puede agregar una ecuación a la función sobre el tipo para manejar ese nuevo caso. La semántica pretendida es simple: el programa debería comportarse como si el tipo de datos y las funciones fuesen cerradas, definidas en un único lugar.
La siguiente declaración introduce un tipo de datos abierto:

```
open data Expr::*
```

La bandera open indica que lo que se está declarando es un tipo de datos abierto, en lugar de un tipo ordinario, por eso la declaración no lista los constructores. Los constructores de un tipo de datos abierto son entidades separadas que pueden ser declarados dando su signatura de tipos en cualquier parte del programa, por ejemplo la declaración:

```
Num::Int -> Expr
```

Introduce un nuevo constructor para el tipo Expr. Las funciones abiertas son declaradas utilizando también la bandera open:

```
open eval::Expr -> Int
```

Y las ecuaciones que definen la función abierta pueden ser introducidas en cualquier punto del programa donde el nombre de la función esté en el *scope*:

```
eval(Num n) = n
```

Los tipo de datos abiertos y las funciones abiertas soportan extensiones en ambas dimensiones [7]. Una nueva función sobre el tipo puede ser introducida como se hace usualmente en Haskell, pero declarada como *open*:

```
open termtoString::Expr -> String
termtoString (Num n) = show n
```

Es también sencillo agregar un nuevo constructor para el tipo Expr, todo lo que debemos hacer es dar su signatura de tipo:

```
Plus::Expr -> Expr -> Expr
```

Cada vez que introducimos un nuevo constructor, debemos adaptar las funciones sobre el tipo Expr definidas hasta el momento para que manejen el nuevo caso, ya que si no lo hacemos y llamamos a alguna de ellas sobre una expresión construida con Plus, tendremos un error de pattern matching:

```
eval(Plus l r) = eval l + eval r

termtoString(Plus l r) = "("++termtoString l ++ ")"
    ++ termToString r ++ ")"
```

La implementación de los tipos de datos abiertos y las funciones abiertas propuesta en [7] es simple: se transforma un programa que utiliza estas nuevas construcciones en otro programa que no las utiliza. Todos los constructores de un tipo de dato abierto son recolectados y aparecen como constructores en una definición de un tipo de dato cerrado ordinario de Haskell. Similarmente se agrupan todas las ecuaciones que pertenecen a una función abierta, convirtiéndola en una definición de función cerrada. Esta implementación de las extensiones abiertas no tiene ninguna consecuencia para el sistema de tipos de Haskell ni para la semántica de otras construcciones del

lenguaje.
La forma en la que se recolectan y agrupan las declaraciones separadas requiere de un análisis. El orden de los constructores del tipo de datos es irrelevante, sin embargo el orden en el que las ecuaciones aparecen en una función es importante, ya que Haskell utiliza *first-fit pattern matching*, es decir, las ecuaciones son recorridas de arriba hacia abajo, la primer ecuación que machea es seleccionada. Esto puede provocar complicaciones al recolectar las ecuaciones que definen una función abierta, ya que un pattern genérico definido por encima de otros mas específicos hará que nunca se llegue a los patrones mas específicos. Por estos motivos los autores proponen que en vez de utilizar *first-fit pattern matching*, utilizar un esquema *best-fit*, donde el patrón mas específico es seleccionado por sobre un patrón mas general.
Hasta ahora no hay ningún compilador de Haskell que soporte la extensión propuesta.

4.2. Implementación en Scala con Herencia y Casos por Default

En esta sección vamos a ver como podemos modelar las ideas de tipos de datos y funciones abiertas en Scala utilizando herencia de clases. Vamos a ver como podemos simular la semántica de dichas construcciones con las características que ofrece Scala y los conceptos vistos en el capítulo 3, sin la necesidad de modificar el compilador y sin utilizar *best-fit pattern matching*.
En Scala un tipo de dato algebraico puede ser extendido agregando nuevos casos fácilmente utilizando herencia de clases. Esa extensión puede efectuarse en cualquier punto del programa mientras que la clase base del tipo algebraico sea visible. Por lo tanto el modelado de tipos de datos abiertos es directo.
Las funciones abiertas son más complicadas de modelar, ya que la semántica indica que debe ser posible *agregar* nuevas ecuaciones a la función, no sólo redefinir la función. En el capítulo 3 definíamos una *nueva* función para manejar los nuevos casos que reutilizaba a la función previamente definida para los casos antiguos. Este no es el comportamiento que estamos buscando, ya que si definimos una función f que en su cuerpo hace un llamado a otra función g definida sobre el tipo de datos algebraico, luego extendemos el tipo de datos pero no agregamos ecuaciones a g, sino que definimos una nueva función g' que se comporte como g pero pueda procesar también los nuevos casos, la función f va a quedar definida con g, que no procesa los nuevos casos y la única forma de extender a f es redefiniéndola para que utilice g' en vez de g. Entonces lo que tenemos que hacer es buscar una forma de simular el agregado de ecuaciones para manejar los nuevos casos. Veamos como podemos simular dicha característica, tenemos como antes el tipo de datos original:

```
abstract class Term

case class Num(value: Int) extends Term
case class Plus(left: Term, right: Term) extends Term
```

Ahora vamos a definir una clase con un solo método llamado apply() que representará las ecuaciones de la función sobre los casos del tipo algebraico definidos hasta el momento, es decir las ecuaciones de la función se encapsularán en una clase:

```
class operacion {
 def apply(t:Term): Int = t match {
  case Num(v) => v
  case Plus(l,r) => apply(l) + apply(r)
 }
}
```

Scala trata de una manera especial a los objetos de una clase que posee un método llamado apply(). Si un objeto posee un método apply() , se puede invocar ese método sin hacer una referencia explícita al mismo, sino que basta con poner la lista de parámetros justo después del nombre objeto:

```
val op = new operacion

op(Num(5))
```

El compilador de Scala automáticamente transforma el llamado anterior en:

```
op.apply(Num(5))
```

Entonces podemos utilizar objetos de la clase operacion como si fuesen una función. En este punto cabe hacer una aclaración, los objetos definidos con un método apply() se comportan como si fuesen una función, pero esto es sólo en la forma de llamar al método, ya que no son una función, por ejemplo si definimos una función de alto orden, que recibe como parámetro otra función y le pasamos una instancia de la clase operacion:

```
def h(f: Term => Int) : Int= f(Num(5))

println(h(op))
```

Obtenemos el siguiente error:

error: type mismatch;
found : this.operacion
required: this.Term => Int
println(h(op))
one error found

El compilador se queja porque la función h espera como parámetro una función de tipo Term => Int y en cambio recibió un objeto de tipo operacion. Entonces una instancia de operacion se comporta como una función, pero sigue siendo una instancia común de una clase, el comportamiento especial se lo da el azúcar sintáctico de Scala. Si queremos pasar el método de la clase operacion a la función h, que espera

una función, tenemos que nombrar explícitamente al método, para que el compilador aplique una expansión-Eta:

```
println(h(op.apply))
```

Para solucionar el inconveniente antes mencionado, podemos hacer que operacion sea una función, declarando la clase operacion para que implemente el trait Function1[A,B], pero para la implementación de las funciones abiertas aquí presentada, bastará con la primer definición de la clase operacion, más adelante veremos por qué.
Ahora creamos un objeto concreto de la clase operacion, pero lo declaramos con la palabra **implicit** y también definimos una función eval() para el tipo de datos Term:

```
implicit val op = new operacion

def eval(t:Term)(implicit op: operacion): Int = op(t)
```

La función eval() será nuestra representación de la función abierta. Toma un parámetro de tipo Term y además toma otro parámetro *implícito* de tipo operacion. Un método con un parámetro implícito puede ser aplicado a argumentos como un método normal. En el caso en que el parámetro marcado como implícito del método falte, es decir, no sea pasado, tal argumento será pasado automáticamente por el compilador [11]. Las reglas formales que utiliza el compilador para seleccionar un argumento para ser pasado como parámetro implícito a una función son complejas y pueden ser consultadas en [11], para este trabajo bastará saber que un argumento elegible para ser pasado como parámetro implícito puede ser cualquier identificador declarado como **implicit** visible en el *scope* actual de invocación, ya sea por declaración explícita, importación, herencia o a través de un objeto package. Si hay varios argumentos elegibles que concuerdan con el tipo del parámetro implícito, el mas específico de ellos será elegido usando las reglas para la resolución del *static overloading*. En este caso siempre se elegirá la clase derivada, es decir si A es un objeto declarado como implícito y B es otro objeto declarado como implícito cuya clase es una clase derivada de la clase de A, B es elegido por sobre A.
La función eval() es la que utilizaremos para representar nuestra función abierta sobre el tipo abierto Term, esta función hace una llamada al objeto operacion declarado como implícito. Ahora vamos a extender el tipo de datos y a agregar una nueva ecuación a la función para que pueda procesar esa extensión. La extensión del tipo se hace con herencia de clases como antes, para agregar una nueva ecuación, sin modificar la declaración de la función eval() lo que hacemos es declarar una nueva clase que extiende de operacion y en su método apply() definimos la ecuación que manejará el nuevo caso del tipo, y agregaremos un caso por default como hicimos con los tipos de datos algebraicos extensibles del capítulo 3, en ese caso por default se llamará al método apply() de la clase base, para que maneje el resto de los casos, así reutilizamos el código ya escrito para los otros casos. De esta manera sólo tenemos que agregar código para el caso que acabamos de extender. Luego creamos una instancia de la

nueva clase derivada y lo declaramos como **implicit**, entonces al utilizar la función eval() el compilador pasará como parámetro implícito el nuevo objeto, por las reglas de desempate mencionadas anteriormente. De esta forma sólo tenemos que agregar una nueva ecuación, encapsulada en un objeto, pero no es necesario modificar la función eval() para que maneje el nuevo caso introducido. El código quedaría:

```
case class Mul(left: Term, right: Term) extends Term

class operacionExtended extends operacion {
 override def apply(t: Term): Int = t match {
  case Mul(l,r) => apply(l) * apply(r)
  case default => super.apply(default)
 }
}


implicit val op2 = new operacionExtended
```

Ahora si llamamos a la función eval() con una expresión que utiliza el constructor Mul, podrá procesarlo ya que el parámetro implícito que se le pasa es op2 que tiene la ecuación necesaria para hacerlo, además como el caso por default llama al método apply() de la clase base, también puede procesar los constructores originales del tipo. El esquema de extensión mostrado soporta todas las extensiones que sean necesarias del tipo. Por cada extensión se debe crear una clase nueva, que defina un método apply() con la ecuación necesaria para manejar el nuevo caso, por ejemplo podemos volver a extender el tipo, agregando el caso Diff y añadimos una ecuación para manejar ese caso:

```
case class Diff(left: Term, right: Term) extends Term

class operacionExtended2 extends operacionExtended {
  override def apply(t: Term): Int = t match {
  case Diff(l,r) => apply(l) - apply(r)
  case default => super.apply(default)
 }
}

implicit val op3 = new operacionExtended2
```

De esta forma, podemos utilizar la *misma* función eval() para todas las extensiones del tipo, no necesitamos redefinir la función para cada extensión que hagamos, como lo hacíamos en el capítulo 3.
Una observación importante es que, aunque eval() esté definida con un parámetro implícito, podemos utilizarla como una función que sólo espera un parámetro de tipo (t: Term) y devuelve un Int, es decir, podemos utilizarla como si su tipo funcional fuera Term => Int, para ver esto recurrimos a un ejemplo:

```
val function = (f: Term => Int) => (t: Term) => f(t)
```

Definimos una función function que toma como parámetro otra función f con tipo Term => Int y devuelve una función que toma como parámetro una expresión t de tipo Term y devuelve f(t) que es de tipo Int. Scala tipa la función function como *(Term => Int) => (Term => Int) = <function1>*. La pregunta es ¿Podemos pasarle como parámetro nuestra función eval()? La respuesta es si. El compilador de Scala primero fija los parámetros implícitos en tiempo de compilación y luego aplica la *expansión-Eta* [9]. Para verlo definimos una función g como una aplicación parcial de function, parametrizándola con eval:

```
val g = function(eval)
```

Scala tipa la función g como *Term => Int = <function1>*. En caso que no haya un valor implícito para pasar como parámetro a eval tendremos un error de compilación. Podemos utilizar la función abierta eval() como una función común y corriente en Scala.
Antes de concluir esta sección tenemos que hacer una aclaración con respecto al código. Debido a un extraño comportamiento del intérprete de Scala, al ejecutar el código tal cual lo hemos presentado, obtenemos un *NullPointerException* debido a que los valores implícitos no llegan a inicializarse. Esto ocurre sólo en el intérprete, si compilamos y ejecutamos el programa no obtenemos ningún error. Este es un comportamiento poco normal, donde el intérprete del lenguaje y el compilador se comportan de manera distinta. Para evitar ese error al correr el código en el intérprete tenemos que declarar los valores implícitos como *lazy*, para que se evalúen cuando se utilizan por primera vez: **implicit lazy val** op = **new** operacion. [1]

4.3. Extensiones Independientes

Con los tipos de datos abiertos y las funciones abiertas podemos ir extendiendo el tipo de datos a medida que vayamos necesitando nuevos casos, y podemos agregar ecuaciones a una función para que maneje esos casos. De esta manera podemos utilizar la misma función para tratar todos los casos. Este es un esquema de extensión lineal. Si dos programadores agregan sus propios casos al tipo y las correspondientes ecuaciones a la función en forma independiente uno del otro, ¿Es posible combinar todos los casos y todas las ecuaciones? ¿Qué pasa con las extensiones independientes? Con los tipos de datos abiertos en Scala es posible también combinar extensiones independientes, modificando un poco la implementación que hicimos en la sección anterior. Como lo hicimos en el capítulo 3, recurrimos a los *traits* para lograr este tipo de combinación. Las ecuaciones a agregar en una función estarán representadas por un *trait* en en vez de una *clase*, y el agregado lo hacemos utilizando *mixin composition*.
El tipo base y la función sobre él:

[1] Este comportamiento lo encontré realizando este trabajo y lo publiqué en listas y foros del lenguaje, pero aún no fue reportado como bug del intérprete.

```
abstract class Term

case class Num(value: Int) extends Term
case class Plus(left: Term, right: Term) extends Term


abstract class Op {
 def apply(t:Term): Int
}

class operacion extends Op{
 def apply(t:Term): Int = t match {
  case Num(v) => v
  case Plus(l,r) => apply(l) + apply(r)
 }
}

implicit val op = new operacion

def evall(t:Term)(implicit op: operacion): Int = op(t)
```

Igual que antes, las ecuaciones para manejar los casos originales del tipo estarán representadas por la clase operacion, pero esta ahora hereda de una clase abstracta Op que define sólo un método apply. Al hacer extensiones del tipo, las ecuaciones a agregar en la operación van a estar representadas por un **trait** que también hereda de Op. Vamos a hacer una extensión agregando el caso Mul. Creamos una instancia de la clase operacion y la mezclamos con el trait operacionExtended, para que la operación pueda manejar el nuevo caso:

```
case class Mul(left: Term, right: Term) extends Term

trait operacionExtended extends Op {
 abstract override def apply(t: Term): Int = t match {
  case Mul(l,r) => apply(l) * apply(r)
  case default => super.apply(default)
 }
}

implicit val op2 = new operacion with operacionExtended
```

Ahora agregamos otro caso al tipo, pero en forma independiente del anterior. Se puede pensar que los dos nuevos casos se declaran en paquetes separados donde solo se importa el módulo que contiene el tipo original. Obsérvese que en el siguiente código no se hace ninguna referencia al código del listado anterior, solo al código original, ya que las dos extensiones son independientes, sólo necesitamos el tipo base y sus operaciones.

```
case class Diff(left: Term, right: Term) extends Term
```

```
trait operacionExtended2 extends Op {
  abstract override def apply(t: Term): Int = t match {
  case Diff(l,r) => apply(l) − apply(r)
  case default => super.apply(default)
 }
}

implicit val op3 = new operacion with  operacionExtended2
```

Como las extensiones son independientes, op2 tiene ecuaciones para manejar los casos Num, Plus, Mul y op3 tiene ecuaciones para manejar los casos Num, Plus, Diff. Para mezclar ambas extensiones, haciendo que evall pueda manejar todos los casos definimos un nuevo objeto implícito de la siguiente manera:

```
implicit val opTotal =
      new operacion with operacionExtended with operacionExtended2
```

De esta manera las llamadas a **super** se van a encadenar como se explicó en el capítulo 3 y evall va a poder manejar todos los casos. En este caso, el objeto que esté mezclado con el mayor número de *traits* será más específico y será elegido por sobre los demás. Los objetos op2 y op3 son igual de específicos y no puede haber desempate entre ellos, asique si ambos están en el *scope* de invocación sin que haya otro valor implícito más específico como opTotal, el compilador se quejará porque hay parámetros implícitos ambiguos.
En este capítulo se dio una implementación de los tipos de datos abiertos y de las funciones abiertas, utilizando los tipos algebraicos extensibles y los *parámetros implícitos* soportados por Scala. Los tipos abiertos y las funciones abiertas, presentados en [7], fueron propuestos originalmente como una solución funcional al *expression problem* para el lenguaje Haskell. Esta solución requería la modificación del compilador y del esquema de *pattern matching* del lenguaje. En este trabajo se implementó la semántica de los tipos y funciones abiertas sin necesidad de modificación del compilador ni del esquema de *pattern matching* de Scala, pudiendo utilizarse como solución al *expression problem* en Scala, manteniendo el estilo funcional de la misma.

Capítulo 5

Extensión del *Fold*

En la programación funcional, la utilización de funciones de alto orden permite encapsular patrones comunes de programación como funciones. Uno de los patrones más comunes es el de la recursión sobre el tipo de dato algebraico, este patrón es encapsulado por la función *fold*. Una gran cantidad de funciones (recursivas) sobre el tipo de datos pueden ser expresadas con la función *fold*. Al ser la función *fold* tan común en la programación funcional, vamos a ver como podemos definirla, de manera que sea fácil de extender para que soporte las extensiones del tipo.
En este capítulo utilizaremos los tipos algebraicos extensibles con defaults para definir una versión extensible de la función *fold*. También seguiremos el esquema presentado en [16] para definir funciones *fold* genéricas, que puedan utilizarse en combinación con los tipos extensibles presentados en el capítulo 3 y con los tipos abiertos del capítulo 4.

5.1. *Fold* para el tipo Term

Veamos primero cómo podemos definir la función *fold* para nuestro tipo de dato Term y cómo podemos utilizarla para definir funciones sobre el tipo. La función *fold* sobre el tipo original tomará un objeto de tipo Term y dos funciones, una para cada constructor del tipo, y estará parametrizada con el tipo de retorno, es decir, es polimórfica con respecto al tipo del valor que devuelve:

```
abstract class Term

case class Num(value: Int) extends Term
case class Plus(left: Term, right: Term) extends Term

def foldTerm[B](f : Int=>B)(g: B => B => B)(t:Term):B = t match {
 case Num(n) => f(n)
 case Plus(l,r) => g(foldTerm(f)(g)(l))(foldTerm(f)(g)(r))
}
```

Y podemos definir la función evaluadora de la siguiente manera:

```
def eval = foldTerm((x:Int) => x)((x:Int) => (y:Int) => x+y) _
```

También podemos nombrar las funciones que pasamos como parámetro a foldTerm, para poder reutilizarlas después, en ese caso la definición de eval() quedaría:

```
val id = (x:Int) => x
val sum = (x:Int) => (y:Int) => x+y

def eval = foldTerm (id) (sum) _
```

5.2. *Fold* Extensible para el tipo Term

Ahora tenemos que extender la función foldTerm() para que maneje las extensiones del tipo de dato. Una opción es utilizar el esquema de casos por default descripto en los capítulos anteriores. Para ello debemos encapsular la función foldTerm() en una clase, para luego poder extenderla utilizando herencia:

```
class foldClass[B]{
 def foldTerm[B](f : Int => B)(g: B => B => B)(t:Term):B = t match {
  case Num(n) => f(n)
  case Plus(l,r) => g(foldTerm(f)(g)(l))(foldTerm(f)(g)(r))
 }
}

val foldObject = new foldClass[Int]
```

Ahora podemos definir la nueva función eval() de la siguiente manera:

```
def eval = foldObject.foldTerm(id)(sum) _
```

Al extender el tipo algebraico, agregando el caso Mul debemos extender también la clase foldClass, para que maneje el nuevo caso. En el método de la clase extendida agregamos también un caso por default, para reutilizar el código que manejaba los casos anteriores:

```
case class Mul(left: Term, right: Term) extends Term

class foldExtended[B] extends foldClass[B] {
 def foldTerm[B](f : Int => B)(g: B => B => B)(h: B => B => B)(t:Term): B =
 t match {
  case Mul(l,r) => h (foldTerm(f)(g)(h)(l)) (foldTerm(f)(g)(h)(r))
  case default => super.foldTerm(f)(g)(default)
 }
}
```

```
val foldObjectExt = new foldExtended[Int]

val mul = (x:Int) => (y:Int) => x*y
```

Definimos la función evaluadora extendida utilizando el nuevo método foldTerm():

```
def evalExt = foldObjectExt.foldTerm(id)(sum)(mul) _
```

Así, con este esquema podemos reutilizar las funciones con las que se parametriza la función foldTerm() y todo el código que maneja los casos originales del tipo, pero tenemos que declarar la extensión de la función evaluadora con un nuevo *fold* distinto al anterior. El problema con este esquema es que torna la función foldTerm() más compleja y hace que su código esté separado en dos clases distintas. Esto hace que las funciones definidas con *fold*, en vez de ser mas claras, pueden ser mas complejas de entender.
En la siguiente sección veremos como dar una única definición del *fold*, que sirva para todas las extensiones que hagamos del tipo.

5.3. Definición del *Fold* utilizando Functores

Sería bueno poder declarar un *fold* genérico que sirva para definir las funciones originales y las funciones extendidas, es decir, declarar un *fold* cuya definición no cambie al extender el tipo de datos sino que permanezca estática. Para generalizar el *fold* primero debemos abstraer, para ello vamos a seguir un enfoque basado en [16]. Vamos a redefinir el tipo Term. Primero vamos a generalizarlo, parametrizándolo con un tipo. Luego definimos un *punto fijo* para los constructores:

```
abstract class Term[A]

case class Num[A](value: Int) extends Term[A]
case class Plus[A](left: A, right:A) extends Term[A]

case class Fix[F[_]](in: F[Fix[F]])
```

El tipo Fix está paramertizado con otro constructor de tipo, que a su vez es paramétrico. Se puede ver al tipo Fix como un constructor que toma como parámetro un constructor polimórfico de tipos. El constructor F está parametrizado con un tipo que corresponde al de las expresiones de los constructores internos [16].
También vamos a definir un sinónimo de tipo y algunas funciones útiles:

```
type TermInt = Fix[Term]
def num = (n:Int) => Fix[Term](Num(n))
def plus = (l:TermInt, r:TermInt) => Fix[Term](Plus(l,r))
def extract(t:Fix[Term]) = t match { case Fix(in) => in }
```

Las funciones num () y plus() sirven para construir expresiones de tipo Fix de manera mas sencilla, extract() sirve para acceder al constructor que parametriza a Fix. Así, podemos crear expresiones del tipo:

```
val n=plus(num(3),num(4))
```

Y podemos definir funciones sobre el tipo que utilizan pattern matching:

```
def caso(t: TermInt) = extract(t) match {
 case Num(n) => println("Es Num!")
 case Plus(l,r) => println("Es Plus!")
}
```

En su definición, el constructor Num no utiliza su parámetro de tipo, ya que no contiene subexpresiones, sólo contiene un número entero [16].
Con todas las definiciones anteriores, surge la pregunta ¿Para qué redefinir nuestro tipo Term así? Para poder declarar los constructores como *functores* y así generalizar el *fold*. Un constructor de tipos F es un functor si posee una función *fmap* con signatura de tipo:

```
(A => B) => F[A] => F[B]
```

Y verifica las propiedades:

```
fmap(id) = id
fmap(f) compose fmap(g) = fmap(f compose g )
```

Un *Functor* puede pensarse como una estructura que puede ser mapeada. Si tengo una estructura de tipo *F[A]* y una función A => B, *fmap* me devuelve otra estructura de tipo *F[B]*. La función *fmap* es una generalización de la función *map* para listas.
En Haskell existe una clase de tipos que representa a los functores, la misma está definida de la siguiente manera:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Debemos instanciar los constructores de un tipo algebraico, definiendo la función *fmap* para cada uno de ellos. En Scala no existen las clases de tipos como en Haskell, tales requerimientos podemos expresarlos como un *trait* que declara la función *fmap*, haciendo que los constructores del tipo implementen ese *trait*. Esta codificación tiene un estilo más orientado a objetos.
Definimos el *trait*:

```
trait  Functor[F[_]]  {
 def fmap[A, B](f: A => B)(r: F[A]): F[B]
}
```

Y hacemos que nuestro tipo Term implemente el trait Functor, es decir, cada constructor del tipo Term va a tener que implementar un método fmap, el código nos queda:

```
abstract class Term[A] extends Functor[Term]

case class Num[A](value: Int) extends Term[A] {
   override def fmap[A, B](f: A => B)(t: Term[A]): Num[B] = t match {
  case Num(n) => Num(n)
 }
}

case class Plus[A](left: A, right:A) extends Term[A] {
   override def fmap[A , B](f: A => B)(t: Term[A]): Plus[B] = t match {
  case Plus(l,r) => Plus(f(l), f(r))
 }
}
```

Ahora, con todas las piezas en su lugar, podemos definir la función *fold* genérica para nuestro tipo Term:

```
def foldTerm[A](f: Term[A] => A)(t: Fix[Term]): A = t match {
 case (Fix(in)) => f(in.fmap(foldTerm[A](f))(in))
}
```

En la nueva definición de foldTerm no se utiliza pattern matching sobre los constructores, la función está definida sobre Term, cuyos constructores están definidos como *Functores*.
Con foldTerm podemos definir las funciones sobre el tipo, para lo cual encapsulamos las funciones que parametrizarán a foldTerm en una clase, para hacerlas extensibles:

```
class foldParam {
 def evalParam(t: Term[Int]):Int = t match {
  case Num(n) => n
  case Plus(l,r) => l + r
 }
}

val param = new foldParam

def eval = foldTerm(param.evalParam) _
```

Ahora si queremos extender el tipo algebraico, agregamos el nuevo caso, definiendo el método *fmap* y extendemos las funciones que parametrizan a foldTerm como sigue:

```
case class Mul[A](left: A, right:A) extends Term[A] {
```

```
  override def fmap[A , B](f: A => B)(t: Term[A]): Mul[B] = t match {
 case Mul(l,r) => Mul(f(l), f(r))
 }
}

def mul = (l:TermInt, r:TermInt) => Fix[Term](Mul(l,r))

class foldParamExtended1 extends foldParam{
 override def evalParam(t: Term[Int]): Int = t match {
  case Mul(l,r) => l * r
  case default => super.evalParam(default)
 }
}
val paramExtended = new foldParamExtended1
def evalExt = foldTerm(paramExtended.evalParam) _
```

De esta manera reutilizamos el código de las funciones que parametrizan a foldTerm() y la función evalExt() utiliza la misma función foldTerm() que se utilizó para definir eval().
Con esta nueva definición del tipo Term utilizando un punto fijo, también podemos definir y extender funciones sin utilizar foldTerm() siguiendo el esquema mostrado en el capítulo 3, por ejemplo definimos la función termtoString() sin utilizar *fold* y la extendemos:

```
class Op {
 def termtoString(t: TermInt): String = extract(t) match {
  case Num(n) => n.toString
  case Plus(l,r) => "("+termtoString(l)+"+"+termtoString(r)+")"
 }
}
```

La extensión para manejar el caso Mul:

```
class OpExtended extends Op {
 override def termtoString(t: TermInt): String = extract(t) match {
  case Mul(l,r) => "("+termtoString(l)+"*"+termtoString(r)+")"
  case default => super.termtoString(t)
 }
}
```

Aquella persona con una fuerte influencia de la programación orientada a objetos argumentaría que la codificación de la función foldTerm() es poco elegante y redundante. El "problema" está en el llamado in.fmap(in, foldTerm[A](f)), fmap es una función embebida (método) del objeto in, que se le pasa como parámetro el mismo objeto in. Este pasaje no es necesario, se hizo de esta manera para que fmap tenga una signatura de tipos similar a la que se define en la clase de tipos *Functor* de Haskell. Siendo Scala un lenguaje que también es orientado a objetos, podemos acercar la

definición de fmap a ese paradigma para evitar ese pasaje de parámetros, el código quedaría:

```
trait Functor[A,F[_]] {
 this: F[A] =>
 def fmaps[B](f: A => B)(r: F[A]): F[B]
 final def fmap[B](f: A => B):F[B] = fmaps(f)(this)
}

abstract class Term[A] extends Functor[A, Term]

case class Num[A](value: Int) extends Term[A] {
   override def fmaps[B](f: A => B)(t: Term[A]): Num[B] = t match {
  case Num(n) => Num(n)
 }
}

case class Plus[A](left: A, right:A) extends Term[A] {
   override def fmaps[B](f: A => B)(t: Term[A]): Plus[B] = t match {
  case Plus(l,r) => Plus(f(l), f(r))
 }
}
```

Y la definición de foldTerm se compactaría un poco:

```
def foldTerm[A](f: Term[A] => A)(t: Fix[Term]): A = t match {
 case (Fix(in)) => f(in.fmap(foldTerm[A](f)))
}
```

5.4. *Fold* para Tipos de Datos Abiertos

En el capítulo 4 vimos como podíamos implementar los tipos de datos y las funciones abiertas utilizando tipos de datos extensibles. En esta sección vamos a describir como se puede definir el *fold* para los tipos de datos abiertos. El esquema es similar al de la sección anterior, vamos a utilizar *Functores*; la definición del tipo y de la función foldTerm es exactamente igual:

```
trait Functor[F[_]] {
 def fmap[A, B](f: A => B)(r: F[A]): F[B]
}

abstract class Term[A] extends Functor[Term]

case class Num[A](value: Int) extends Term[A] {
   override def fmap[A, B](f: A => B)(t: Term[A]): Num[B] = t match {
  case Num(n) => Num(n)
```

```
 }
}

case class Plus[A](left: A, right:A) extends Term[A] {
   override def fmap[A , B](f: A => B)(t: Term[A]): Plus[B] = t match {
  case Plus(l,r) => Plus(f(l), f(r))
 }
}

case class Fix[F[_]](in: F[Fix[F]])

type TermInt = Fix[Term]
def num = (n:Int) => Fix[Term](Num(n))
def plus = (l:TermInt, r:TermInt) => Fix[Term](Plus(l,r))

def foldTerm[A](f: Term[A] => A)(t: Fix[Term]): A = t match {
 case (Fix(in)) => f(in.fmap(foldTerm[A](f))(in))
}
```

También definimos las funciones que parametrizarán a foldTerm en una clase para hacerlas extensibles:

```
class classParam {
 def evalParam(t: Term[Int]):Int = t match {
  case Num(n) => n
  case Plus(l,r) => l + r
 }
 def toStringParam(t: Term[String]):String = t match {
  case Num(n) => n.toString()
  case Plus(l,r) => "("+l+"+"+r+")"
 }
}
```

Ahora tenemos que definir las funciones abiertas sobre el tipo utilizando foldTerm. Para ellos definimos las funciones para que tomen un valor explícito de tipo TermInt y otro implícito de tipo classParam, el valor implícito es un objeto que contiene las funciones abiertas que parametrizan a foldTerm, esas funciones las pasamos explícitamente. Tenemos que crear también una instancia implícita de classParam para que el compilador pueda pasarla como parámetro:

```
implicit val op = new classParam

def eval(t: TermInt)(implicit op: classParam ) = foldTerm(op.evalParam)(t)
def toString(t: TermInt)(implicit op: classParam ) = foldTerm(op.toStringParam)(t)
```

Para adaptar eval() y toString() a las extensiones solamente debemos definir nuevas funciones para parametrizar a foldTerm, es decir, sólo debemos agregar ecuaciones

a las funciones que se pasan como parámetro, ya que foldTerm es genérico. Al extender declaramos el nuevo caso del tipo y "agregamos ecuaciones" a las funciones que parametrizan a foldTerm para que manejen el nuevo caso:

```
case class Mul[A](left: A, right:A) extends Term[A] {
   override def fmap[A , B](f: A => B)(t: Term[A]): Mul[B] = t match {
  case Mul(l,r) => Mul(f(l), f(r))
 }
}

def mul = (l:TermInt, r:TermInt) => Fix[Term](Mul(l,r))

class classParamExtended extends classParam {
 override def evalParam(t: Term[Int]):Int = t match {
  case Mul(l,r) => l * r
  case default => super.evalParam(default)
 }
 override def toStringParam(t: Term[String]):String = t match {
  case Mul(l,r) => "("+l+"*"+r+")"
  case default => super.toStringParam(default)
 }
}

val m = plus(mul(num(2), num(2)), plus(mul(num(3), num(4)), num(1)))

implicit val op = new classParamExtended()
```

Sólo tuvimos que extender las funciones que parametrizan al fold y ya podemos utilizar la función eval() con la extensión del tipo: println(eval(m)).
En este capítulo se dió una versión de la función *fold* extensible compatible con los tipos algebraicos extensibles con defaults y otra versión compatible con los tipos de datos abiertos. Ambas versiones están basadas en el trabajo presentado en [16]. Se mostró que el esquema presentado en [16] puede utilizarse junto con las soluciones presentadas en los capítulos 3 y 4 de este trabajo.

Capítulo 6

Conclusiones

En este trabajo se presentaron dos soluciones al *expression problem* en el lenguaje de programación Scala.
Scala es un lenguaje que combina y unifica conceptos de los paradigmas de programación funcional y orientado a objetos. Se puede decir que Scala es un lenguaje orientado a objetos funcional, que ofrece lo mejor de ambos mundos. Sin embargo tiende a ser visto como un lenguaje orientado a objetos con algunas características funcionales y es utilizado para codificar programas en un estilo orientado a objetos. En este trabajo se utilizó una visión diferente, considerándolo un lenguaje funcional con características orientadas a objetos.
Se presentaron los tipos de datos algebraicos extensibles con defaults. En esta implementación se utilizaron los tipos algebraicos de datos que soporta Scala, vía *case clases*, y funciones de alto orden, herencia de clases, *traits* y *mixin composition*. En la propuesta original hecha por Zenger y Odersky [19] sólo era posible realizar extensiones lineales del tipo. Con la implementación en Scala se mostró que pueden realizarse tanto extensiones lineales como así también extensiones independientes, estas últimas pudiendo ser combinadas para utilizarse juntas. Para esta implementación son fundamentales los conceptos de herencia, *traits* y *mixin composition* que soporta el lenguaje. La codificación presentada sigue un diseño funcional, donde se separa el tipo de datos de las operaciones y todas las operaciones se codifican utilizando *pattern matching*. Es posible extender el tipo y las operaciones sin modificar ni recompilar el código existente. Se presentaron dos formas de extender las operaciones sobre el tipo de datos. Una utiliza funciones de alto orden, donde cada operación se parametriza con una función que será la encargada de manejar los casos por default. La otra encapsula las funciones sobre el tipo en métodos de una clase o trait y utiliza herencia de clases para extender la funcionalidad de los métodos y así agregar código que maneja los nuevos casos del tipo. La segunda forma resulta mas directa, ya que no es necesario parametrizar los métodos como se hace en la primera. En ambas formas, las operaciones extendidas reutilizan todo el código de las operaciones originales, solo agregando el código necesario para manejar las extensiones del tipo. También se mostró que utilizando la combinación de paradigmas del lenguaje

y los *parámetros implícitos* soportados por Scala, es posible implementar los tipos de datos abiertos y las funciones abiertas, que originalmente fueron propuestas como una extensión de Haskell por Löh y Hinze [7]. Para ambas propuestas se mostró como pueden extenderse tanto el tipo de datos como las operaciones sobre él, dando así dos soluciones al *expression problem*. En las soluciones presentadas se resuelven los tipos estáticamente, sin utilizar *casts* y son lo suficientemente simples y directas que no requieren asistencia del compilador.
Por último se mostró como combinar la soluciones propuestas en este trabajo con la codificación del *fold* extensible presentado en [16]. Esto hace posible definir una función *fold* extensible junto a los tipos de datos extensibles con defaults y los tipos de datos abiertos, en sintonía con el estilo funcional de las soluciones presentadas. Una contra que se presenta en estas soluciones es que, al agregar casos al tipo de datos utilizando herencia de clases, el compilador no puede chequear si el *pattern matching* es exhaustivo en la definición de una función, entonces se puede pasar un valor para el cual no exista un *pattern*, y eso dará un error en tiempo de ejecución.

Apéndice A

El Lenguaje de Programación Scala

En esta sección presentaremos brevemente las principales características del lenguaje de programación Scala.
Scala fué diseñado por Martin Odersky, quién también trabajó junto a Philip Wadler en *Generic Java*, un superconjunto de Java que agrega soporte para la programación genérica. Generic Java fue incorporado al lenguaje Java a partir de la versión J2SE 5.0. Odersky también construyó la versión actual del compilador de Java (javac).
Scala combina y unifica la programación funcional y la programación orientada a objetos, ya que ambas tienen características complementarias: la programación funcional hace que sea fácil construir cosas complejas a partir de partes simples, usando funciones de alto orden, tipos algebraicos, pattern matching y polimorfismo paramétrico. La programación orientada a objetos hace que sea fácil adaptar y extender sistemas complejos, utilizando subtipado y herencia, configuraciones dinámicas y clases como abstracciones parciales.
Como ya vimos, en Scala podemos definir funciones usando la palabra clave **def**:

```
def square(x:Int) = x*x
```

Scala distingue entre *definiciones* y *valores*. En una definición **def** x = e la expresión e no será evaluada hasta que x sea usada, y será evaluada en cada uso (by-name). Para definir un valor se utiliza la palabra clave **val**:

```
val y = 2
val z = square(y)
```

En una definición con **val**, el lado derecho se evalúa en el momento de la definición, luego el nombre refiere a ese valor evaluado (by-value). Scala no hace distinción alguna entre declaraciones y expresiones, mas aún, todas las declaraciones son expresiones que se evalúan a algún valor. Scala utiliza la evaluación estricta por default, pero también soporta evaluación *lazy* en la definición de valores:

```
lazy val = square(y)
```

También pueden declararse variables, que admiten asignación:

```
var n:Int = 0

n = 1
```

por lo tanto, el lenguaje no fuerza la transparencia referencial y la programación funcional es una disciplina en Scala.
Para los argumentos de una función, Scala utiliza normalmente *call-by-value*, pero si el tipo del parámetro de la función comienza con => , utiliza *call-by-name*:

```
def square(x: =>Int) = x*x
```

Al ser un lenguaje orientados a objetos, en Scala se pueden declarar clases e instanciarlas:

```
class Persona
val p1 = new persona
val p2 = new persona()
```

Si un método no toma parámetros, los paréntesis son opcionales en su llamada. La declaración de la clase Persona anterior corresponde a la declaración en Java:

```
public class Persona{
}
```

También se pueden declarar objetos directamente, sin definir una clase para el mismo, y así implementar el patrón de diseño *singleton*:

```
object Simple {
 def metodoSimple() = "Soy un objeto"
}
```

Y podemos extender clases y objetos:

```
object Presidente extends Persona
class Mujer extends Persona {
 def m = "soy mujer"
}

class Chica extends Mujer {
 override def m = "soy una chica"
}
```

En Scala todo valor es un objeto y toda operación es una llamada a método. Es decir, en Scala no existen los tipos de datos primitivos como en Java (int, boolean, etc) y no existen operadores, sino llamadas a métodos de una clase. En Scala, el tipo Int es una clase en la que se definen métodos, por ejemplo, las siguientes expresiones son equivalentes en Scala:

```
1+1
1.+(1)
```

es decir, para sumar dos enteros, se llama al método + del objeto 1, de tipo Int. Scala soporta los tipos de datos algebraicos, a través de las *case classes*, y el *pattern matching*, por ejemplo, definimos los árboles binarios y funciones sobre ellos:

```
abstract class BinTree
case object EmptyTree extends BinTree
case class Node(elem: Int, left: BinTree, right: BinTree) extends BinTree

def inOrder(t: BinTree): List[Int] = t match {
 case EmptyTree => List()
 case Node(n,l,r) => inOrder(l):::List(n):::inOrder(r)
}

def depth(t: BinTree): Int = t match {
 case EmptyTree => 0
 case Node(_,EmptyTree,r) => 1 + depth(r)
 case Node(_,l,EmptyTree) => 1 + depth(l)
 case Node(_,l,r) => scala.math.max(depth(l),depth(r)) + 1
}

var t1 = EmptyTree
var t2 = Node(4,Node(7,EmptyTree,EmptyTree),Node(1,EmptyTree,EmptyTree))
```

En Scala se pueden definir estructuras de datos mutables como en Java y C++ como así también estructuras de datos persistentes inmutables. Un ejemplo de éstas últimas son la listas:

```
List(1,2,3)
```

representa una lista de enteros con los elementos 1, 2 y 3, equivalentemente se puede definir utilizando los constructores del tipo:

```
1::2::3::Nil
```

y se puede operar sobre ellas, por ejemplo con las funciones map y filter:

```
List(1,2,3,4).filter(x => x % 2 == 0)
```

res:List[Int] = List(2, 4)

```
List(1,2,3,4).map(x => x*2 )
```

res1: List[Int] = List(2, 4, 6, 8)

También podemos definir funciones sobre listas utilizando *pattern matching*:

```
def qsort(l: List[Int]): List[Int] = {
```

```
  l match {
    case Nil        => Nil
    case pivot::tail =>
      qsort(tail.filter(_ < pivot)) ::: pivot :: qsort(tail.filter(_ >= pivot))
  }
}
qsort(List(2,7,4,9,10,5,1))
```

res: List[Int] = List(1, 2, 4, 5, 7, 9, 10)

Al igual que Haskell y ML, Scala soporta *polimorfismo paramétrico*, funciones de *alto orden* y *currificación*. Por ejemplo, la composición de funciones puede definirse como sigue:

```
def comp [A, B, C] (f : B => C) (g : A => B) (x : A) : C = f (g (x))
```

Scala también ofrece valores opcionales para manejar errores. Un valor de tipo Option[T] puede tener dos formas: None o Some(n), donde n es un valor de tipo T. Así una función para dividir dos enteros puede manejar el caso de la división por cero de la siguiente manera:

```
def div(a:Int, b:Int) : Option[Int] = if(b==0) None else Some(a/b)
```

Y podemos hacer pattern matching sobre el resultado de esa función:

```
div(4,0) match {
 case Some(x) => println(x)
 case None => println("Division por cero!")
}
```

En lugar de interfaces, Scala ofrece el concepto mas general de *traits*. Al igual que las interfaces, los *traits* pueden ser usados para definir métodos abstractos, es decir, su signatura, sin implementación concreta. Pero a diferencia de las interfaces, también permiten la definición de métodos concretos. Los *traits* pueden ser combinados (mezclados) con una clase, usando *mixin composition* y una clase puede ser mezclada con muchos *traits* haciendo posible un forma de herencia múltiple segura. Un *trait* que define sólo métodos concretos, puede ser combinado con una clase u objeto, para especializarlo, agregando el código necesario de la especialización.

```
trait Boolean {
 def ifThenElse[T](t: => T, e: => T): T
 def && (x: => Boolean): Boolean = ifThenElse(x, False)
 def || (x: => Boolean): Boolean = ifThenElse(True, x)
}

trait Printable {
 def print : String = "Soy un Boolean"
}
```

```
object True extends Boolean with Printable {
 def ifThenElse[T](t: => T, e: => T) = t
 override def print = "True"
}

object False extends Boolean with Printable {
 def ifThenElse[T](t: => T, e: => T) = e
 override def print = "False"
}
```

Apéndice B

Demostración de las propiedades de los Functores

Vamos a demostrar que los constructores del tipo Term son functores. Tenemos que demostrar las siguientes propiedades para cada constructor del tipo:

(P1) fmap(id) = id
(P2) fmap(f) _ compose fmap(g) _ = fmap(f compose g)

Primero vamos a hacer unas observaciones. La función fmap es en realidad un método definido en el *trait* Functor[F[_]], que implementa cada clase constructora del tipo, entonces se llama a través de un objeto: Num(3).fmap(). Este método se usa en la definición del *fold*, en el llamado in.fmap(foldTerm[A](f))(in), donde in es un objeto del tipo. En el llamado, el mismo objeto es pasado como parámetro al método y sobre ese parámetro es donde se opera. Así, en vez de escribir Num(n).fmap() en nuestra demostración, vamos a escribir fmap() para simplificar la misma. Por otro lado, compose es un método que está definido en el *trait* Function1, es decir, puede aplicarse a funciones; como fmap es un método, el compilador de Scala aplica una *expansión-Eta* como se explicó anteriormente. La definición del método compose en el *trait* Function1 es:

```
trait Function1[−T1, +R] extends AnyRef { self =>

  def apply(v1: T1): R

  def compose[A](g: A => T1): A => R = { x => apply(g(x)) }
}
```

Es decir, es un método que toma una función g y devuelve una función que toma un x y aplica el método apply de la función desde la que se está llamando al resultado de g(x).
La función id puede definirse como:

```
def id[A](x: A) = x
```

Num es un Functor:

P1

```
fmap(id)(Num(n))
=                                          { definición de fmap para Num
Num(n)
```

Por otro lado:

```
id(Num(n))
=                                          { definición de id
Num(n)
```

Como ambas expresiones reducen al mismo valor, (P1) es verdadera para Num(n).

P2

```
(fmap(f) _ compose fmap(g)) (Num(n))
=                                          { definición de compose
(x=>apply(fmap(g)(x))) (Num(n))
=                                          { aplicación
apply(fmap(g)(Num(n)))
=                                          { definición de fmap para Num
apply(Num(n))
=                                          { apply de fmap(f) para Num
Num(n)
```

Por otro lado:

```
fmap(f _ compose g)(Num(n))
=                                          { definición de fmap para Num
Num(n)
```

Plus es un Functor:

P1

```
fmap(id)(Plus(l,r))
=                                          { definición de fmap para Plus
```

```
Plus(id(Plus(l,r)),id(Plus(l,r)))
=                                                    { definición de id
Plus(l,r)
```

Por otro lado:

```
id(Plus(l,r))
=                                                    { definición de fmap para Plus
Plus(l,r)
```

P2

```
(fmap(f) _ compose fmap(g)) (Plus(l,r))
=                                                    { definición de compose
(x=>apply(fmap(g)(x))) Plus(l,r)
=                                                    { aplicación
apply(fmap(g)(Plus(l,r)))
=                                                    { definición de fmap para Plus
apply(Plus(g(l),g(r)))
=                                                    { apply de fmap(f) para Plus
Plus(f(g(l)),f(g(r)))
```

Por otro lado:

```
fmap(f _ compose g)(Plus(l,r))
=                                                    { definición de fmap para Plus
Plus((f _ compose g)(l),(f _ compose g)(r))
=                                                    { definición de compose
Plus((x=>apply(g(x)))(l),(x=>apply(g(x)))(r))
=                                                    { aplicación
Plus(apply(g(l)),apply(g(r)))
=                                                    { apply de fmap(f) para Plus
Plus(f(g(l)),f(g(r)))
```

Mul es un Functor:

P1

```
fmap(id)(Mul(l,r))
=                                                    { definición de fmap para Mul
Mul(id(Mul(l,r)),id(Mul(l,r)))
```

```
=                                                    { definición de id
Mul(l,r)
```

Por otro lado:

```
id(Mul(l,r))
=                                                    { definición de fmap para Mul
Mul(l,r)
```

P2

```
(fmap(f) _ compose fmap(g)) (Mul(l,r))
=                                                    { definición de compose
(x=>apply(fmap(g)(x))) Mul(l,r)
=                                                    { aplicación
apply(fmap(g)(Mul(l,r)))
=                                                    { definición de fmap para Mul
apply(Mul(g(l),g(r)))
=                                                    { apply de fmap(f) para Mul
Mul(f(g(l)),f(g(r)))
```

Por otro lado:

```
fmap(f _ compose g)(Mul(l,r))
=                                                    { definición de fmap para Mul
Mul((f _ compose g)(l),(f _ compose g)(r))
=                                                    { definición de compose
Mul((x=>apply(g(x)))(l),(x=>apply(g(x)))(r))
=                                                    { aplicación
Mul(apply(g(l)),apply(g(r)))
=                                                    { apply de fmap(f) para Mul
Mul(f(g(l)),f(g(r)))
```

Apéndice C

Organización del Código

En Scala hay varias formas de ejecutar un programa. Se puede ejecutar interactivamente por la linea de comando, insertando línea por línea el código en el intérprete de Scala. Se puede correr un script que consiste en un único archivo, pasándolo al intérprete (scala). También se puede utilizar el compilador de Scala (scalac) para generar archivos *class* y ejecutarlos con la máquina virtual de Java. El compilador de Scala requiere que el archivo fuente contenga una o más definiciones de clases, traits u objetos. Los paquetes (*package*) son usados para agrupar clases, objetos y traits y así separar el código del programa en diferentes archivos. Existen dos formas de indicar que el código de un determinado archivo pertenece a un determinado paquete. Podemos introducir como primer línea de código en el archivo la declaración del paquete:

```
package A;
//codigo del paquete
```

que indica que el código siguiente pertenece al paquete *A*. También podemos definir el nombre del paquete y el código que pertenece al mismo entre llaves:

```
package A {
 //codigo del paquete
}
```

La primer forma de definir un paquete es en realidad *syntactic sugar* para la segunda, que es mas general. Es posible definir incluso paquetes anidados:

```
package A {
 //codigo del paquete A
 package B {
  //codigo del paquete B
 }
}
```

Scala nos permite también definir varios paquetes en el mismo archivo fuente. Cuando se compila el código, el compilador separa los archivos *.class* generados en distintos

directorios, siguiendo la estructura de paquetes declarados. Así podemos declarar en un mismo archivo:

```
package A {
 //codigo del paquete A
}
package B {
 //codigo del paquete B
}
```

El mismo resultado se obtiene si se declaran los paquetes A y B en distintos archivos y se compilan por separado.
En Scala un paquete es un objeto especial que define un conjunto clases, objetos y paquetes, todos miembros del paquete. Una definición **package** p {ds} agrega todas las definiciones en ds como miembros del paquete p. Los miembro de un paquete se llaman definiciones de primer nivel (*top-level definitions*). Si una definición en ds es etiquetada como privada, será visible sólo para otros miembros del paquete [9]. De esta manera Scala unifica los módulos con los objetos [8].
Como se mencionó anteriormente, en un paquete pueden definirse clases, objetos o traits, pero la programación funcional no utiliza ninguna de esas construcciones, salvo por las *case classes* en Scala ¿Cómo podemos estructurar código funcional con paquetes? Mas aún, en el capítulo 3 definimos las funciones sobre el tipo algebraico con **def**, pero esas definiciones no podemos empaquetarlas con *package* si no están dentro de una clase. Para incluir estas definiciones en un paquete Scala posee *package object*. Cualquier tipo de definición que se puede poner dentro de una clase, también puede ponerse dentro de un *package object*. Cada *package* puede tener un único *package object* asociado y cualquier definición dentro del *package object* es considerada miembro del *package*. Para declarar un paquete y su *package object* asociado en un mismo archivo podemos declarar:

```
package A {
 //codigo del paquete
}
package object A {
 //otras definiciones de A
}
```

Y si en otro archivo se hace un **import** A, también se importarán todas las definiciones del *package object* A.
En este trabajo presentamos el código de ejemplo desestructurado, para que pueda probarse con el intérprete de Scala, ya sea introduciéndolo línea por línea por la línea de comandos, o agrupándolo en un script y llamando al intérprete sobre ese script. Ahora vamos a mostrar cómo puede organizarse el código para que pueda ser compilado, para ello vamos a mostrar algunos ejemplos presentados anteriormente. El código de las extensiones independientes utilizando funciones de alto orden puede estructurarse en paquetes, utilizando un único archivo fuente, de la siguiente manera:

```
package Base {

 abstract class Term

 case class Num(value: Int) extends Term
 case class Plus(left: Term, right: Term) extends Term

}

package object Base{
 def eval(f: Term => Int)(t:Term): Int = t match {
  case Num(v) => v
  case Plus(l,r) => eval(f)(l) + eval(f)(r)
  case default => f(default)
 }
 def termtoString(f: Term => String)(t: Term):String= t match {
  case Num(v) => v.toString
  case Plus(l,r) => "("+termtoString(f)(l)+"+"+termtoString(f)(r)+")"
  case default => f(default)
 }
}

//----- El siguiente codigo pertenece a otro paquete

package Extended1 {
 import Base._

 abstract class ExtendedTerm1 extends Term

 case class Mul(left: Term, right: Term) extends ExtendedTerm1
}

package object Extended1{
 import Base._

 def evalExtended1(f: Term => Int)(t: Term): Int = t match{
  case Mul(l,r) => evalExtended1(f)(l) * evalExtended1(f)(r)
  case default => f(default)
 }
 def termtoStringExtended1(f: Term => String)(t: Term): String = t match{
  case Mul(l,r) => "("+termtoStringExtended1(f)(l)+ "*"
      +termtoStringExtended1(f)(r)+")"
  case default => f(default)
 }

}

//----- El siguiente codigo pertenece a otro paquete
```

```
package Extended2 {
 import Base._
 import Extended1._

 abstract class ExtendedTerm2 extends ExtendedTerm1

 case class Diff(left: Term, right: Term) extends ExtendedTerm2
}

package object Extended2{
 import Base._
 import Extended1._

 def evalExtended2(f: Term => Int)(t: Term): Int = t match{
  case Diff(l,r) => evalExtended2(f)(l) - evalExtended2(f)(r)
  case default => f(default)
 }

 def termtoStringExtended2(f: Term => String)(t: Term): String = t match{
  case Diff(l,r) => "("+termtoStringExtended2(f)(l)+ "-"
       +termtoStringExtended2(f)(r)+")"
  case default => f(default)
 }

}

//----- Declaramos un objeto con un metodo main para que pueda ejecutarse

object TestLineal{
 import Base._
 import Extended1._
 import Extended2._
    def main(args: Array[String]) {
  val term1 = Plus(Num(2), Num(3))
  val term2 = Plus(Diff(Num(3), Num(2)), Mul(Plus(Num(1), Num(0)),  Num(1)))

  def evalGeneral(t:Term):Int = eval(evalExtended1(evalExtended2(evalGeneral)))(t)

  def toStringGeneral(t:Term): String =
   termtoString(termtoStringExtended1(termtoStringExtended2(toStringGeneral)))(t)

  println("Eval: "+evalGeneral(term2))
  println("toStringGeneral: "+toStringGeneral(term2))

  def enlacePorTipos(t:Term): String = t match{
   case(t: ExtendedTerm2) => termtoStringExtended2(enlacePorTipos)(t)
   case(t: ExtendedTerm1) => termtoStringExtended1(enlacePorTipos)(t)
   case(t: Term) => termtoString(enlacePorTipos)(t)
  }
```

```
  println("Enlace por Tipos: ")
  println(enlacePorTipos(term2))

  def simplificar(f: Term => Term)(t:Term): Term = t match{
   case Mul(l, r) => (l,r) match {
    case (Num(0), _) => Num(0)
    case (Num(1), _) => r
    case (_, Num(0)) => Num(0)
    case (_, Num(1)) => l
    case _ => Mul(simplificar(f)(l), simplificar(f)(r))
   }
   case Plus(l,r) => Plus(simplificar(f)(l), simplificar(f)(r))
   case Diff(l,r) => Diff(simplificar(f)(l), simplificar(f)(r))
   case Num(n) => Num(n)
   case default => f(default)
  }
  println("Despues de simplificar term2: ")
  println(enlacePorTipos(simplificar(t => throw new Exception())(term2)))
    }
}
```

Bibliografía

[1] R. Bird. *Introduction to Functional Programming Using Haskell*. Prentice-Hall, 1998.

[2] B. Emir, M. Odersky, and J. Williams. Matching objects with patterns. In *Proceedings of European Conference on Object-Oriented Programming*, 2007.

[3] E. Gamma, R. Helm, R. Jhonson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[4] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. *The Java Language Specification*. Oracle America Inc., 2013.

[5] S. Krishnamurthi, M. Felleisen, and D. P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *European Conference on Object-Oriented Programming,*, Brussels, Belgium, 1998.

[6] C. K. K. Loverdos and A. Syropoulos. *Steps in Scala: An Introduction to Object-Functional Programming*. Cambridge University Press, 2010.

[7] A. Löh and R. Hinze. Open data types and open functions. In *Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*, 2006.

[8] M. Odersky. The scala experiment – can we provide better language support for component systems? Technical report, Google TechTalk, 2006.

[9] M. Odersky. *The Scala Language Specification*. Programming Methods Laboratory, EPFL, 2011. Version 2.9.

[10] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[11] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima, 2008.

[12] M. Odersky and P. Wadler. Pizza into java: Translating theory into practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, page pages 146–159, 1997.

[13] M. Odersky and M. Zenger. Scalable component abstractions. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2005.

[14] B. Oliveira and J. Gibbons. Scala for generic programmers. In R. Hinze, editor, *Workshop on Generic Programming*, Victoria, BC, Sep 2008.

[15] B. C. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2010.

[16] W. Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, July 2008.

[17] M. Torgersen. The expression problem revisited - four new solutions using generics. In *Proceedings of the 18th European Conference on Object-Oriented Programming*, Nicosia, Cyprus, March 2004.

[18] P. Wadler. The expression problem. Posted on the Java Genericity mailing list, June 1998.

[19] M. Zenger and M. Odersky. Extensible algebraic data types with defaults. In *Proceedings of the Internacional Conference on Functional Programming*, Firenze, Italy, September 2001.