IMPLEMENTACIÓN DE UN MODELO DE PROGRAMACIÓN ORIENTADO A ASPECTOS BAJO *ARROWS* DE HASKELL

MARÍA EUGENIA MARÍN MORA

UNIVERSIDAD EAFIT
DEPARTAMENTO DE INGENIERÍA DE SISTEMAS
MEDELLÍN
2010

IMPLEMENTACIÓN DE UN MODELO DE PROGRAMACIÓN ORIENTADO A ASPECTOS BAJO *ARROWS* DE HASKELL

MARÍA EUGENIA MARÍN MORA

Trabajo de grado presentado como requisito parcial para optar al título de Ingeniera de Sistemas

Asesor: JUAN FRANCISCO CARDONA MC CORMICK

UNIVERSIDAD EAFIT
DEPARTAMENTO DE INGENIERÍA DE SISTEMAS
MEDELLÍN
2010

AGRADECIMIENTOS

A Juan Francisco Cardona Mc Cormick, Magister en Informática y asesor del proyecto, por su entusiasmo y dedicación con este trabajo.

CONTENIDO

INTRODUCCIÓN **11**

1. OBJETIVOS **13**

1.2 OBJETIVO GENERAL 13

1.3 OBJETIVOS ESPECÍFICOS 13

2. PROGRAMACIÓN FUNCIONAL **14**

2.1 LENGUAJES FUNCIONALES PUROS 16

2.1.1 Transparencia referencial 17

2.1.2 Noción de estado 17

2.1.3 Funciones de orden superior 17

2.1.4 Recursión 18

2.1.5 Evaluación perezosa 18

2.2 LENGUAJES FUNCIONALES IMPUROS 18

2.3 LENGUAJES FUNCIONALES EN LA INDUSTRIA 19

3. EL LENGUAJE FUNCIONAL HASKELL **20**

3.1 FUNCIONES 21

3.1.1 Definición 21

3.1.2 Comparación de patrones 22

3.1.3 Expresiones lambda 22

3.1.4 Currificación 23

3.2 SISTEMA DE TIPOS 24

3.3 POLIMORFISMO 25

3.4 TIPOS DE DATOS ALGEBRAICOS 27

3.4.1 Tipos enumerados 28

3.4.2 Productos 28

3.4.3 Uniones 28

3.4.4 Tipos recursivos 29
3.4.5 Tipos algebraicos polimórficos 29
3.4.6 Algunos tipos polimórficos predefinidos 30
3.5 SISTEMA DE CLASES 30
3.6 IMPLEMENTACIONES 34
3.6.1 Glasgow Haskell Compiler (GHC) 34
3.6.2 Hugs 34
3.6.3 nhc98 34
3.6.4 York Haskell Compiler (YHC) 34
3.6.5 Utrecht Haskell Compiler (UHC) 35

4. MÓNADAS 36
4.1 DEFINICIÓN 36
4.2 LEYES DE LAS MÓNADAS 38
4.3 CLASES MONÁDICAS 38
4.3.1 Clase `Functor` 38
4.3.2 Clase `Monad` 39
4.3.3 Clase `MonadPlus` 39
4.4 NOTACIÓN do 40
4.5 EJEMPLO 41

5. *ARROWS* 43
5.1 CLASES *ARROW* 44
5.1.1 *Arrows* y pares 44
5.1.2 *Arrows* y condicionales. 47
5.1.3 *Arrows* y realimentación. 50
5.1.4 Mónadas como *Arrows* 51
5.1.5 Funciones como *Arrows*. 52
5.2 NOTACIÓN 52
5.2.1 Abstracciones *Arrow* 52
5.2.2 Comando de aplicación 52

5.2.3 Comando condicional 53
5.2.4 Notación do 53
5.2.5 Combinadores de comandos 53

6. PROGRAMACIÓN ORIENTADA A ASPECTOS 55
6.1 CÓDIGO DISPERSO Y ENREDADO 56
6.2 ENFOQUE DE AOP 57
6.2.1 Aspectos y componentes 57
6.2.2 Beneficios 58
6.2.3 Estructura 58

7. TRABAJO RELACIONADO 61
7.1 AOP CON CLASES DE TIPOS 61
7.1.1 Enfoque 61
7.1.2 Limitaciones 63
7.2 GRAMÁTICAS DE ATRIBUTOS 64
7.3 RELACIÓN ENTRE MÓNADAS Y AOP 65

8. DISEÑO DEL MODELO 68
8.1 ASPECTOS Y COMPONENTES 68
8.2 PUNTOS DE UNIÓN 72
8.3 TEJIDO 73
8.3.1 Aspectos antes de componentes 75
8.3.2 Aspectos después de componentes 78
8.3.3 Aspectos en lugar de componentes 81
8. 4 EJEMPLO 83
8.4.1 Aspecto de seguridad 83
8.4.2 Aspecto de depuración 84
8.4.3 Aspecto de restricción de un valor 84
8.4.4 Aspecto de log 85
8.4.5 Combinación de aspectos y componentes 85

9. CONCLUSIONES **87**

BIBLIOGRAFÍA **89**

LISTA DE FIGURAS

Figura 1. Clases del Prelude 33
Figura 2. Combinador first 45
Figura 3. Combinador second 46
Figura 4. Combinador &&& 46
Figura 5. Combinador *** 47
Figura 6. Combinador left 48
Figura 7. Combinador right 49
Figura 8. Combinador ||| 49
Figura 9. Combinador +++ 50
Figura 10. Operador loop 51
Figura 11. *Arrow* del lenguaje y *Arrow* nombrado 69
Figura 12. Puntos de unión 73
Figura 13. Combinador #>| 75
Figura 14. Combinador #<| 76
Figura 15. Combinador ?>| 77
Figura 16. Combinador ?<| 78
Figura 17. Combinador |># 79
Figura 18. Combinador |<# 79
Figura 19. Combinador |>? 80
Figura 20. Combinador |<? 81
Figura 21. Combinador =># 82
Figura 22. Combinador #>= 82

RESUMEN

Este proyecto fue realizado con el objetivo de explorar el lenguaje funcional Haskell y los mecanismos de computación que éste proporciona, mediante la implementación de un modelo que simule Programación Orientada a Aspectos (AOP). El modelo presentado no implementa todas las características encontradas en AOP, sólo aquellas relacionadas con puntos de unión y algunos elementos para la elaboración del tejido, que es realizado de forma manual.

Para su desarrollo fue necesario el estudio y aprendizaje del lenguaje funcional Haskell y algunos de sus mecanismos más avanzados como *Arrows*, mónadas, y elementos del sistema de tipos que incluyen clases de tipos, polimorfismo, tipos de datos algebraicos, entre otros. La profundización en estos elementos y otros más novedosos y complejos puede permitir la ampliación del modelo para desempeñar funciones más sofisticadas de AOP.

Palabras clave: Programación funcional, Haskell, *Arrows*, Mónadas, Programación Orientada a Aspectos.

ABSTRACT

The objective of this project was to explore the functional language Haskell and the computing mechanisms it provides through the implementation of a model that simulates Aspect Oriented Programming (AOP). The model does not implement all the features found in AOP, only those related to join points and some elements for the weaving, which is performed manually.

For its development was necessary the research and learning of the Haskell functional language and some of its more advanced mechanisms such as Arrows, monads, and type system components including type classes, polymorphism, algebraic data types, and others. The model can be extended to perform more sophisticated functions of AOP, by using the mentioned mechanisms and including other more novel and complex aproaches.

Keywords: Functional programming, Haskell, Arrows, Monads, Aspect-Oriented Programming.

INTRODUCCIÓN

La Ingeniería de Sistemas por su naturaleza interdisciplinaria se enfrenta a una amplia gama de problemas en las distintas aéreas del conocimiento cuyas soluciones pueden ser más o menos complejas dependiendo en gran medida del enfoque utilizado para diseñar la solución. Por tanto, para tener criterios suficientes a la hora de escoger una u otra alternativa, es necesario conocer las diferentes posibilidades que existen para abordar un problema.

Sin embargo, la mayoría del trabajo que se desarrolla en la Universidad actualmente está enfocado en un estilo imperativo, dejando de lado paradigmas como la Programación funcional o la Programación lógica, que son prácticamente desconocidos por estudiantes y egresados, sesgando así el diseño de soluciones.

En respuesta a esta carencia de trabajo en el tema y en un intento por difundir la programación funcional en el ámbito académico en Colombia y, en particular en la Universidad, este proyecto pretende mediante la implementación de un modelo que simule Programación Orientada a Aspectos, explorar el lenguaje funcional Haskell y los mecanismos de computación que éste proporciona.

El modelo no implementa todas las características de la AOP. La idea se centra en definir los puntos de unión (*Join points*) de la AOP a través de las *Arrows* nombradas y ampliar los combinadores de *Arrows* para definir el sistema de tejido. Utilizando este esquema se presenta un ejemplo sencillo con la implementación de algunos aspectos clásicos, como *logs*, seguridad y depuración.

Por la naturaleza exploratoria del proyecto, antes de la implementación del modelo fue necesaria una fase de aprendizaje de los temas involucrados en su

desarrollo. Esta fase se cubrió en las dos de las cuatro etapas del proyecto. La primera etapa se dedicó al aprendizaje de Haskell y sus mecanismos de computación como herramientas para la implementación del modelo y, la segunda, al estudio de las nociones básicas de AOP que serían implementadas. En la tercera etapa se realizo el diseño e implementación del modelo con los conocimientos adquiridos en las primeras etapas y en la etapa final se realizó la documentación del proceso presentada en este trabajo.

La mayoría del contenido corresponde a la fundamentación teórica necesaria para el desarrollo del modelo y es presentada en los primeros capítulos. En el capítulo 2 se presenta el paradigma de programación funcional, sus características y enfoques. En el capítulo 3 se presenta el lenguaje funcional Haskell, incluyendo algunas de sus características más relevantes que además son utilizadas en el desarrollo de la solución. En los capítulos 4 y 5 se presenta la definición y utilización de mónadas y *Arrows*, respectivamente. En el capítulo 6 se presenta la AOP, sus conceptos y estructura. En el capítulo 7 se presenta una descripción general del trabajo relacionado con AOP en Haskell. Finalmente, en el capítulo 8 se presenta el modelo implementado utilizando los elementos descritos en la primera parte y un ejemplo ilustrativo de su manejo.

1. OBJETIVOS

1.2 OBJETIVO GENERAL

Explorar los mecanismos avanzados de computación del lenguaje *Haskell* mediante la implementación de un modelo que simule la Programación Orientada a Aspectos, utilizando *Arrows* y mónadas.

1.3 OBJETIVOS ESPECÍFICOS

- Aprender y entender los mecanismos de computación avanzada de *Haskell*: evaluación perezosa, mónadas, *Arrows*, el sistema de tipos de *Haskell*, tipos fantasmas y programación inicial de tipos.

- Diseñar e implementar un modelo utilizando características avanzadas de *Haskell*, de manera que permita a un programador imperativo acercarse a la complejidad de mónadas y *Arrows*.

- Ampliar las posibilidades de solución de problemas a través del aprendizaje de otro paradigma de programación.

- Introducir la Universidad y el Departamento de Sistemas en la programación declarativa.

2. PROGRAMACIÓN FUNCIONAL

La programación funcional es un paradigma de programación declarativa que se basa teóricamente en los principios del Cálculo lambda (Church, 1936), una teoría matemática de funciones desarrollada por Alonzo Church durante los años 30.

La programación declarativa se caracteriza por la utilización de expresiones que describen la solución de un problema sin manejar un estado implícito del programa, como en el paradigma imperativo, donde éste es controlado y modificado a través de la ejecución secuencial de comandos. Esto permite diseñar programas más claros y concisos pensando en "que" resultados se desean obtener y los elementos necesarios para llegar a ellos, en lugar de preocuparse por "cómo" debe hacerse el procesamiento para conseguirlos (Hudak, 1990), que oscurece y complica la lógica del algoritmo subyacente.

En particular, la programación funcional utiliza como concepto esencial la función, entendida en el sentido matemático de regla de correspondencia que asocia a cada de elemento de un conjunto de entrada un único elemento de un conjunto de salida. Así, al igual que en las matemáticas, las funciones denotan valores y pueden combinarse entre sí para incrementar su potencia de cálculo (Bird, 2000).

Por tanto, un programa funcional consiste enteramente de funciones. El programa principal es escrito como una función que recibe las entradas del programa como sus argumentos y produce la salida del programa como su resultado. La función principal, entonces, se define en términos de otras funciones más concretas, hasta llegar al nivel de las funciones primitivas del lenguaje (Hughes, 1989).

Este tratamiento de los programas proporciona un mayor nivel de abstracción para manipular datos, permitiendo escribir programas más concisos, legibles y de alto nivel (Mendel, et al., 1998) que modelan de forma más precisa el problema, especialmente si dicho problema está descrito en términos de funciones, como es el caso de aquellos relacionados con la ingeniería y las matemáticas.

Las ventajas de los lenguajes funcionales sobre los lenguajes imperativos los han hecho populares para el prototipado, la inteligencia artificial, los sistemas de comprobación matemática y las aplicaciones lógicas, pero han sido relegados de aplicaciones más generales por ser considerados menos eficientes (Louden, 2004).

Aunque los lenguajes funcionales dan un gran paso hacia un modelo de programación de alto nivel, ofrecen al programador menos control sobre la máquina, por lo que los programas funcionales suelen tener un desempeño inferior al de programas imperativos (HASKELL, 2006). Además, ciertos lenguajes funcionales (como Lisp (McCarthy, 1978)), por su naturaleza dinámica usualmente son interpretados en lugar de compilados, lo que resulta en una pérdida considerable en velocidad de ejecución (Louden, 2004).

Sin embargo, en la actualidad existen lenguajes funcionales como Clean (Plasmeijer, y otros, 2001), Miranda (Thompson, 1995) y Haskell REF, que ofrecen muy buen desempeño (casi cercano a los programas implementados en el lenguaje de programación C) debido a que son lenguajes compilados en lugar de interpretados y, además, cuentan con una gran cantidad de análisis y optimizaciones que permiten mejorar el desempeño del código de máquina traducido. Por otra parte, incluyen características como funciones de orden superior, evaluación perezosa, reconocimiento de patrones, varios mecanismos de abstracción de datos, entre otras (Hudak, 1990), que facilitan la modularización de los programas, permitiendo construir programas más

pequeños y más fáciles de implementar, mantener, depurar y reusar (Hughes, 1989).

Estas mejoras han hecho que los lenguajes funcionales sean muy atractivos para la programación en diversas áreas, convirtiéndose así en una alternativa razonable a los lenguajes imperativos, particularmente en situaciones donde la eficiencia de ejecución no es un requisito primordial.

2.1 LENGUAJES FUNCIONALES PUROS

Los lenguajes funcionales puros excluyen la utilización de variables en el sentido de la programación tradicional (como nombre de una localización de memoria), limitando su uso al de nombre para un valor. Como consecuencia de la ausencia de variables tampoco se tiene la asignación como una operación disponible, pues una vez que una variable es declarada no es posible asignarle un nuevo valor, de manera que conserva el mismo a lo largo de todo el programa (Louden, 2004)

Así, un programa en un lenguaje funcional puro es escrito como un conjunto de ecuaciones y todos los flujos de datos se realizan de forma explícita, lo que asegura que cada función solo depende del valor de sus argumentos e impide la ocurrencia de efectos colaterales sobre otras partes del programa. Esto permite, por un lado, utilizar evaluación perezosa con las ventajas subyacentes de generalidad y modularidad que ofrece (Wadler, 1995) y, por otro, razonar sobre los programas, pudiendo inferir propiedades generales de la función mediante la utilización de principios de inducción matemática y razonamiento algebraico (Mendel, et al., 1998).

A continuación se describen las propiedades más destacadas de los lenguajes funciones puros, algunas de las cuales también están presentes en los lenguajes impuros.

2.1.1 Transparencia referencial. Es la propiedad por la cual una expresión representa siempre el mismo valor en cualquier contexto en el que aparezca (Ruiz, et al., 2004). De esta forma, el valor de una función sólo depende del valor de sus parámetros y en ningún caso es afectado por el orden de ejecución o por cálculos previos a la llamada de la misma (Louden, 2004).

Gracias a esta propiedad las funciones de un programa tienen las mismas propiedades que las funciones matemáticas, con lo que es posible sustituir expresiones por otras equivalentes sin cambiar el significador del programa y llevar a cabo verificaciones del programa utilizando razonamientos matemáticos (Ruiz, et al., 2004).

2.1.2 Noción de estado. La carencia de asignación y la transparencia referencial implican que no existe una noción explícita del estado, debido a que no se maneja el concepto de localizaciones en la memoria con valores cambiantes. El ambiente de ejecución asocia nombres sólo a valores y no a localizaciones de memoria y una vez introducido un nombre en el ambiente su valor no puede cambiar jamás. Como consecuencia el orden de evaluación es irrelevante, ya que éste no avanza con base en el valor actual de los elementos, sino de acuerdo a las necesidades de cálculo de los valores involucrados en la ejecución de la funciones.

2.1.3 Funciones de orden superior. Son funciones que tienen parámetros o producen resultados que a su vez son funciones (Louden, 2004). Esto es posible debido a que en los lenguajes funcionales las funciones son consideradas *valores de primera clase*, lo que significa que una función puede aparecer en cualquier lugar donde aparezca un valor de otro tipo (Ruiz, et al., 2004).

2.1.4 Recursión. Como consecuencia de la ausencia de asignaciones no es posible realizar iteraciones. Para llevar a cabo los procesos iterativos se utiliza la recursividad.

2.1.5 Evaluación perezosa. Gracias a la transparencia referencial el orden de ejecución es irrelevante, de manera que los cálculos pueden realizarse en cualquier momento y arrojar el mismo resultado. Esto permite aplazar el cálculo de los valores hasta que se necesiten y evitar cálculos innecesarios.

La idea consiste en realizar orden de reducción normal al evaluar una expresión y recordar los valores de los argumentos ya calculados para evitar que sean recalculados. El orden de reducción normal evalúa la expresiones de afuera hacia adentro, en contraposición al orden aplicativo, que lo hace de adentro hacia afuera. Esta estrategia de evaluación siempre alcanza la forma normal de una expresión que la posea y además cuenta con la ventaja de reducir sólo aquellas subexpresiones que sean necesarias para calcular el resultado final. Esto además de evitar realizar cálculos innecesarios brinda la posibilidad de trabajar con estructuras de datos infinitas en los programas (Ruiz, et al., 2004).

2.2 LENGUAJES FUNCIONALES IMPUROS

Los lenguajes funcionales impuros extienden el Cálculo lambda con un número de posibles efectos, como asignación de estado, manejo de excepciones o invocación de continuaciones. Con esto ofrecen beneficios en el desempeño de los programas y algunas veces posibilitan un modo más compacto de expresión (Wadler, 1995).

La principal propiedad que se pierde al incluir estos efectos colaterales es la transparencia referencial, lo que a su vez deteriora el razonamiento matemático que puede ser realizado en los programas funcionales (Hudak, 1990).

Algunos lenguajes funcionales impuros son: Scala (Odersky, y otros, 2008), Lisp, Ocaml (Smith, 2006) y Standard ML (Milner, y otros, 1997).

2.3 LENGUAJES FUNCIONALES EN LA INDUSTRIA

Aunque tradicionalmente los lenguajes funcionales han sido utilizados en temas académicos y de investigación, en los últimos años están siendo adoptados en diversos entornos del mundo real que van desde tecnologías recientes hasta compañías financieras, de telecomunicaciones o biomédicas, entre muchas otras.

Commercial Users of Functional Programming CUFP (http://cufp.org/) es un seminario anual dedicado a divulgar la utilización de lenguajes funcionales en aplicaciones y, por compañías, reales. En su sitio en Internet se encuentran los reportes de cada una de las conferencias, con una gran cantidad de testimonios que ejemplifican los usos de este paradigma en la actualidad.

En el reporte de CUFP 2008 (Thompson, 2008) aparecen varios casos de la utilización de Haskell en diversas compañías, incluyendo compañías de componentes eléctricos, comunicaciones inalámbricas, biotecnología y financieras. Asimismo, se encuentran ejemplos de otros lenguajes como Erlang (Armstrong, 2007), un lenguaje funcional diseñado por la compañía Ericsson para construir sistemas paralelos, distribuidos y tolerantes a fallas. En el reporte es mencionado en aplicaciones para compañías web como Yahoo o Mochimedia (http://www.mochimedia.com/).

3. EL LENGUAJE FUNCIONAL HASKELL

Haskell recibe su nombre en honor al matemático estadounidense Hakell B. Curry, quien trabajó en los fundamentos de la lógica combinatoria (Curry, et al., 1958): uno de los pilares en los que se sustenta la programación funcional. Fue desarrollado por un grupo de expertos de varias universidades a finales de la década de los 80, con el objetivo de tener un lenguaje que reuniera las principales características de los lenguajes funcionales de la época e introdujera nuevos elementos como la sobrecarga de funciones y mecanismos para el manejo de entrada/salida, de manera que sirviera como estándar para la enseñanza y la industria (Ruiz, et al., 2004).

Haskell es un lenguaje funcional puro de propósito general, que incluye muchas innovaciones recientes en el diseño de lenguajes de programación. Proporciona funciones de orden superior, evaluación perezosa, reconocimiento de patrones, un sistema de módulos y un sistema monádico de entrada/salida (Peyton Jones, 2003). Cuenta igualmente con un sofisticado sistema de tipos que, además de ser polimórfico, ofrece una solución uniforme para sobrecarga, tipos de datos algebraicos e inferencia de tipos.

Desde su creación, el lenguaje ha pasado por varias versiones, siendo la versión actual Haskell 98, cuya sintaxis está definida en (Peyton Jones, 2003). A partir de esta definición se han realizado varias implementaciones estándar de Haskell y también algunas extensiones, que complementan el lenguaje base con aspectos que no han sido cubiertos durante el diseño de Haskell 98, pero que se consideran importantes en algunos ámbitos de aplicación. Estas implementaciones son de libre distribución y se encuentran disponibles en la página de Haskell *www.haskell.org*.

Haskell es un lenguaje robusto que involucra una gran cantidad de elementos y propiedades, de diferentes niveles de complejidad, que lo convierten en un lenguaje apto para desempeñarse en diversas aéreas de aplicación. En este trabajo no se pretende profundizar en todos los aspectos de Haskell, sólo se describirán algunas de sus características más notables o aquellas que juegan un rol importante en la implementación del modelo objeto de este proyecto.

3.1 FUNCIONES

3.1.1 Definición. Para introducir una función en Haskell, primero se declara, indicando el tipo de los argumentos que recibe y el tipo del resultado que produce. Luego se define, proporcionando las ecuaciones que darán lugar al cálculo que la función efectúa.

Para declarar el tipo correspondiente a las distintas funciones se utiliza el constructor (→). Si `t1, t2,…,tn, tr` son tipos validos, entonces `t1 → t2 → … → tn → tr` es el tipo de una función con `n` argumentos de tipos `t1, t2,…,tn` y un valor de retorno de tipo `tr`.

La definición consta del nombre de la función, seguida de los nombres asociados a cada parámetro formal, el signo = y la expresión que constituye el cuerpo de la función. Ejemplo:

```
inc :: Integer -> Integer
Inc x = x + 1
```

En esta definición, la primera línea declara `inc` como una función que tiene como argumento un valor de tipo `Integer` y devuelve como resultado un valor del mismo tipo. En la segunda línea se define `inc` como una función que incrementa en 1 el valor del argumento recibido.

En Haskell, las declaraciones de tipo pueden ser omitidas, ya que utiliza un sistema de inferencia de tipos con el cual siempre es posible conocer el tipo de un elemento a partir de su definición.

3.1.2 Comparación de patrones. En Haskell los parámetros formales de las funciones pueden ser tanto nombres de variables como patrones. De esta manera, las funciones pueden ser definidas a partir de varias ecuaciones, donde cada una de ellas establece el comportamiento de la función para distintas formas del argumento. Estas formas se capturan mediante patrones que al invocar la función son comparados para establecer la ecuación que debe utilizarse. Ejemplo:

```
length:: [a] -> Integer
length []= 0
length(x:xs) = 1 + length xs
```

Esta función determina el tamaño de una lista. Como puede verse en la forma de sus argumentos, la función está definida para cualquier lista. Si la lista es vacía se utilizará la primera ecuación y en otro caso, la segunda.

3.1.3 Expresiones lambda. En Haskell no es necesario nombrar las expresiones antes de usarlas, ya que permite definir funciones anónimas (sin nombre) usando expresiones lambda.

En Haskell una expresión lambda tiene la forma `\x → f(x)`, donde los parámetros formales se encuentran en la parte izquierda del símbolo → y el cuerpo o definición de la función en la parte derecha (Ruiz, et al., 2004). Por ejemplo la expresión lambda `\x y → x * x + y * y` denota una función que toma dos argumentos y devuelve como resultado la suma de sus cuadrados.

3.1.4 Currificación. La currificación es un mecanismo que permite remplazar argumentos estructurados por una secuencia de argumentos más simples (Bird, 2000). Así, para cada función `f` del tipo `f :: (a, b) → c` puede construirse una funciona análoga `f' :: a → b → c` que retorna el mismo resultado que `f` (Mendel, et al., 1998).

Asimismo, en Haskell las funciones con más de un argumento pueden interpretarse como funciones de un único argumento cuyo resultado es otra función. Por tanto, que una función tenga el tipo `f :: a → b →c` indica que la función `f` toma un argumento y retorna una función que tomará el siguiente valor y retornará el resultado buscado.

De acuerdo a esto, es posible definir funciones de `n` argumentos como funciones de orden superior que tomen un solo argumento y devuelvan otra función con `n-1` argumentos. Ejemplo:

```
sumaCuadrados :: Integer → Integer → Integer
sumaCuadrados x y = x * x + y * y
```

Es la definición de una función de dos argumentos que devuelve como resultado la suma del cuadrado de sus argumentos. Una definición equivalente puede darse utilizando expresiones lambda:

```
sumaCuadrados :: Integer → Integer → Integer
sumaCuadrados = \x y → x * x + y * y
```

Esta definición es un modo más cómodo de escribir:

```
sumaCuadrados :: Integer → (Integer → Integer)
sumaCuadrados = \x → (\y → x * x + y * y)
```

Esta equivalencia puede generalizarse de la siguiente manera (Ruiz, et al., 2004):

```
\x1 x2 … xn  → e ⇏ \x1 → (\x2 → (…  → (\xn → e)))
```

La currificación de funciones conlleva dos ventajas. En primer lugar la currificación puede ayudar a reducir el número de paréntesis que han de escribirse en las expresiones. En segundo lugar, las funciones currificadas pueden ser aplicadas a un solo argumento dando como resultado otra función (Bird, 2000). Ejemplo:

```
múltiploDe :: Integer → Integer → Bool
múltiploDe p n = n ‘mod’ p == 0
```

`múltiploDe` es una función que determina si el segundo argumento es múltiplo del primero. A partir de esta función es posible obtener otra que compruebe si un número es par, aplicando parcialmente el primer argumento:

```
esPar :: Integer → Bool
esPar = múltiploDe 2
```

Como puede verse, la función `esPar` utiliza en su definición `múltiploDe` pero le pasa un argumento, en lugar de dos.

3.2 SISTEMA DE TIPOS

El sistema de tipos de Haskell es estático y fuertemente tipado. La tipificación fuerte indica que todos los elementos del lenguaje (funciones, datos, operadores) tienen un tipo (Ruiz, et al., 2004), lo que permite verificar que se hace un uso consistente de los elementos del lenguaje. Estático, significa que la comprobación de tipos se hace en tiempo de compilación, permitiendo detectar

inconsistencias de tipo antes de la ejecución para garantizar que los programas son ejecutados sin errores de tipo (Cardelli, et al., 1985).

Este enfoque facilita la detección temprana de errores de tipo y permite una mayor eficiencia en tiempo de ejecución, sin embargo, hace el lenguaje menos flexible que sus homólogos no tipados. Haskell contrarresta esta rigidez con un sofisticado sistema de tipos que, además de ser polimórfico, permite la sobrecarga de funciones y la inferencia de tipos (Ruiz, et al., 2004). Consta de los siguientes sistemas de tipos:

- Un sistema de inferencia de tipos (Milner).
- Un sistema de tipos universal (Sistema F).
- Un sistema de tipos existencial que maneja los módulos.
- Un sistema de clasificación de alto orden que maneja las clases de tipos parametrizadas.

La descripción de estos sistemas es extensa y compleja por lo que no será dada aquí. Una explicación completa puede ser vista en (Pierce, 2002).

3.3 POLIMORFISMO

Haskell utiliza el sistema tradicional de tipos definido por Hindley-Milner (Milner, 1978), que le permite manejar polimorfismo paramétrico. Además, extiende el sistema de tipos con *clases de tipos,* para proporcionar una forma estructurada de manejar funciones sobrecargadas (Peyton Jones, 2003). Este enfoque fue propuesto por Wadler (Wadler, y otros, 1989) como una solución que generalizaba y unificaba la forma de tratar con *polimorfismo ad hoc* en los lenguajes funcionales.

En polimorfismo paramétrico, una función puede ser aplicada a un rango de tipos, que normalmente poseen una estructura común y, se comporta de

manera uniforme con todos los tipos (Cardelli, et al., 1985). Este hecho puede expresarse por cuantificación universal, para lo cual Haskell proporciona variables de tipo (denotan cualquier tipo) que son cuantificadas universalmente de forma implícita (Hudak, et al., 2000). Por ejemplo la declaración de la función identidad `id :: a → a` debe leerse como `id :: ∀a . a → a`. En esta declaración `a` es una variable de tipo. La declaración indica que la función `id` puede ser aplicada a cualquier tipo que se obtenga al sustituir la variable de tipo `a` por un tipo concreto (Ruiz, et al., 2004).

Para el caso del polimorfismo *ad hoc,* la función es aplicada a diferentes tipos, posiblemente con estructuras no relacionadas y, puede comportarse de forma diferente de acuerdo al tipo de sus argumentos (Cardelli, et al., 1985). Para manejar este comportamiento, la idea es introducir una noción de clases de tipos que capturen una colección de operadores sobrecargados de una manera consistente (Hudak, 1990), proporcionando así una solución uniforme para la sobrecarga, que incluya operadores de igualdad, aritméticos y conversión de cadenas (Hall, et al., 1996).

La estructura básica de las clases de tipos consiste en: una declaración de Clase que introduce una nueva clase de tipo y las operaciones sobrecargadas que deben ser soportadas por cualquier tipo que sea instancia de la clase y; una declaración de instancia, que declara que un tipo pertenece a una clase e incluye la definición de las operaciones sobrecargadas (métodos de la Clase), instanciadas en el tipo creado (Peyton Jones, 2003).

Una explicación precisa de la definición de clases de tipos en Haskell está dada en (Wadler, y otros, 1989) y (Hall, et al., 1996), donde se definen un conjunto de reglas de inferencia para resolver la sobrecarga introducida por las clases de tipos.

3.4 TIPOS DE DATOS ALGEBRAICOS

Los tipos de datos algebraicos son tipos definidos por el usuario, que proporcionan un mecanismo para modelar tipos que describen de una forma precisa los elementos que componen un problema. Son semejantes a la declaración de gramáticas.

Un tipo de dato algebraico se declara con la palabra reservada `data`, seguido por el nombre del tipo, el signo igual y los constructores del tipo definido. Esta construcción adiciona al sistema de tipos un nuevo tipo y una serie de constructores para crear datos de dicho tipo. La forma general de la definición de un tipo algebraico es:

data T

$= Con_1 \; t_{11} \ldots t_{1k_1}$

$Con_2 \; t_{21} \ldots t_{2k_2}$

…

$Con_n \; t_{n1} \ldots t_{nk_n}$

Donde `T` es el constructor de tipo (el nombre del tipo) y cada Con_ies un constructor de datos seguido por k_i tipos (k_i es un entero mayor o igual a cero) (Thompson, 1999). Un constructor de tipo construye un tipo mientras que un constructor de datos (de ahora en adelante, constructor) construye valores de dicho tipo (Ruiz, et al., 2004).

Cada constructor Con_i puede ser visto como una función constructora del tipo `T` con k_i argumentos de tipos t_{ik_i}. Los elementos del tipo `T` se construyen aplicando estas funciones constructoras a argumentos de los tipos dados en la definición. Por lo tanto, estas funciones tienen el tipo:

$Con_i = :: t_1 \rightarrow \ldots t_k \rightarrow T$

De acuerdo a la forma en que sean definidos, lo tipos algebraicos pueden ser clasificados en: tipos numerados, productos, uniones, tipos recursivos o tipos algebraicos polimórficos (Thompson, 1999).

3.4.1 Tipos enumerados. Es el tipo algebraico más simple. Consta de un número finito de valores que son enumerados en la definición del tipo. Ejemplo:

```
data DiaSemana = Lunes | Martes | Miércoles | Jueves
                 | Viernes | Sábado | Domingo
```

Esta declaración adiciona el tipo `DiaSemana` cuyos posibles valores son `Lunes`, `Martes`,... o `Domingo`

3.4.2 Productos. Son tipos que están compuestos por varias componentes. Se llaman así debido a que los elementos del nuevo tipo son el producto cartesiano de los tipos de las componentes. Ejemplo:

```
data Racional = Par Integer Integer
```

Según esta definición un `Racional` es un `Par` constituido por dos valores de tipo `Integer` cuyos elementos están constituidos por el producto cartesiano de Z x Z

3.4.3 Uniones. Son tipos que se definen uniendo tipos existentes, que dan alternativas para la creación de elementos del tipo definido (Ruiz, et al., 2004). Ejemplo:

```
data Figura = Círculo Float | Rectángulo Float Float
```

El tipo `Figura` se define uniendo los tipos `Círculo` y `Rectángulo`, lo que significa que hay dos formas de construir un elemento `Figura`: la primera es

utilizando el constructor `Círculo` seguido de su radio (que debe ser de tipo `Float`) y la segunda, utilizando el constructor `Rectángulo` seguido de sus lados (también de tipo `Float`).

3.4.4 Tipos recursivos. Son tipos que se definen en términos de ellos mismos, es decir utilizando recursión primitiva. Ejemplo:

```
data NTree = NilT | Node Int NTree NTree
```

el tipo `Ntree` define un árbol de enteros que puede ser nulo o estar formado por un nodo, que es la combinación de un valor entero con dos subárboles.

3.4.5 Tipos algebraicos polimórficos. Son tipos que contienen en su definición variables de tipo, que pueden ser reemplazadas por tipos concretos, permitiendo la creación de tipos polimórficos (Thompson, 1999).

Un tipo polimórfico permite construir tipos concretos al ser sustituido por un tipo específico. Ejemplo:

```
data Par a = UnPar a a
```

`Par a` representa el tipo de todos los pares formados por dos elementos de tipo `a`. Al sustituir la variable de tipo `a` por un tipo concreto como `Int`, se obtiene el tipo `Par Int`, que representa el tipo de todos los pares formados por dos elementos de tipo `Int`.

3.4.6 Algunos tipos polimórficos predefinidos

- **El tipo `Maybe`**

```
data Maybe a = Nothing | Just a
```

El tipo polimórfico `Maybe` proporciona un mecanismo para tratar con valores no definidos u opcionales. Un resultado correcto de tipo `a` es representado como `Just a` mientras que un resultado incorrecto es representado como `Nothing` (Peyton Jones, 2003).

El tipo `Maybe` es además una mónada (capítulo 4) utilizada para el manejo de errores, en la que éstos son representados con `Nothing`. Así, el tipo `Maybe a` representa el tipo de todas computaciones que pueden retornar un valor de tipo `a` o ningún valor (`Nothing`) (Newbern, 2006).

- **El tipo `Either`**

```
data Either a b = Left a | Right b
```

El tipo `Either` representa valores con dos posibilidades: `Left a` o `Right b`. Este tipo algunas veces es usado para representar un valor que es correcto o es un error: por convención el constructor `Left` se usa para encapsular un valor erróneo y el constructor `Right` se usa para encapsular un valor correcto (HASKELL, 2006).

3.5 SISTEMA DE CLASES

Con el sistema de clases de Haskell es posible restringir el tipo de ciertas funciones polimórficas (Ruiz, et al., 2004), ya que permite definir funciones que

sólo tienen sentido para un conjunto de tipos, que se agrupan al pertenecer a la misma a clase y, que no están disponibles para los tipos ajenos a la clase.

A diferencia de POO, donde las clases se construyen utilizando el concepto de subtipos, que da lugar a una relación de pertenencia a otro tipo (relación padre - hijo), en Haskell, el concepto de clases está basado en la agrupación de tipos en un conjunto, que es una relación de pertenencia.

De esta manera, una clase de tipo en Haskell puede pensarse como un conjunto de tipos que soporta una cierta colección de funciones, que son especificadas como parte de la declaración de la clase (Jones, 1999). Cada instancia de la clase es un tipo que define de forma particular las funciones sobrecargadas asociadas con la clase.

Una declaración de clase introduce una nueva clase y proporciona los nombres y declaraciones de tipo de las operaciones que son compartidas por los tipos de la clase (Hall, et al., 1996). Una declaración de clase tiene la forma:

```
class cx => C u where
    vi :: cxi => ti
```

Esta declaración define una nueva clase de tipo `C`, con v_i funciones miembro o métodos que serán compartidos por las instancias de la clase (Peyton Jones, 2003). `C u` no es una expresión de tipo, sino que expresa una restricción sobre un tipo y se denomina un *contexto* (Hudak, et al., 2000). La variable de tipo `u` representa un tipo genérico instancia de la clase y su uso es para poder parametrizar el tipo de las funciones v_i. `cx` es un contexto que especifica las superclases de `C`.

En la declaración de cada método v_i, t_i corresponde a la declaración de tipo del método, la cual debe contener la variable de tipo `u` y puede incluir otras

variables de tipo w, sin embargo, el contexto cx_i solo puede aparecer para restringir variables de tipo diferentes a u (Peyton Jones, 2003). Un ejemplo es la declaración de la clase Eq que se utiliza en Haskell para agrupar aquellos tipos susceptibles de ser comparados.

```
class Eq a where
    (==) :: a → a → Bool
    (/=) :: a → a → Bool
```

Esta declaración define la clase de tipo Eq con los métodos (==) y (/=).
Una declaración de instancia especifica un tipo como instancia de una clase y define el comportamiento de cada método de la clase para el tipo instanciado (Hudak, et al., 2000). La forma general de una declaración de instancia es:

```
instance cx' => C (T u1 ... uk) where { d }
```

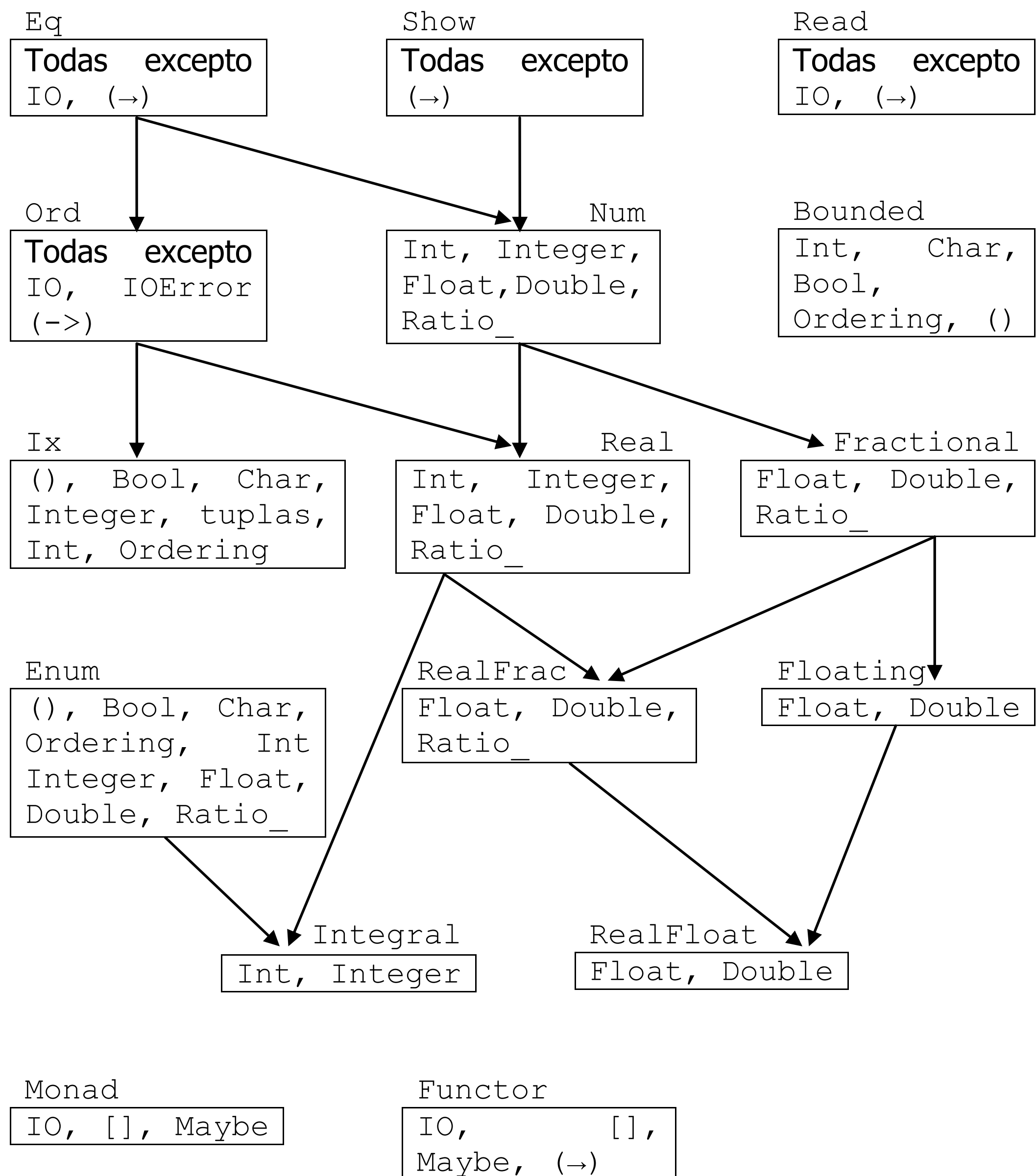
Esta declaración define $T\ u_1\ \ldots\ u_k$ como instancia de la clase C. El tipo $(T\ u_1\ \ldots\ u_k)$ representa un constructor de tipo T aplicado a las variables de tipo $u_1, \ldots u_k$ (con $k>=0$), que deben satisfacer las restricciones en el contexto de instancia cx'. d contiene las definiciones de los métodos de la clase C, especificados en la definición de ésta (Peyton Jones, 2003). Ejemplo:

```
instance Eq Integer where
    x == y = integerEq x y
    …
```

En este ejemplo el tipo Integer es declarado como instancia de la clase Eq y se da la definición para el método (==) utilizando la función integerEq que es una primitiva que compara dos enteros.

Las relaciones entre contextos dan lugar a una jerarquía de clases. El Prelude establece la jerarquía inicial dada en la figura 1, que puede ser ampliada a medida que se introducen nuevas clases (Ruiz, et al., 2004).

Figura 1. Clases del Prelude



3.6 IMPLEMENTACIONES

Como se mencionó anteriormente, se han desarrollado varias implementaciones para Haskell 98. La mayoría soporta casi todas las definiciones estándar del lenguaje y algunas incluyen extensiones de éste, con fines experimentales o para mejorar sus prestaciones. Toda la información sobre estas implementaciones puede encontrarse en la página http://www.haskell.org/haskellwiki/Implementations. A continuación se mencionan algunas de las más conocidas.

3.6.1 Glasgow Haskell Compiler (GHC). Es un compilador optimizado para Haskell, que proporciona muchas extensiones del lenguaje. Produce programas con una alta velocidad de ejecución. Está disponible para la mayoría de plataformas, incluyendo Windows, Mac OS X, y algunas variantes Unix (Linux, *BSD, Solaris).

3.6.2 Hugs. Es un intérprete de Haskell pequeño y portable, escrito en C. Está disponible para todas las plataformas Unix, incluyendo Linux, DOS, Windows 3.x, y Win 32 (Windows 95, Win32s, NT).

3.6.3 nhc98. Es un compilador estándar de Haskell 98, pequeño y fácil de instalar. Produce código con velocidad de ejecución media-alta y su velocidad de compilación es bastante rápida. Está disponible para todas las plataformas Unix, incluyendo Mac OS X, Cygwin/Windows, Linux, Solaris, *BSD, AIX, HP-UX, Ultrix, IRIX, etc.

3.6.4 York Haskell Compiler (YHC). Es una derivación de nhc98, con el objetivo de ser más sencillo, más portable, más eficiente e integrar soporte para Hat, un depurador de Haskell (http://www.haskell.org/hat/).

3.6.5 Utrecht Haskell Compiler (UHC). Es una implementación de Haskell realizada por la Universidad de Utrecht. Es compatible con casi todas las características Haskell98, además de incluir muchas extensiones experimentales. Está disponible para plataformas Mac OS X, Windows (cygwin), y algunos sistemas Unix.

4. MÓNADAS

Las mónadas, un concepto de la teoría de categorías (Mac Lane, 1998), fueron introducidas por Moggi (Moggi, y otros, 2001) para estructurar la descripción de diversas características de los lenguajes de programación, como estado y manejo de excepciones (Hudak, et al., 2007). Basado en este trabajo, Wadler propuso las mónadas como un nuevo método para la estructuración de programas funcionales que permite encapsular efectos impuros de una manera pura (Wadler, 1990). De esta forma, las mónadas proporcionan un marco de trabajo que simula efectos encontrados en otros lenguajes, como estado global, manejo de excepciones, entrada/salida o indeterminismo (Wadler, 1995).

Las mónadas, además de imitar efectos de características impuras, proporcionan beneficios que no son fácilmente obtenidos con tales características, como el hecho de que los tipos de un programa reflejen que efectos ocurren (Wadler, 1992). En muchos lenguajes de programación la evaluación puede tener implícitos efectos laterales que no son predichos por el tipo de la expresión. Con mónadas, se asignan diferentes tipos para valores (cuya evaluación es pura) y para computaciones (cuya evaluación puede tener efectos laterales) (Moggi, y otros, 2001), de manera que los tipos son usados para indicar que partes de un programa pueden tener que clases de efectos (Wadler, 1990). Por ejemplo, en Haskell cualquier función con tipo `IO a` representa una función que tendrá como resultado un valor de tipo `a` pero además realizará una operación de entrada/salida.

4.1 DEFINICIÓN

Una mónada consiste en un constructor de tipo `M` y un par de funciones polimórficas (Wadler, 1992):

```
return :: a → M a
(>>=) :: M a → (a → M b) → M b
```

El constructor de tipo `M` define un tipo de computación (Newbern, 2006), por lo que con distintas declaraciones de `M` se modelarán diferentes características. `M a` representa un valor de tipo `a` encapsulado por `M` (Ruiz, et al., 2004). Así, `M a` debe leerse como el tipo de una computación cuyo valor de retorno es de tipo `a` con un posible efecto lateral capturado por `M` (Hudak, et al., 2007).

La función `return` construye valores del tipo de computación `M` (Newbern, 2006) encapsulando un valor de tipo `a` en un valor de tipo `M a`, es decir, convierte un valor simple en el equivalente monádico.

El operador `(>>=)`, combina los valores monádicos `M a` y `M b` a partir de una función que transforme valores de tipo `a` en valores de tipos `M b`. Así la expresión `m >>= f` donde `m` es una mónada con tipo `M a` y `f` una función con tipo `a → M b`, aplica la función `f` a cada dato encapsulado en la mónada `m`, devolviendo una nueva mónada con tipo `M b` (Ruiz, et al., 2004).

De acuerdo a las definiciones anteriores, una mónada puede ser vista como un contenedor en el que es posible almacenar diferentes valores. `M a` representa un contenedor que tiene valores de tipo `a`. La función `return` pone valores en el contenedor y el operador `>>=` toma el valor del contendor y lo pasa a una función que produce un contenedor con nuevos valores, posiblemente de un tipo diferente (Newbern, 2006).

4.2 LEYES DE LAS MÓNADAS

Matemáticamente las mónadas están caracterizadas por un conjunto de leyes (o propiedades) que deberían cumplir las operaciones monádicas (Hudak, et al., 2000):

1. `(return x) >>= f == f x`
2. `m >>= return == m`
3. `(m >>= f) >>= g == m >>= (\x -> f x >>= g)`

La primera y segunda ley requieren que `return` se comporte como identidad, por la izquierda y por la derecha, con respecto al operador `>>=`. La tercera ley exige que el operador `>>=` sea asociativo (Newbern, 2006).

4.3 CLASES MONÁDICAS

En Haskell, una mónada se construye sobre un tipo polimórfico que se declara como instancia de una o todas las clases monádicas, `Functor`, `Monad`, y `MonadPlus` (Hudak, et al., 2000).

4.3.1 Clase `Functor`

```
class Functor m where
    fmap :: (a → b) → (m a → m b)
```

Define un método `fmap` que se utiliza para aplicar una función a todos los elementos pertenecientes a una estructura arbitraria `m`, devolviendo una estructura con la misma forma.

4.3.2 Clase `Monad`

```
class Monad m where
    return :: a → m a
    (>>=) :: m a → (a → m b) → m b
    (>>) :: m a → m b → m b
    return :: a → m a
    fail :: String → m a
```

donde:

```
m >> k = m >>= \_ → k
fail s = error s
```

Además de las funciones básicas `return` y `>>=`, en esta definición aparecen dos nuevas funciones: `(>>)` y `fail`. El operador `>>` se comporta como el operador `>>=` pero no usa el valor producido por la primera operación monádica. `fail s` corresponde a una computación que falla, dando el mensaje de error `s`.

4.3.3 Clase `MonadPlus`

```
class(Monad m) => MonadPlus m where
    mzero :: m a
    mplus :: m a → m a → m a
```

`MonadPlus` es una subclase de la clase `Monad` en la que se definen dos operaciones: `mzero`, que representa una estructura vacía y, `mplus` que permite unir dos estructuras.

4.4 NOTACIÓN do

La notación do proporciona una forma simple para expresar computaciones monádicas (Ruiz, et al., 2004). Las reglas para la traducción de esta notación son:

```
do e1 = e1
do e1 ; e2 = e1 >> do e2
do p ← e1; e2 =
              e1 >>= (\v -> case v of p -> e2; _ -> fail s)
```

Donde El tipo de `e1` y `e2` es un tipo monádico `m a` y la expresión `p` es un patrón que se compara con el valor dentro de la mónada (Newbern, 2006). Así, en el tercer caso, como `e2` puede depender del patrón `p`, si éste falla se produce un error y se invoca la función `fail` con el mensaje de error `s` identificando el patrón que falló (Ruiz, et al., 2004).

En síntesis, la notación `do` permite escribir computaciones monádicas utilizando un estilo pseudo-imperativo con variables nombradas. El resultado de una computación monádica puede ser "asignado" a una variable mediante el operador (<-). Luego, se realiza el enlace utilizando esa variable en una computación monádica subsiguiente.

De esta forma, la notación `do`, se asemeja a un lenguaje de programación imperativo, en el que se construye una computación a partir de una secuencia explicita de computaciones más simples. Con esto, las mónadas ofrecen la posibilidad de crear computaciones de estilo imperativo dentro de un programa funcional más amplio (Newbern, 2006).

4.5 EJEMPLO

El tipo Lista es una mónada que puede ser utilizada para representar cómputos indeterministas (Ruiz, et al., 2004), es decir computaciones que pueden tener 0, 1 o más resultados válidos. El conjunto de posibles resultados en representado como una lista.

La mónada Lista incorpora la estrategia de combinar una secuencia de computaciones indeterminista, mediante la aplicación de las operaciones a todos los valores posibles en cada paso. Esto resulta útil cuando las computaciones deben lidiar con ambigüedad, ya que permite explorar todas las posibilidades (Newbern, 2006).

La definición de la Lista como mónada es:

```
instance Monad [] where
    m >>= f  = concatMap f m
    return x = [x]
    fail s   = []
```

En este caso `return x` es la función que crea una lista unitaria `[x]`.el operador `(>>=)` permite secuenciar cómputos indeterministas, aplicando la función `f` a todos los elementos de la lista y creando una nueva lista con sus resultados.

```
instance MonadPlus [] where
    mzero = []
    mplus = (++)
```

mzero es la computación que no produce ningún resultado, m1 ‘mplus’ m2 es la computación indeterminista que puede producir tanto los resultados de m1 como los de m2.

Un ejemplo de aplicación de la mónada Lista es:

```
type Indet a = [a]
demo :: Indet Integer
demo = do let amb = return 10 ‘mplus’ return 4
    x ← amb
    y ← amb
    return (x + y)
```

En este ejemplo la variable amb es una lista con dos posibles valores: 10 y 4. El resultado de la computación demo es la suma de la variable amb consigo misma, lo que al ejecutarse tendrá como resultado la lista [20,14,14,8].

5. *ARROWS*

Las *Arrows*[1] fueron definidas por Hugs (Hughes, 2000) como una generalización de las mónadas que, al igual que éstas, proporcionan una interfaz estandarizada para bibliotecas, pero pueden ser aplicadas a tipos de bibliotecas no monádicas, por lo que es posible emplearlas para brindar los beneficios de la programación monádica a una amplia clase de aplicaciones.

Este mecanismo permite dar una estructura común a programas basados en diferentes nociones de computación como computaciones con componentes estáticos, independientes de la entrada, o computaciones con múltiples entradas (Paterson, 2001). De esta forma, las *Arrows* ofrecen un modo competitivo de representar computaciones en Haskell. Sin embargo, su propósito no es reemplazar a las mónadas, sino brindar los beneficios de una interfaz compartida a una clase más amplia de aplicaciones que las mónadas no pueden cubrir.

Las mónadas a pesar de lo convenientes que resultan en la estructuración de programas funcionales, sufren una severa restricción respecto a la forma en que pueden tomar sus argumentos. Un programa monádico aunque puede producir su salida de muchas maneras diferentes, puede tomar su entrada sólo de una forma: a través de los parámetros de una función. Esto se debe a que las computaciones monádicas son parametrizadas sobre el tipo de sus salidas y no de sus entradas. Las computaciones *Arrows*, por su parte, se parametrizan sobre ambas, por lo que pueden tomar su entradas de muchas formas diferentes, dependiendo del *Arrow* que se utilice (Hughes, 2005).

---

[1] En este capítulo se introducirá un concepto que tiene sus raíces en la Teoría de Categorías (Mac Lane, 1998), por lo que muchas de las definiciones internas están basadas en esta teoría. Aquí los conceptos serán presentados desde un punto de vista programático. Si se requiere una mayor profundización en el diseño teórico, seguir las referencias.

Para hacer la dependencia en la entrada explicita, se construye un *Arrow* sobre un tipo parametrizado con dos parámetros, en lugar de uno. Así, de la misma manera que un tipo monádico `m a` representa una computación cuyo valor de retorno es de tipo `a`, un tipo *Arrow* `a b c` (que es la aplicación del tipo parametrizado `a` a los parámetros `b` y `c`) representa una computación que toma una entrada de tipo `b` y produce una salida de tipo `c` (Hughes, 2000).

5.1 CLASES *ARROW*

Las operaciones que pueden ser aplicadas a *Arrows* están divididas en varias clases debido a que no todos los tipos de *Arrows* soportan todas las operaciones. Estas clases están definidas en `Control.Arrow` (HASKELL, 2006), incluyendo todos sus combinadores y métodos (Hughes, 2005).

En la definición inicial dada por Hugs, la clase `Arrow` definía las operaciones `arr` y `>>>`, análogas a `return` y `>>=` de la clase `Monad` y, el método `first`, como combinador básico de pares. Sin embargo, en la implementación actual de la clase `Arrow`, ésta se define como una subclase de la clase `Category` y sólo se definen `first` y `arr` como métodos básicos. El operador `>>>` ha sido definido en la clase `Category`, por lo cual sigue siendo un operador valido para la clase `Arrow` aunque no se encuentre definido dentro de ésta.

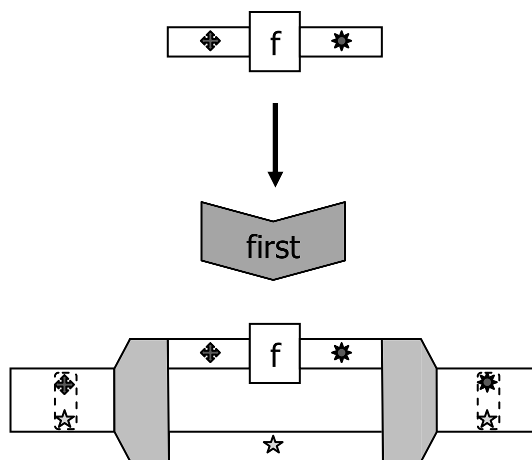
5.1.1 *Arrows* y pares. `Arrow`, la clase básica de *Arrows*, define el método `arr` y una serie de operadores que permiten combinar los resultados de varias *Arrows* (Hughes, 2005). `arr` y `first` son los métodos primitivos, por lo que deben ser definidos al declarar una instancia de la clase. El resto de combinadores tienen definiciones por defecto a partir de los combinadores primitivos, pero pueden ser sustituidas si es necesario.

```
class Category a => Arrow a where
    arr :: (b -> c) -> a b c
    first :: a b c -> a (b, d) (c, d)
    second :: a b c -> a (d, b) (d, c)
    (***) :: a b c -> a b' c' -> a (b, b') (c, c')
    (&&&) :: a b c -> a b c' -> a b (c, c')
```

- `arr f` convierte una función de `b` a `c` en un *Arrow* de `b` a `c`.

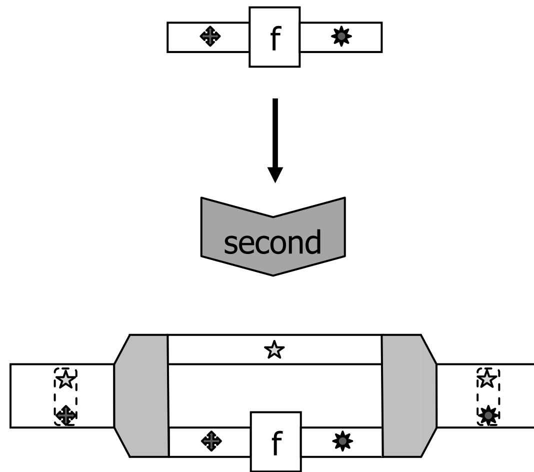
- `first f` construye un *Arrow* de pares a pares que pasa el primer componente a través de `f` y el segundo lo pasa sin cambios. Gráficamente:

Figura 2. Combinador first



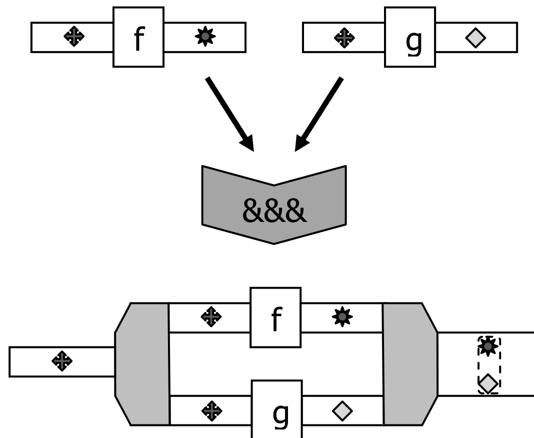
- `second f` construye un *Arrow* de pares a pares que pasa el segundo componente a través de `f` y el primero lo pasa sin cambios. Gráficamente:

Figura 3. Combinador second



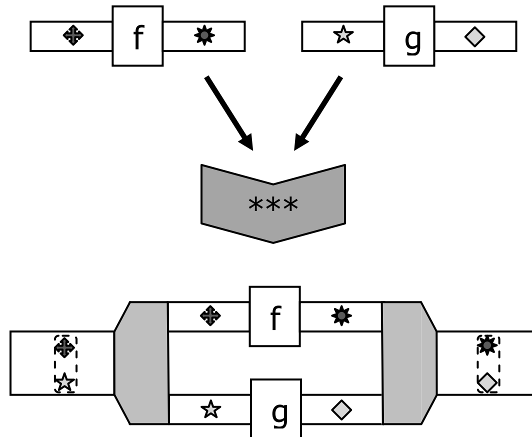
• `f &&& g` toma dos *Arrows* con el mismo tipo de entrada y la duplica de forma que pueda ser procesada por `f` y `g`, poniendo sus resultados en el par de salida. Gráficamente:

Figura 4. Combinador &&&



• `f *** g` construye un *Arrow* de pares a pares que pasa el primer componente a través de `f` y el segundo componentes a través de `g`. Gráficamente:

Figura 5. Combinador ***



5.1.2 *Arrows* y condicionales. `ArrowChoice`, una subclase de la clase `Arrow`, define una serie de operadores que permiten tomar una decisión entre dos *Arrows* basándose en un resultado previo. No todos los tipos de *Arrows* tienen soporte para los operadores de decisión, por lo que fueron definidos en `ArrowChoice`, para diferenciar entre los *Arrows* que soportan o no dichos operadores.

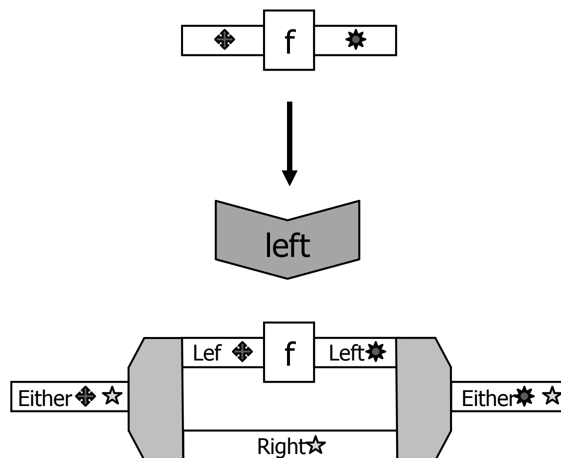
Al igual que en la clase `Arrow`, donde `first` es el combinador básico, en la clase `ArrowChoice` se define `left` como combinador primitivo y los métodos restantes son definidos a partir de éste (Hughes, 2005).

En la definición de todos los combinadores de la clase `ArrowChoice` se utiliza el tipo `Either` (sección 3.4.6) para tener una manera estándar de tratar los argumentos los *Arrows* y poder tomar decisiones sobre ellos.

```
class Arrow a => ArrowChoice a where
    left :: a b c -> a (Either b d) (Either c d)
    right :: a b c -> a (Either d b) (Either d c)
    (|||) :: a b d -> a c d -> a (Either b c) d
    (+++) :: a b c -> a b' c'
                  -> a (Either b b') (Either c c')
```

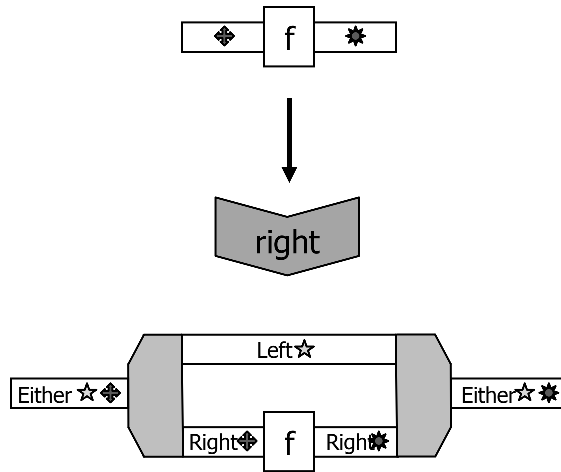
- `left f` aplica la función `f` a entradas etiquetadas con `Left`, pasando sin cambios entradas etiquetadas con `Right` y, etiqueta la salida de `f` con `Left`. Gráficamente:

Figura 6. Combinador left



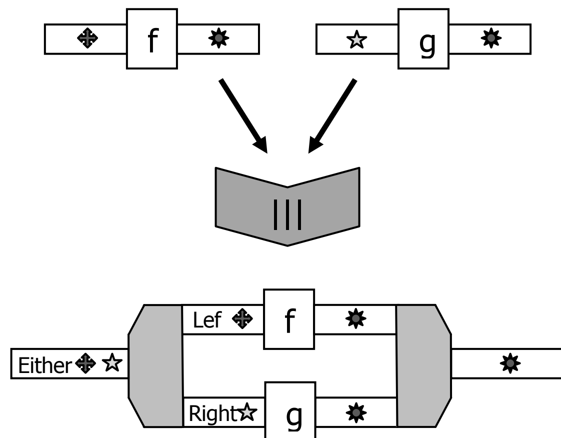
- `right f` aplica la función `f` a entradas etiquetadas con `Right`, pasando sin cambios entradas etiquetadas con `Left` y, etiqueta la salida de `f` con `Right`. Gráficamente:

Figura 7. Combinador right



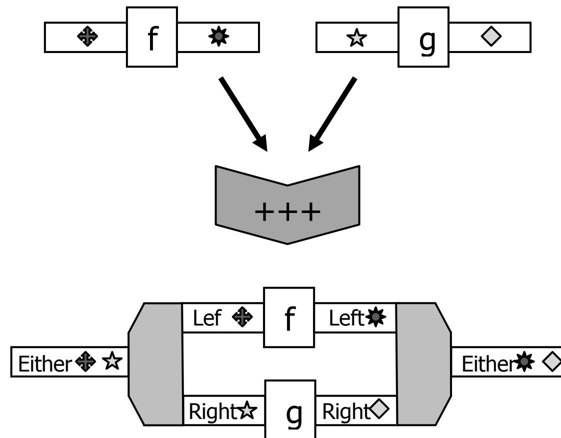
- `f ||| g` aplica la función `f` a entradas etiquetadas con `Left` y la función `g` a entradas etiquetadas con `Right`. Es importante notar que en este *Arrow* se requiere que `f` y `g` tengan el mismo tipo de salida. Gráficamente:

Figura 8. Combinador |||



- `f +++ g` aplica la función `f` a entradas etiquetadas con `Left` y la función `g` a entradas etiquetadas con `Right`, etiquetando la salida de `f` con `Left` y la salida de `g` con `Right`. Permite tener *Arrows* con diferentes tipos de salida. Gráficamente:

Figura 9. Combinador +++



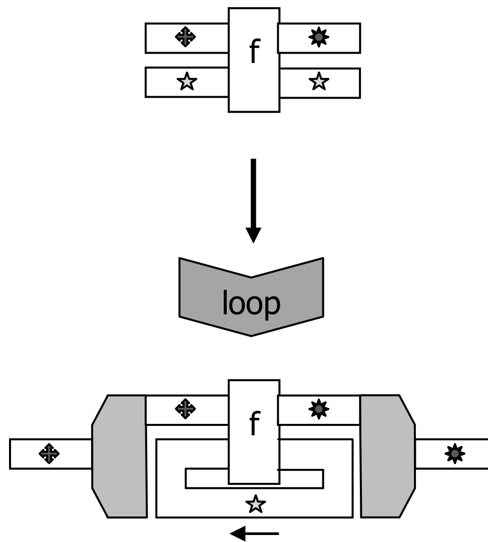
5.1.3 *Arrows* y realimentación. En (Paterson, 2001), la noción de realimentación fue extendida a *Arrows* para soportar la definición recursiva de valores en una computación, donde una salida es utilizada como entrada.

Para expresar este comportamiento se introduce la clase: `ArrowLoop`, subclase de la clase `Arrow`, con un combinador de realimentación.

```
class Arrow a => ArrowLoop a where
    loop :: a (b,d) (c,d) → a b c
```

Con `loop f` el segundo componente de la salida de `f` es realimentado como segundo componente de su entrada. Gráficamente:

Figura 10. Operador loop



El operador `loop` debe satisfacer un conjunto de axiomas que no se mencionaran aquí. Una explicación detallada de éstos puede ser encontrada en (Paterson, 2001).

5.1.4 Mónadas como *Arrows*. Para cualquier mónada `m`, las funciones del tipo `a → m b` pueden ser representadas mediante *Arrows*. Con este propósito se define el tipo `Kleisli`, que puede ser parametrizado sobre cualquier mónada y se declara como instancia de la clase `Arrow`, con las respectivas implementaciones de sus métodos.

```
newtype Kleisli m a b = Kleisli {runKleisli :: a → m b}

instance Monad m => Arrow (Kleisli m) where
    arr f = Kleisli (return . f)
    first (Kleisli f) = Kleisli (\(a,c) → do b ← f a
                                return (b,c))
```

5.1.5 Funciones como *Arrows*. Para utilizar las funciones ordinarias como *Arrows*, se declara el tipo función `(→)` como instancia de la clase `Arrow` (Hughes, 2000).

```
instance Arrow (→) where
    arr f = f
    first f = f *** id
    second f = id *** f
```

5.2 NOTACIÓN

La escritura de programas utilizando *Arrows* impone un estilo de programación *point-free*, que consiste en construir funciones por composición de otras funciones (Hughes, 2005). Este estilo es conveniente para definiciones generales pero puede ser muy difícil de manejar para definiciones más específicas.

Para facilitar la programación con *Arrows*, Ross Paterson realizó en (Paterson, 2001) una extensión de Haskell, similar al estilo monádico, definiendo la sintaxis y un conjunto de reglas de traslación asociadas a ésta, para una nueva forma de expresión en la definición de *Arrows*.

5.2.1 Abstracciones *Arrow*. Una abstracción *Arrow* es una expresión de la forma `proc pat → body`. Es análoga a una expresión lambda en el sentido de definir un *Arrow* y ligar un nombre o patrón a su entrada, que puede ser usado en el cuerpo de la abstracción. El cuerpo de una abstracción es nueva categoría sintáctica llamada comando, que permite construir *Arrows* utilizando sus combinadores en una notación más conveniente.

5.2.2 Comando de aplicación. El tipo más simple de comando es el de aplicación de *Arrow*: `f -< exp`, que envía el valor de la expresión `exp` como

entrada del *Arrow* `f`. A partir de este comando es posible formar comandos compuestos por otros más simples, como el comando condicional. La regla de traslación es:

```
proc pat -> a -< e = arr (\pat -> e) >>> a
```

5.2.3 Comando condicional. Este comando permite seleccionar entre dos comandos `c1` o `c2` de acuerdo al valor de la expresión `e`. La regla de traslación es:

```
proc pat → if e then c1 else c2 =
    arr (\pat → if e then Left pat else Right pat)
    >>> (proc pat → c1 ||| proc pat → c2)}
```

5.2.4 Notación do. La notación `do` brinda la posibilidad de nombrar valores intermedios en un comando. Sin embargo, mientras un bloque `do` monádico es una expresión, un bloque `do` en *Arrows* es un comando y por tanto sólo puede aparecen dentro de una abstracción *Arrow*. Asimismo, mientras las sentencias `x ← e` en un `do` monádico ligan un nombre al resultado de una expresión `e`, la forma *Arrow* liga un nombre a la salida de un comando. La regla de traslación es:

```
proc pat → do x ← c1 =
    (arr id &&& proc pat → c1) >>> proc (pat,x) → c2
```

5.2.5 Combinadores de comandos. En notación *Arrow*, un comando describe un *Arrow* de tipo `arr env a` (donde `env` es el tipo del ambiente en el que el comando aparece y `a` es el tipo de su salida), lo que permite utilizar combinadores de *Arrows* como combinadores de comandos.

Sin embargo, todos los combinadores de *Arrows* no pueden usarse como combinadores de comandos debido a que es necesario que en la definición del combinador todos los *Arrows* argumentos tengan el mismo tipo de entrada. Esta restricción se debe a que al aplicar un combinador de comandos, todos los comandos que se pasan como argumentos ocurren en el mismo ambiente en el que aparece el combinador y, por lo tanto, todos tienen entradas de tipo `env`. Así, el tipo de un combinador de comandos debe tener la forma:

```
arr env a → arr env b → ... → arr env c
```

La regla de traslación para un operador infijo es:

```
proc pat → c1 op c2 = proc pat → (| (op) c c2 |)
```

Cuando un combinador de comandos no es un operador infijo, las aplicaciones deben ser encerradas en *banana brackets* `(| |)` para distinguirlos de llamadas a funciones ordinarias. La regla de traslación de operadores prefijos es:

```
proc pat → (| op c1...cn |) =
    op (proc pat → c1)...(proc pat → cn)
```

6. PROGRAMACIÓN ORIENTADA A ASPECTOS

El concepto de programación orientada a aspectos (AOP) (Kiczales, et al., 1997) fue introducido por Gregor Kiczales y su grupo en 1997, como un nuevo enfoque para mejorar la separación de incumbencias *(concerns)* de software y de esta forma resolver las limitaciones que se presentan en muchos paradigmas de programación, cuando no es posible separar y modularizar algunas propiedades especiales encontradas en sistemas complejos.

La separación de incumbencias es un principio de software introducido por Dijkstra (Dijkstra, 1976), en el que se establece que el software debe ser descompuesto de manera que diferentes incumbencias de un problema sean resueltas en módulos separados (Win, et al., 2002). Una incumbencia es una propiedad o área de interés de un problema (Elrad, et al., 2001) que se quiere tratar como una única unidad conceptual. Las incumbencias son modularizadas a través del desarrollo de software usando diferentes abstracciones.

Los paradigmas de programación actuales proporcionan mecanismos de abstracción como subrutinas, procedimientos, funciones, objetos, módulos y otros que permiten separar y modularizar incumbencias descomponiendo el programa en unidades concretas que desempeñan una función bien definida (Mens, et al., 1998). Estos mecanismos funcionan bastante bien en el modelamiento e implementación de la funcionalidad básica del sistema, pero no son suficientes para capturar claramente todas las decisiones de diseño que el programa debe implementar, debido a que ciertas incumbencias representan propiedades del comportamiento del sistema y atraviesan transversalmente la modularidad del resto de la implementación (Kiczales, et al., 1997).

Algunos ejemplos de incumbencias transversales incluyen políticas de sincronización que requieren todo un conjunto de operaciones para el

seguimiento de un protocolo de bloqueo consistente, mecanismos de tolerancia a fallos que requieren creación consistente de copias redundantes, políticas de calidad del servicio que requieren una puesta a punto de las prioridades del sistema, entre muchas otras (Elrad, et al., 2001).

6.1 CÓDIGO DISPERSO Y ENREDADO

El inadecuado manejo de incumbencias transversales produce efectos poco deseables en el código de los programas: por un lado, la aparición de código disperso *(scattered code)*, es decir, que el código para una incumbencia particular se encuentra en múltiples lugares en el diseño y, por otra parte, código enredado *(tangled code)*, que se presenta cuando en la definición de una unidad funcional, ésta debe interactuar simultáneamente con múltiples incumbencias que no hacen parte de su funcionalidad. Estas dos características producen los siguientes problemas:

- El código no puede ser reusado debido a que contiene parte de una incumbencia transversal que posiblemente no existirá en la nueva implementación.

- El código es difícil de mantener debido a que cualquier cambio en la implementación de la incumbencia transversal requiere múltiples cambios en múltiples lugares. Además, el código enredado oscurece la funcionalidad básica de un componente, haciendo difícil centrarse en el código esencial.

- Facilita la introducción de errores debido a que es difícil asegurar que todo el código disperso esté escrito consistentemente y, si se agrega una nueva característica también es difícil asegurar que todas las instancias del código disperso son actualizadas. Además, el tener que tratar con incumbencias transversales cuando se intenta escribir la funcionalidad de un componente, distrae de lo que el componte se supone que debe hacer (Robinson, 2006).

6.2 ENFOQUE DE AOP

AOP se enfoca en mecanismos para simplificar el manejo y comprensión de incumbencias transversales (Elrad, et al., 2001), centrándose en la noción de aspecto como una abstracción destinada a modularizar tales incumbencias y proporcionar así mantenibilidad y reusabilidad al sistema. Sin embargo, las técnicas orientadas a aspectos no abogan por descartar los mecanismos para separación de incumbencias existentes, en su lugar, el objetivo es proporcionar mecanismos complementarios para soportar la identificación, representación, modularización y composición sistemática de incumbencias que atraviesan transversalmente el sistema y que de otra manera estarían dispersas a través de varios módulos (Rashid, y otros, 2003).

6.2.1 Aspectos y componentes. En la separación por aspectos, se sostiene que en una aplicación se pueden distinguir dos tipos de incumbencias:

- **Aspectos.** Son propiedades no funcionales, que hacen parte de la definición del sistema en sí mismo, describiendo cuestiones claves relacionadas con la semántica o el rendimiento y que normalmente cortan transversalmente la aplicación al estar presentes en muchos de sus unidades funcionales. Algunos ejemplos son los patrones de acceso a memoria, la sincronización de procesos concurrentes, el manejo de errores, etc.

- **Componentes.** Son las unidades de descomposición funcional del sistema, es decir, aquellas propiedades que se puede encapsular claramente en un procedimiento.

En términos de las definiciones anteriores, el objetivo de la programación orientada a aspectos es, entonces, posibilitar una adecuada modularización de las aplicaciones permitiendo la separación de los componentes y los aspectos al

proporcionar mecanismos que hagan posible abstraerlos y componerlos para formar todo el sistema, en contraste con los paradigmas tradicionales que sólo se preocupan por la separación de componentes (Kiczales, et al., 1997).

6.2.2 Beneficios. Con la adecuada encapsulación y localización de aspectos se busca obtener las siguientes ventajas:

- **Modularidad.** El código para una incumbencia transversal debe estar localizado en un archivo de código fuente.

- **Uniformidad.** Una incumbencia transversal debe ser tratada uniformemente en toda la aplicación.

- **No invasividad.** Debe ser posible cambiar o extender la implementación de incumbencias transversales sin invadir o afectar la implementación de otros componentes.

- **Transparencia.** Las incumbencias transversales deber ser transparentes a los desarrolladores.

- **Reusabilidad.** Los componentes de software reusables deben ser desarrollados sin conocimiento del ambiente y de las incumbencias transversales con las que serán reutilizados (Schwanninger, y otros, 2004)

6.2.3 Estructura. La estructura de una aplicación implementada con AOP es semejante a la de otros métodos de generalización de procedimientos. En el caso de AOP la estructura consta de:

1. Elementos sintácticos:
a. Un lenguaje de componentes con el cual programar los componentes.
b. Uno o más lenguajes de aspectos con los cuales programar los aspectos.

2. Un tejedor de aspectos *(aspect weaver)* para combinar los lenguajes.

3. Runtime:
a. Un programa de componentes que implemente los componentes de la aplicación usando el lenguaje de éstos.
b. Uno o más programas de aspectos que implementen los aspectos de la aplicación usando los lenguajes de aspectos.

- **Lenguaje de componentes.** Este lenguaje debe permitir la implementación de componentes acordes a las funcionalidades del sistema, mientras asegura que estos programas no interfieren con los aspectos que se controlaran con los programas de aspectos.

- **Lenguaje de aspectos.** Este lenguaje permite la implementación de los aspectos definidos.

Los lenguajes de componentes y aspectos tienen diferentes mecanismos de abstracción y composición, pero deben poseer ciertos términos en común que permitan al tejedor combinar ambos tipos de programas para la formación del sistema definido.

- **Tejedor de aspectos.** Su función es combinar los lenguajes de componentes y de aspectos de tal manera que se obtenga la funcionalidad del sistema diseñado.

Un concepto esencial en el tejedor de aspectos es el de punto de unión *(join point)*, que se refiere a puntos comunes entre el programa de componentes y el de aspectos que permiten su coordinación (Kiczales, et al., 1997).

Los componentes y los aspectos pueden ser combinados de manera estática o de manera dinámica.

- **Tejido estático.** El tejido estático implica modificar el código fuente de un componente insertando sentencias específicas de aspectos en ciertos puntos de unión. Es decir, que el código del aspecto se introduce en el del componente. Como resultado se obtiene un tejido de código altamente optimizado, cuya velocidad de ejecución es comparable a la del código escrito sin utilizar aspectos. Por tanto, el tejido estático impide que el nivel de abstracción adicional introducido por la programación orientada a aspectos provoque un impacto negativo en el rendimiento del programa. Sin embargo, es bastante difícil identificar los aspectos en el código tejido, por lo que la adaptación o sustitución de aspectos dinámicamente en tiempo de ejecución puede consumir demasiado tiempo o no ser siempre posible.

- **Tejido dinámico**. Un requisito para el tejido dinámico es la existencia explícita de aspectos tanto en tiempo de compilación como en tiempo de ejecución. Por lo tanto los aspectos y las estructuras tejidas se deben materializar como objetos y deben mantenerse en el ejecutable. Dada una interfaz de reflexión de aspectos, el tejedor es capaz de añadir, adaptar y eliminar aspectos dinámicamente, durante la ejecución. El tejido dinámico facilita el tejido incremental y hace más fácil la depuración. Sin embargo, en una implementación, los aspectos que no deban adaptarse en tiempo de ejecución deben ser tejidos estáticamente por motivos de rendimiento (Böllert, 1999).

7. TRABAJO RELACIONADO

7.1 AOP CON CLASES DE TIPOS

En (Sulzmann, et al., 2007) se desarrolla un modelo de AOP, basado en clases de tipos, realizando una extensión AOP de Haskell (AOP HAskell) mediante un esquema de transformación dirigido por la sintaxis, donde los estilos de programación AOP son mapeados a clases de tipos. Así, el tipado y traslación de AOP Haskell puede ser explicado en términos de tipado y traslación del programa de clases de tipos resultante. El esquema diseñado se basa en clases de tipos multiparamétricas e instancias superpuestas, características que no son parte de Haskell 98 estándar pero son soportadas por GHC.

7.1.1 Enfoque. Para imitar AOP mediante Clases de tipos se introduce AOP Haskell, que extiende la sintaxis de Haskell soportando definiciones de aspectos de alto nivel de la siguiente forma:

```
N@advice #f1,...,fn# :: (C => t) = e
```

Esta expresión define un *point cut* en el que se declara que el aspecto `N` será aplicado sobre las funciones `f1,…, fn` y se define su comportamiento. `N` es una etiqueta que representa el aspecto y `f1, …, fn` son un conjunto de funciones que representan los puntos de unión (*join points*), en los cuales sólo pueden ser insertados aspectos que remplazan funciones. Cada definición de *point cut* tiene un tipo asociado, definido por la notación de tipo `C => t` que sigue la sintaxis de Haskell. Los aspectos son aplicados si el tipo de un punto de unión `fi` es una instancia de `t` que satisface las restricciones de `C`. finalmente, `e` define el cuerpo del aspecto y sigue la sintaxis para expresiones de Haskell.

Para embeber AOP Haskell en Haskell se utiliza el potente sistema de tipos de Haskell con dos extensiones de GHC: clases de tipos multiparamétricas e instancias superpuestas, en el desarrollo de un esquema de transformación con el cual el programa fuente en AOP Haskell es descrito en términos de un programa objeto Haskell.

En el modelo desarrollado se emplea tejido estático y la idea clave es expresar el tejido estático dirigido por la sintaxis mediante clases de tipo, basándose en los siguientes principios:

- Emplear instancias de clases de tipos para representar aspectos.

- Usar un preprocesador sintáctico para la instauración de los puntos de unión con llamadas a la función sobrecargada de tejido.

- Describir el tejido estático dirigido por tipos como una resolución de clase de tipo, que es el proceso de realizar el tipado y traslación de programas de clases de tipos.

El esquema de traslación diseñado traduce declaraciones de aspectos a declaraciones de instancia. Para hacerlo se introduce la clase de tipo biparamétrica `Advice` con el método `joinpoint`:

```
class Advice n t where
joinpoint :: n -> t -> t
joinpoint _ = id -- caso por defecto
```

Luego cada aspecto es convertido en una instancia, donde el parámetro `n` de la clase `Advice` representa el nombre `N` del aspecto y el parámetro `t` el tipo del *point cut*:

```
data N = N
instance C => Advice N t where
joinpoint _ proceed = e
instance Advice N a
```

Para la definición del método `joinpoint` en cada instancia, simplemente se copia el cuerpo del advice remplazando `proceed` por el nombre de la función que será sustituida con el advice. Adicionalmente, para cada advice `N` se introduce una instancia `Advice N a`, donde el cuerpo de esta instancia es establecido al caso por defecto especificado en la declaración de la clase. Así para cada advice se crean dos instancias superpuestas que pueden ser potencialmente usadas para resolver una restricción de clase de tipo.

El tejido en los puntos de unión se realiza remplazando cada función `f` con:

```
joinpoint N1 (... (joinpoint Nm f)...)
```

Donde `N1,…Nm` son aspectos en los que `f` aparece en su *point cut*.

7.1.2 Limitaciones. Este esquema de traslación tiene las siguientes restricciones:

- Los aspectos implementados deben ser puros, es decir, libres de la ocurrencia de efectos laterales. Esto significa que aspectos como el *log* no pueden ser implementados en este modelo.

- Las funciones que serán modificadas con aspectos no deben tener declaración de tipo, pues éstas tendrían que ser reescritas para evitar incompatibilidades al aplicar el esquema de transformación.

- El esquema de transformación se basa en tipos de clases multiparamétricas e instancias superpuestas, características que no hacen parte de la definición estándar de Haskell 98 aunque son soportadas por GHC.

7.2 GRAMÁTICAS DE ATRIBUTOS

En los trabajos que se describen a continuación no se desarrolla propiamente un modelo de AOP para la definición de aspectos de manera general, sino que se utiliza AOP para la implementación de gramáticas de atributos. La idea básica es descomponer las gramáticas en diferentes aspectos que luego son combinados mediante los operadores definidos para formar la gramática completa.

En (Moor, et al., 2000) se muestra la utilización de AOP para estructurar compiladores implementados como gramáticas de atributos, las cuales proporcionan una notación conveniente para especificar las funciones que interactúan con cada regla de producción de la sintaxis abstracta del lenguaje. Sin embargo los compiladores escritos de esta manera sufren una carencia de modularidad al considerar todos los aspectos de la semántica simultáneamente, sin la posibilidad de separar sus diferentes incumbencias.

En el compilador propuesto se utiliza una versión de Haskell equipada con registros extensibles para la manipulación de los aspectos, donde éstos representan un conjunto de definiciones de uno o más atributos relacionados, implementados como unidades semánticas independientes que pueden ser parametrizadas, manipuladas y compiladas independientemente. En el ejemplo se utilizan aspectos para definir los atributos de nivel léxico (`levels`), ambiente (`envs`), variables locales (`locss`) y código objeto (`codes`) los cuales son combinados en una gramática de atributos de la siguiente manera:

```
ag () = knit (levels() ‘cat’ envs() ‘cat’ locss()
       ‘cat’ codes())
```

Donde `knit` y `cat` son funciones para combinar aspectos en gramáticas de atributos. Con este esquema es posible escribir nuevos aspectos y adicionarlos a la gramática de atributos utilizando los combinadores definidos.

En este trabajo el tejido de los aspectos en el código existente se basa únicamente en nombres y no depende de un análisis sofisticado del programa como en otras implementaciones de AOP. Por otra parte, en AOP el método tradicional para la composición de programas no es la sustitución sino la introducción o combinación de aspectos, sin embargo el ejemplo de este trabajo divide completamente la gramática de atributos original en términos de aspectos, abandonando así el método de composición primaria.

En un trabajo independiente (Viera, et al., 2009), aunque relacionado con el anterior, se define una biblioteca para embeber gramáticas de atributos en Haskell. Los aspectos representan una colección de atributos relacionados, con sus reglas de definición. La solución se expresa en términos de colecciones heterogéneas, mejorando así la solución dada en el ejemplo preliminar al evitar el amplio uso de combinadores indexados que se realiza en éste. Para la combinación de aspectos se define una clase y las instancias que implementan el operador de combinación según diferentes casos. En la implementación de la biblioteca presentada se utilizan ampliamente conceptos y mecanismos avanzados como tipos fantasmas, clases multiparamétricas y programación a nivel de tipos.

7.3 RELACIÓN ENTRE MÓNADAS Y AOP

Estos trabajos no se centran en el desarrollo de modelos AOP o en la utilización de sus principios en la programación de de un problema particular, en su lugar,

presentan una discusión teórica en la que se busca establecer la relación existente entre mónadas y AOP. El primer trabajo al respecto fue elaborado por Meuter (Meuter, 2007), quien plantea que las semejanzas entre AOP y programación monádica son notables en varios puntos.

Para empezar, ambos estructuran el código por capas. La capa superior es la capa funcional, es decir, el código actual del programa y, las otras capas son los aspectos (o diferentes mónadas) que ayudan al código actual a completar su tarea. Los puntos de unión permiten a diferentes aspectos mezclase con el código funcional en un programa monádico y son representados por las funciones que acceden a la información de las monadas.

Por otra parte, tanto mónadas como aspectos tienen repercusiones en todo el sistema. Cuando se amplía un programa funcional monádico con la utilización de `return` y `bind` se obtiene código que está completamente enredado con la información global del programa.

Finalmente, al igual que en AOP, la programación monádica obliga al programador a separar los diferentes "aspectos" en mónadas, de forma que al describir su funcionalidad e interacción sea posible integrarlas fácilmente con el uso del compilador.

Más adelante, en (Hofer, et al., 2007) se realiza una comparación entre ambos mecanismos con respecto a sus capacidades y efectos sobre la modularidad, para lo cual se especifican las nociones consideradas como la esencia de cada concepto. A partir de allí se analizan las mónadas como una forma de expresar incumbencias transversales en programación funcional, y se discute la medida en que la programación monádica puede considerarse como una forma de AOP. Luego, considerando que las mónadas expresan incumbencias de las computaciones al realizar la abstracción sobre sus tipos, se analiza en qué

medida los aspectos de AOP son capaces de desempeñar este rol de abstracción al representar incumbencias.

Como resultado de la discusión se concluye que las mónadas y los aspectos deben ser considerados como mecanismos diferentes que no pueden expresarse unos a otros. Por un lado, con las mónadas no es posible especificar un conjunto de condiciones en el programa para la introducción de incumbencias de forma automática. Por otra parte, los aspectos no son muy útiles en la abstracción de capacidades computacionales, como por ejemplo expresar computaciones que puedan fallar, al igual que las representadas por la mónadas `Maybe` o `Error`, mediante mecanismos AO, pues para hacerlo se utilizan mecanismos de manejo de excepciones que hacen parte del lenguaje base y no de los mecanismos computacionales de AOP.

8. DISEÑO DEL MODELO

8.1 ASPECTOS Y COMPONENTES

Una de las tareas fundamentales de la AOP es facilitar la modularidad al combinar aspectos con componentes, donde los primeros representan una incumbencia transversal al programa que se modulariza en computaciones para ser combinada con los segundos, que son también computaciones con una función bien definida. Los *Arrows*, por su parte, sirven para representar computaciones, por lo que serán utilizadas como unidad fundamental de este modelo, representando tanto aspectos como componentes.

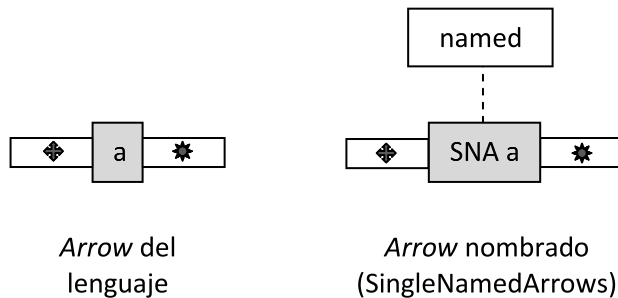
Para realizar el enlace entre estos dos elementos se optó por agregar un identificador a cada *Arrow*, de manera que fuera útil para decidir qué aspecto se enlaza a que componente. El *Arrow* nombrado (`SingleNamedArrows`) debe ser similar en estructura y comportamiento a un *Arrow* del lenguaje (Figura 11) para lo cual se definió un nuevo tipo de dato, que no es más que un *Arrow* acompañado de un nombre y, se declaró como instancia de las clases `Category` y `Arrow` para tener acceso a sus métodos. La declaración del *Arrow* nombrado (`SingleNamedArrow`) es la siguiente:

```
data (Arrow a) => SingleNamedArrow a b c = SNA
                  { named :: String
                  , runNamedArrow :: a b c }
```

`SingleNamedArrow` es un tipo de dato algebraico polimórfico y de múltiples parámetros, donde `Arrow a` indica que `a` debe ser un *Arrow* y por lo tanto sólo pueden construirse tipos de `SingleNamedArrow` con *Arrows*, asegurando así que el tipo creado es consistente con el modelo. `b` y `c` son las entradas y salidas del *Arrow*, `named` corresponde al campo de identificación con la cual es

posible dar un nombre al *Arrow* (que no necesariamente es único) y `runNamedArrrow` es un campo que permite ejecutar el *Arrow* interno.

Figura 11. *Arrow* del lenguaje y *Arrow* nombrado



La declaración de `SingleNamedArrow` como instancia de la Clase `Category` y `Arrow` es:

```
instance (Arrow a) => Category (SingleNamedArrow a) where
    id = SNA "id" (returnA id)
    (SNA s f) . (SNA _ g) = SNA s (f <<< g)

instance (Arrow a) => Arrow (SingleNamedArrow a) where
    arr f           = SNA "Unkown" (arr f)
    first (SNA s f) = SNA s (first f)
```

Debido a que el tipo `SingleNamedArrow` es un tipo polimórfico que forma tipos concretos al remplazar la variable de tipo `a` por un *Arrow* particular, la implementación de los métodos de las clases `Category y Arrow` básicamente es la definición dada en las mismas clases, precedida del constructor de datos `SNA` y un identificador, de manera que la implementación real de los métodos está dada por el *Arrow* específico con el que se cree el tipo `SingleNamedArrow`.

Por otra parte, se creó la clase `NamedArrow` con operaciones específicas para el nuevo tipo. La siguiente es la definición de la clase `NamedArrow` y la declaración de `SingleNamedArrow` como instancia de ésta:

```
class (Arrow a) => NamedArrow a where
    getName :: a b c -> String

instance (Arrow a) => NamedArrow (SingleNamedArrow a) where
    getName (SNA s _) = s
```

Como puede verse la clase `NamedArrow` define el método `getName` para obtener un `String` a partir de un *Arrow* dado. Al ser implementado con el tipo `SingleNamedArrow` el método `getName` se convierte en el método que dado un *Arrow* nombrado `SingleNamedArrow`, devuelve su identificador.

Para realizar el tejido entre aspectos y componentes se define la clase `Aaop`, cuyos métodos son los combinadores descritos en la sección 7.3 y que se encuentran definidos e implementados en `AaopBasis.hs`.

```
class (Arrow a) => Aaop a where
   (#>|) :: a b b  -> a b c -> a b c
   (#<|) :: a b c  -> a b b -> a b c
   (?>|) :: a b () -> a b c -> a b c
   (?<|) :: a b c -> a b () -> a b c
   (|>#) :: a c c  -> a b c -> a b c
   (|<#) :: a b c  -> a c c -> a b c
   (|>?) :: a c () -> a b c -> a b c
   (|<?) :: a b c -> a c () -> a b c
   (=>#) :: a b c  -> a b c -> a b c
   (#>=) :: a b c  -> a b c -> a b c
```

Para tener acceso a estos métodos, `SingleNamedArrow` es declarado como instancia de la clase `Aaop`. La implementación de los combinadores utiliza el identificador de `SingleNamedArrow` para decidir si se combinan o no aspectos y componentes utilizando los operadores implementados en `AaopBasis.hs`.

```
instance (Arrow a) => Aaop (SingleNamedArrow a)  where
   (SNA s f) #>| ar@(SNA s' g)
        | s =~ s'   = SNA s' (f AB.#>| g)
        | otherwise = ar

   ar@ (SNA s’ g) #<| (SNA s f)
        | s =~ s'   = SNA s' (g AB.#<| f)
        | otherwise = ar

   (SNA s f) ?>| ar@(SNA s' g)
        | s =~ s'   = SNA s' (f AB.?>| g)
        | otherwise = ar

   ar@ (SNA s’ g) ?<| (SNA s f)
        | s =~ s'   = SNA s' (g AB.?<| f)
        | otherwise = ar

   (SNA s f) |># ar@(SNA s' g)
        | s =~ s'   = SNA s' (f AB.|># g)
        | otherwise = ar

   ar@(SNA s’ g) |<# (SNA s f)
        | s =~ s'   = SNA s' (g AB.|<# f)
        | otherwise = ar
```

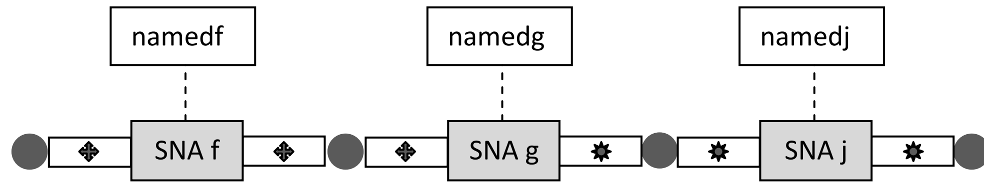
```
(SNA s f) |>? ar@(SNA s' g)
      | s =~ s'   = SNA s' (f AB.|>? g)
      | otherwise = ar

ar@(SNA s’ g) |<? (SNA s f)
      | s =~ s'   = SNA s' (g AB.|<? f)
      | otherwise = ar

(SNA s f) =># ar@(SNA s' g)
      | s =~ s'   = SNA s' (f AB.=># g)
      | otherwise = ar

(SNA s f) #>= ar@(SNA s' g)
      | s =~ s'   = SNA s' (f AB.#>= g)
      | otherwise = ar
```

La coincidencia entre los nombres del aspecto y el componente se hace a través de expresiones regulares facilitando la creación de expresiones regulares complejas para ser aplicadas en un mayor número de situaciones.

8.2 PUNTOS DE UNIÓN

Los *Arrows* interactúan entre sí a través de sus entradas y salidas, en las que intercambian valores que terminan uniéndolos para formar un *Arrow* más complejo. Para llevar a cabo la combinación de aspectos y componentes se definieron los puntos de unión en las intersecciones entre *Arrows* (Figura 12), por lo que un aspecto puede ser insertado antes o después de un componente. En estos puntos es posible insertar aspectos que modifican o no su argumento y, como consecuencia, la entrada o salida del componente. También es posible poner un aspecto en lugar de un componente, remplazando así la función desarrollada por éste.

Figura 12. Puntos de unión



Cuando se une un aspecto antes de un componente, el enlace es realizado a través de la entrada del componente, lo que implica que la salida del aspecto puede ser enlazada con la entrada del componente. Por esta razón deben tener el mismo tipo de entrada y la salida del aspecto deberá coincidir o no con la entrada del componente, dependiendo de si el aspecto modifica o no el valor de entrada del componente.

En el caso de insertar un aspecto después de un componente, el enlace es realizado mediante la salida del componente, lo que implica que la salida del componente puede ser enlazada a la entrada del aspecto. De forma análoga al caso anterior, el componente y el aspecto deben tener el mismo tipo de salida y la entrada del aspecto deberá coincidir o no con la salida del componente, dependiendo de si el aspecto modifica o no el valor de salida del componente.

8.3 TEJIDO

Una de las características de AOP es el mecanismo de tejido, que combina los aspectos en los puntos definidos por el programador. En la mayoría de compiladores AOP, este proceso se realiza de forma estática (aunque existe versiones dinámicas) utilizando el sistema de tipos del compilador del lenguaje de componentes para garantizar la validez de los mismos. Por ejemplo, en AspectJ el tejido es realizado por el tejedor combinando los aspectos en diferentes puntos del programa de componentes para generar código en este lenguaje (Java en este caso). Luego, el programa modificado por la

combinación de aspectos es compilado con el compilador de Java, que se encarga de realizar la verificación de los tipos.

En Haskell, es posible realizar el tejido de forma automática utilizando programación a nivel de tipos, pero esto requiere un conocimiento de la teoría de tipos que está fuera del alcance de este proyecto. Por esta razón el tejido se realiza de forma manual a nivel de programación. En un proyecto futuro, se retomará la construcción de este tejedor.

Con el fin de realizar el enlace entre aspectos y componentes se definió una serie de operadores (también llamados combinadores) que permiten combinar aspectos antes y después de componentes de dos formas diferentes: aspectos que modifican el argumento y aspectos que no lo modifican.

Los aspectos que modifican el argumento que reciben realizan alguna operación cuyo resultado será puesto en la salida. Así, si el aspecto es insertado antes, el componente recibirá su argumento modificado por la operación que realizó el aspecto. Si el aspecto es insertado después del componente, el valor de salida se verá alterado por la operación del aspecto.

Otros aspectos no modifican el argumento que reciben, como en el caso de aspectos de *log* o de persistencia, que sólo realizan una operación que involucra el valor recibido pero sin alterarlo. De esta forma, si el aspecto es insertado antes, el componente recibe su entrada sin modificación y, si es insertado después del componente, el valor de salida seguirá siendo el que arrojó el componente.

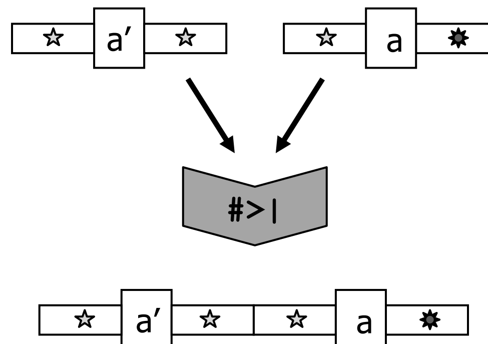
8.3.1 Aspectos antes de componentes

- **Combinador (#>|)**

```
(#>|) :: Arrow a => a b b -> a b c -> a b c
a' #>|  a = proc b -> do
             b' <- a' -< b
             c  <- a  -< b'
                returnA -< c
```

`a' #>| a` inserta el aspecto `a'` antes del componente `a`, pasando el resultado de `a'` como argumento de `a`. Al aplicar `a'` sobre el valor de entrada, éste es modificado antes de ser utilizado por `a`, por lo cual el tipo de entrada y de salida de `a'` debe ser el mismo y debe coincidir con el tipo de entrada de `a`. Gráficamente:

Figura 13. Combinador #>|



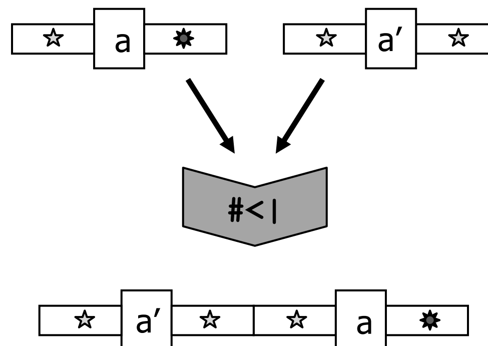
Este combinador es útil para insertar aspectos que modifican el valor de entrada del componente. Por ejemplo un aspecto para incrementar el valor de las entradas de los componentes.

- **Combinador `(#<|)`**

```
(#<|) :: Arrow a => a b c -> a b b -> a b c
a #<| a' = a' #>| a
```

`a #<| a'` realiza la misma operación que el combinador anterior pero modificando el orden de los argumentos. De esta manera, el aspecto `a'` es insertado antes del componente `a`. Gráficamente:

Figura 14. Combinador #<|

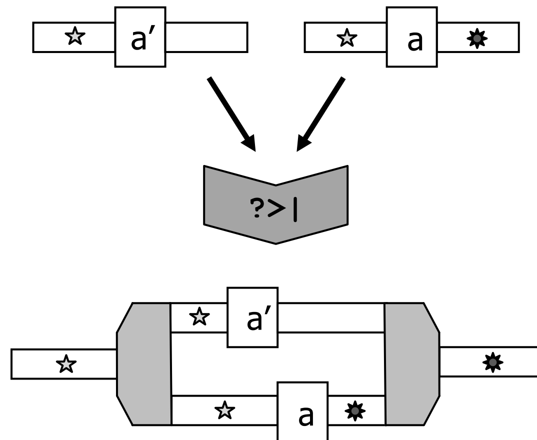


- **Combinador `(?>|)`**

```
(?>|) :: Arrow a => a b () -> a b c -> a b c
a' ?>| a = proc b -> do
             b' <- a' -< b
             c  <- a  -< b
             returnA -< c
```

`a' ?>| a` inserta el aspecto `a'` antes del componente `a`, sin embargo el resultado de `a'` no es pasado como argumento de `a`, que toma su argumento directamente de la entrada, sin modificación alguna por la aplicación de `a'`. De acuerdo a esto, el tipo de entrada de `a'` y `a` debe ser el mismo. Gráficamente:

Figura 15. Combinador ?>|



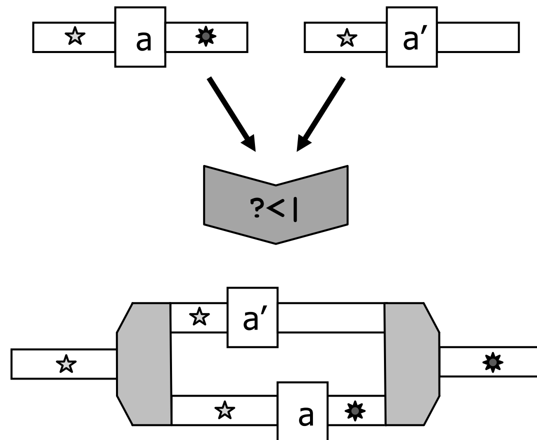
Con este operador es posible insertar aspectos antes del componente que no modifican su valor de entrada. Por ejemplo un aspecto de *log* para hacer seguimiento a la ejecución del programa.

- **Combinador `(?<|)`**

```
(?<|) :: Arrow a => a b c -> a b () -> a b c
a ?<| a’ = a’ ?>| a
```

`a ?<| a’` realiza la misma operación que el combinador anterior pero modificando el orden de los argumentos. De esta manera, el aspecto `a’` es insertado antes del componente `a`, pero sin que `a` utilice el resultado producido por `a’`. Gráficamente:

Figura 16. Combinador ?<|



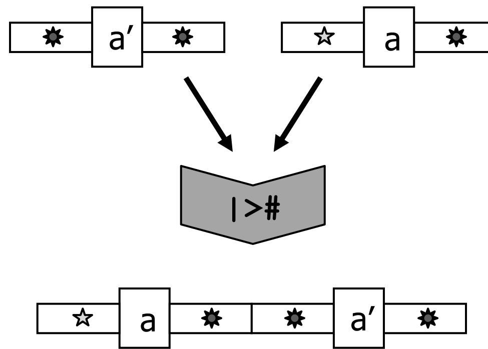
8.3.2 Aspectos después de componentes

- **Combinador** `(|>#)`

```
(|>#) :: Arrow a => a c c -> a b c -> a b c
a' |># a = proc b -> do
             c  <- a  -< b
             c' <- a' -< c
             returnA -< c'
```

`a' |># a` inserta el aspecto `a'` después del componente `a`, pasando el resultado de `a` como argumento de `a'`. Al aplicar `a'` sobre el valor de retorno de `a`, éste es modificado antes de ser puesto en la salida, de modo que el tipo de entrada y de salida de `a'` debe ser el mismo y debe coincidir con el tipo de salida de `a`. Gráficamente:

Figura 17. Combinador |>#



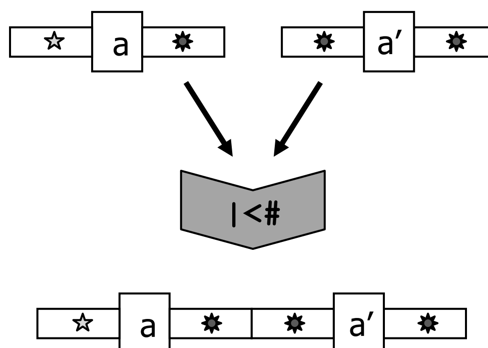
Este operador permite insertar aspectos después del componente que modifican su valor de salida. Por ejemplo un aspecto que tome el resultado del componente y lo restrinja de acuerdo a una política implementada en el aspecto.

- **Combinador (|<#)**

```
(|<#) :: Arrow a => a b c -> a c c -> a b c
a |<# a’ = a’ |># a
```

`a |<# a’` realiza la misma operación que el combinador anterior pero modificando el orden de los argumentos. De esta manera, el aspecto `a’` es insertado después del componente `a`. Gráficamente:

Figura 18. Combinador |<#

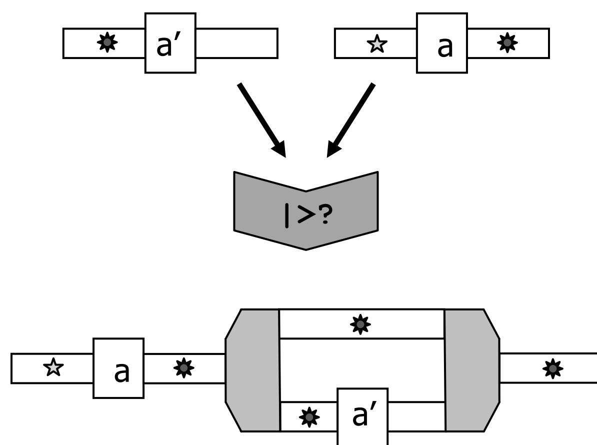


- **Combinador `(|>?)`**

```
(|>?) :: Arrow a => a c () -> a b c -> a b c
a' |>? a = proc b -> do
             c  <- a -< b
             c' <- a'  -< c
             returnA -< c
```

`a' |>? a` inserta el aspecto `a'` después del componente `a`, pasando el resultado de `a` como argumento de `a'` y como resultado de la combinación de ambos *Arrows*. En esta combinación, la aplicación de `a'` no modifica el valor de retorno de `a`, ya que éste es puesto directamente en la salida. De acuerdo a esto, el tipo de entrada de `a'` debe coincidir con el tipo de salida de `a`. Gráficamente:

Figura 19. Combinador |>?



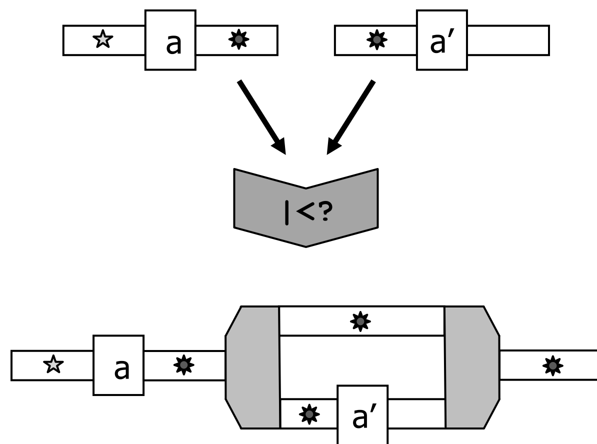
Este combinador es útil para insertar aspectos después de componentes sin alterar su valor de salida. Por ejemplo insertar un aspecto de persistencia después de la realización de una operación.

- **Combinador `(|<?)`**

```
(|<?) :: Arrow a => a b c -> a c () -> a b c
a |<? a′ = a′ |>? a
```

`a |<? a′` realiza la misma operación que el combinador anterior pero modificando el orden de los argumentos. De esta manera, el aspecto `a′` es insertado después del componente `a`, pero sin que `a′` modifique el resultado producido por `a`. Gráficamente:

Figura 20. Combinador |<?



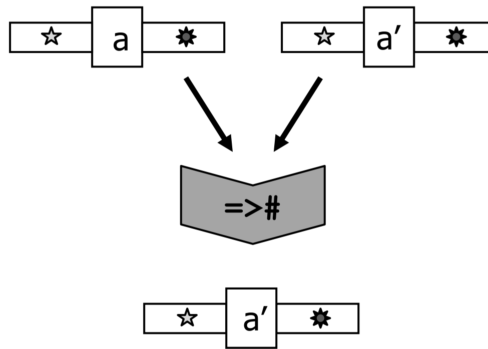
8.3.3 Aspectos en lugar de componentes

- **Combinador `(=>#)`**

```
(=>#) :: Arrow a => a b c -> a b c -> a b c
a =># a′ = a′
```

`a =># a'` remplaza el componente `a` con el aspecto `a′`, por lo que `a` y `a′` deben tener el mismo tipo de entrada y de salida. Gráficamente:

Figura 21. Combinador =>#

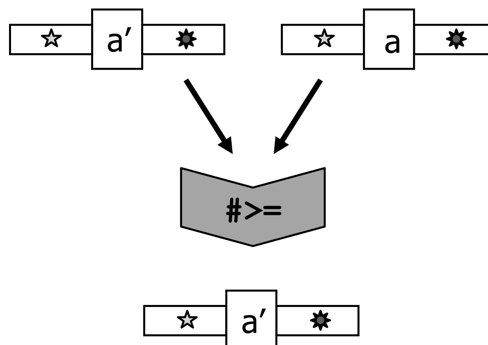


- **Combinador** **(#>=)**

```
(#>=) :: Arrow a => a b c -> a b c -> a b c
(#>=) = const
```

Al igual que el combinador anterior remplaza un componente por el aspecto indicado, pero modificando el orden de los argumentos: primero el aspecto y luego el componente. Gráficamente:

Figura 22. Combinador #>=



8. 4 EJEMPLO

La utilización del modelo diseñado se ilustrará partiendo de un ejemplo descrito en (Hughes, 2005) para contar el número de ocurrencias de una palabra en un archivo dado. El *Arrow* que realiza esta operación es el siguiente:

```
count w = Kleisli readFile >>> arr words >>> arr (filter
(==w)) >>> arr length >>> Kleisli print
```

Como puede observarse, el *Arrow* está formado por la unión en secuencia de varios *Arrows* con una función específica.

A continuación se definirán aspectos de seguridad depuración, *log* y restricción de valores.

8.4.1 Aspecto de seguridad. Este aspecto recibe como entrada la información de seguridad de los archivos (en `SecurityInfo`) y la ruta de un archivo para verificar su acceso. Si el acceso al archivo es denegado o desconocido se muestra un mensaje de error indicando el suceso.

```
data SecurityInfo = SI { allowedFiles :: [FilePath]
                       , deniedFiles  :: [FilePath]
                       }
                  deriving Show

data SecurityException = AccessDenied    String
                       | NoAccessGranted String
                       deriving (Show, Typeable)

instance Exception SecurityException
```

```
securityAspect :: SecurityInfo
               -> SingleNamedArrow (Kleisli IO) FilePath ()

securityAspect sinfo =
     SNA "readFile|writeFile"
     (Kleisli
     (\x -> if elem x (deniedFiles sinfo)
             then do hPutStr stderr
                     $ "Error: access denied"
                     ++x++ (show sinfo)
                   throw (AccessDenied $ show x)
             else if elem x (allowedFiles sinfo)
                  then return ()
                  else do hPutStr stderr
                          $ "Error: access not granted"
                          ++ (show sinfo)
                        throw (NoAccessGranted $ show x)))
```

8.4.2 Aspecto de depuración. Muestra en pantalla los datos recibidos como entrada.

```
debugAspect :: (Show a)
            => SingleNamedArrow (Kleisli IO) a ()

debugAspect =
      SNA "readFile|writeFile|words|filter|length|print"
      $ Kleisli (\x -> hPutStr stdout $ show x)
```

8.4.3 Aspecto de restricción de un valor. Verifica si el valor recibido es superior a 10, de ser así, lo modifica por el valor 5, sino lo deja sin cambios.

```
limitAspect :: SingleNamedArrow (Kleisli IO) Int Int
limitAspect = SNA "lenght" $ arr (\x -> if x > 10
                                  then 5
                                  else x)
```

8.4.4 Aspecto de log. Registra los datos que recibe como entrada en el archivo indicado como argumento.

```
logAspect :: (Show a) => FilePath
          -> SingleNamedArrow (Kleisli IO) a ()

logAspect f =
      SNA "readFile|print|filter|"
      (Kleisli (\x -> appendFile f (show x)))
```

8.4.5 Combinación de aspectos y componentes. Para insertar los aspectos definidos en el *Arrow* `count`, éste es redefinido como un *Arrow* nombrado, dando a cada uno de sus *Arrows* componentes el tipo SingleNamedArrow.

```
countNamed w = SNA "readFile" (Kleisli readFile)
               >>> SNA "words" (arr words)
               >>> SNA "filter" (arr (filter (==w)))
               >>> SNA "length" (arr length)
               >>> SNA "print" (Kleisli print)
```

Finalmente se combinan aspectos y componentes en el *Arrow* nombrado `countNamedA`, utilizando los combinadores de la clase `Aaop` definidos en `NamedArrow.hs`.

```
localFile = "TestNamedArrow.hs"
secInfo = SI [localFile] []
```

```
countNamedA f w =
       securityAspect secInfo  ?>| logAspect f
       ?>| SNA "readFile" (Kleisli readFile)

       >>> logAspect f ?>| SNA "words" (arr words)

       >>> logAspect f ?>| SNA "filter" (arr (filter(==w)))

       >>> limitAspect |># logAspect f ?>| SNA "length"
      (arr length)

       >>> logAspect f ?>| SNA "print" (Kleisli print)
```

En esta nueva versión del *Arrow* contador de palabras se tiene un aspecto de seguridad, al principio del *Arrow,* para verificar el acceso del archivo; también se incluye un aspecto de restricción de valores, después de contar el número de palabras del archivo, para limitar el valor de salida a 10. Además, cada paso de de la ejecución del *Arrow* es registrado en un archivo por el aspecto de *log*, que fue insertado antes de la aplicación de cada componente.

9. CONCLUSIONES

En este proyecto se realizó la implementación de un modelo que simula AOP, utilizando programación funcional en Haskell y sus mecanismos avanzados de programación. El modelo presentado no es una simulación completa de AOP, sólo se implementaron cuestiones relacionadas con puntos de unión y algunos elementos para la elaboración del tejido, que es realizado de forma manual.

Sin embargo, la implementación de estos puntos permitió una amplia profundización en los conceptos del lenguaje Haskell, principalmente en el manejo de *Arrows*, eje principal del modelo en el que se construyen los elementos del mismo y los operadores encargados de enlazarlos. Naturalmente, al utilizar *Arrows* se hace uso de otros conceptos importantes como tipos de datos algebraicos, necesarios para definir un tipo que se adaptara a las necesidades del modelo; polimorfismo, para la definición de tipos y funciones de amplia usabilidad; currificación, un mecanismo ampliamente utilizado para flexibilizar y simplificar la aplicación de funciones y; mónadas, para manejar todos los asuntos relacionados con entrada/salida.

Otro elemento importarte en el desarrollo del modelo fue el sistema de clases, que permite agrupar diferentes tipos de datos en grupos de tipos que tienen comportamiento similar, restringiendo el ámbito de aplicación de las operaciones definidas para brindar consistencia entre éstas y los elementos a los que son aplicadas. Un ejemplo de este concepto, también conocido como polimorfismo *ad-hoc* puede ser encontrado en la clase `Aaop`, en la que se definen los combinadores de tejido retomando combinadores previamente definidos y reutilizándolos para determinar el comportamiento de nuevos tipos, en este caso el tipo `SingleNamedArrow`. De esta forma, es posible definir nuevos tipos de datos que tengan este mismo comportamiento al incluirlos en la clase `Aaop`. Por ejemplo si se desea manejar un *Arrow* nombrado más

complejo que permita identificar el número de veces que el *Arrow* ha sido utilizado, se crea un tipo de dato algebraico `CountNamedArrow`:

```
data (Arrow a) => CountNamedArrow a b c = CNA
                  { named :: String
                  , count :: Int
                  , runCountNamedArrow :: a b c }
```

Luego se declara como instancia de la clase `Aaop` para darle acceso a los combinadores de tejido, lo que hace posible combinar `SingleNamedArrow` con `CountNamedArrow`.

Aunque el modelo desarrollado puede parecer simple a primera vista, en realidad es bastante complejo debido a la potencia del lenguaje Haskell. El modelo permite simular la mezcla de elementos que realiza la AOP, haciendo posible la construcción de sistemas complejos a partir de sistemas sencillos y previamente probados con la utilización de combinadores. Éstos permiten combinar aspectos y componentes de manera que formen un componente más complejo, que puede ser nuevamente combinado de la misma forma.

Como se mencionó anteriormente, para la implementación del modelo se utilizaron mecanismos avanzados de Haskell, incluyendo tipos de datos algebraicos, clases de tipos, polimorfismo, mónadas, y *Arrows* que facilitaron su construcción. Sin embargo, elementos más complejos de la AOP pueden ser construidos utilizando otras herramientas de Haskell (fuera del alcance de este proyecto) como: tipos de datos algebraicos genéricos, tipos de datos fantasmas (*phantom types*), tipos de datos existenciales y programación a nivel de tipos. Uno de esos elementos de AOP es el tejedor, que con los mecanismos descritos podría ser implementado como un combinador o una función que mezcle aspectos y componentes sin intervención del programador.

BIBLIOGRAFÍA

Armstrong, Joe. 2007. *Programming Erlang: software for a concurrent world.* s.l. : Pragmatic Bookshelf, 2007. 515 p.

Bird, Richard. 2000. *Introducción a la programación funcional con Haskell. 2 ed.* Madrid : Pearson Educación, 2000. 369 p.

Böllert, Kai. 1999. On Weaving Aspects. *Object-Oriented Technology ECOOP'99 Workshop Reader , Lisbon, Portugal.* Berlin : Springer-Verlag, 1999. Vol. 1743. p. 301-302.

Cardelli, Luca and Wegner, Peter. 1985. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR).* New York : ACM, 1985. Vol. 17, 4. p. 471-523.

Church, Alonzo. 1936. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics.* 1936. Vol. 58, 2. p. 345-363.

Curry, Haskell B. and Feys, Robert. 1958. *Combinatory Logic. Vol. 1.* Amsterdam : North-Holland, 1958.

Dijkstra, Edsger Wybe. 1976. *A Discipline of Programming.* Englewood : Prentice Hall, 1976. 217 p.

Elrad, Tzilla, Filman, Robert E. and Bader, Atef. 2001. Aspect-oriented programming: Introduction. *Comunications of ACM.* New York : ACM, 2001. Vol. 44, 10. p. 29-32.

Hall, Cordelia V., et al. 1996. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems (TOPLAS).* New York : ACM, 1996. Vol. 18, 2. p. 109-138.

HASKELL. 2006. Control.Arrow. [Online] 2006. http://www.haskell.org/ghc/docs/6.12.2/html/libraries/base-4.2.0.1/Control-Arrow.html.

—. 2006. Introduction. [Online] 2006. http://www.haskell.org/haskellwiki/Introduction.

—. 2006. Prelude: data Either a b. [Online] 2006. http://haskell.org/ghc/docs/6.12.1/html/libraries/base-4.2.0.0/Prelude.html#t%3AEither.

Hofer, Christian and Ostermann, Klaus. 2007. On the relation of aspects and monads. *FOAL '07: Proceedings of the 6th workshop on Foundations of aspect-oriented languages.* New York : ACM, 2007. p. 27-33.

Hudak, Paul. 1990. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys (CSUR).* New York : ACM, 1990. Vol. 21, 3. p. 359-411.

Hudak, Paul, et al. 2007. A history of Haskell: being lazy with class. *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages.* New York : ACM, 2007. p. 12-55.

Hudak, Paul, Fasel, Joseph and Peterson, John. 2000. A Gentle introduction to Haskell. Version 98. [Online] 2000. http://haskell.org/tutorial/.

Hughes, John. 2000. Generalising monads to arrows. *Science of Computer Programming.* Amsterdam : Elsevier North-Holland, Inc., 2000. Vol. 37, 1-3. p. 67--111.

—. 2005. Programming with Arrows. *Advanced Functional Programming.* Berlin : Springer-Verlag, 2005. Vol. 3622. p. 73-129.

—. 1989. Why functional programming matters. *he Computer Journal.* Oxford : Oxford University Press, 1989. Vol. 32, 1. p. 98-107.

Jones, Mark P. 1999. Typing Haskell in Haskell. *Proceedings of the 1999 Haskell Workshop, París. Published in Technical Report UU-CS-1999-28.* s.l. : Meijer, Erik (ed), 1999.

Kiczales, Gregor, et al. 1997. Aspect-Oriented Programming. *ECOOP'97 — Object-Oriented Programming 11th European Conference Jyväskylä, Finland.* Berlin : Springer-Verlag, 1997. Vol. 1241. p. 220-242.

Kiczales, Gregor, y otros. 1997. Aspect-Oriented Programming. *University of Maryland: Department of Computer Science.* [En línea] Junio de 1997. http://www.cs.umd.edu/class/spring2003/cmsc838p/Design/aop.pdf.

Louden, Kenneth C. 2004. Programación funcional. *Lenguajes de programación: Principios y práctica. 2 ed.* México : Thomson, 2004. p. 431-492.

Mac Lane, Saunders. 1998. Categories for the Working Mathematician. 2nd ed. *Graduate Texts in Mathematics.* s.l. : Springer-Verlag, 1998. Vol. 5. 314 p.

McCarthy, John. 1978. History of LISP. *ACM SIGPLAN Notices.* New York : ACM, 1978. Vol. 13, 8. p. 217-223.

Mendel, Ricardo H., Ferreira Szpiniak, Ariel and Luna, Carlos D. 1998. *Manual - Guia de aprendizaje en programación avanzada.* Rio Cuarto : Editorial de la Fundación Universidad Nacional de Rio Cuarto, 1998. p. 91-128.

Mens, Kim, et al. 1998. Aspect-Oriented Programming Workshop Report. *Object-Oriented Technologys ECOOP'97 Workshop Reader Jyväskylä, Finland.* Berlin : Springer-Verlag, 1998. Vol. 1357. p. 483-496.

Meuter, Wolfgang de. 2007. Monads as a theoretical foundation for AOP. *ECOOP '97 Workshop on Aspect-Oriented Programming.* 2007.

Milner, Robin. 1978. A theory of type polymorphism in programming. *Journal of Computer and System Sciences.* 1978. Vol. 17. p. 348-375.

Milner, Robin, et al. 1997. *The definition of Standard ML. Revised edition.* Cambridge : MIT Press, 1997. 114 p.

Moggi, E. and Sabry, Amr. 2001. Monadic encapsulation of effects: a revised approach (extended version). *Journal of Functional Programming.* New York : Cambridge University Press, 2001. Vol. 11, 6. p. 591-627.

Moor, Oege de, Peyton Jones, Simon L. and Wyk, Eric Van. 2000. Aspect-Oriented Compilers. *GCSE '99: Proceedings of the First International Symposium on Generative and Component-Based Software Engineering.* Berlin : Springer-Verlag, 2000. p. 121-133.

Newbern, Jeff. 2006. All about Monads: A comprehensive guide to the theory and practice of monadic programming in Haskell. Version 1.1.0. [Online] 2006. http://www.haskell.org/all_about_monads/html/index.html.

Odersky, Martin, Spoon, Lex and Venners, Bill. 2008. *Programming in Scala. A comprehensive step-by-step guide.* California : Artima, 2008. 736 p.

Paterson, Ross. 2001. A new notation for Arrows. *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming.* Florence : ACM, 2001. p. 229-240.

Peyton Jones, Simon (ed.). 2003. Haskell 98 Language and Libraries: The Revised Report. [Online] 2003. http://www.haskell.org/onlinereport.

Pierce, Benjamin C. 2002. *Types and programming languages.* Cambridge : MIT Press, 2002. 623 p.

Plasmeijer, Rinus and Eekelen, Marko van. 2001. Clean Language Report 2.0. *University of Nijmegen.* [Online] 2001. http://clean.cs.ru.nl/CleanExtra/report20/index.html.

Rashid, A. and Blair, L. 2003. Editorial: Aspect-oriented Programming and Separation of Crosscutting Concerns. *The ComputerJournal.* 2003. Vol. 46, 5. p. 527-528.

Robinson, David. 2006. An Introduction to Aspect Oriented Programming in e. *Verilab.* [Online] 2006. http://www.verilab.com/files/sample_chapter_verilab_aop_cookbook.pdf.

Ruiz, Blas C., et al. 2004. *Razonando con Haskell. Un curso sobre programación funcional.* Madrid : Thomson, 2004. 511 p.

Schwanninger, Christa, Wuchner, Egon and Michael, Kircher. 2004. Encapsulating Crosscutting Concerns in System Software. *Proceedings of the Third AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software.* Lankaster : s.n., 2004. p. 9-14.

Smith, Joshua B. 2006. *Practical OCaml.* s.l. : Apress, 2006. 456 p.

Sulzmann, Martin and Wang, Meng. 2007. Aspect-oriented programming with type classes. *FOAL '07: Proceedings of the 6th workshop on Foundations of aspect-oriented languages.* New York : ACM, 2007. p. 65-74.

Thompson, Simon. 2008. CUFP 2008 Report. [Online] 2008. http://cufp.org/archive/2008/report.pdf.

—. 1999. *Haskell: The Craft of Functional Programming. 2 ed.* Boston : Addison - Wesley, 1999. 487 p.

—. 1995. *Miranda: The Craft of Functional Programming.* Boston : Addison - Wesley, 1995. 451 p.

Viera, Marcos, Swierstra, S. Doaitse and Swierstra, Wouter. 2009. Attribute grammars fly first-class: how to do aspect oriented programming in Haskell. *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming.* New York : ACM, 2009. p. 245-256.

Wadler, Philip and Blott, S. 1989. How to make ad-hoc polymorphism less ad hoc. *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* New York : ACM, 1989. p. 60-76.

Wadler, Philip. 1990. Comprehending monads. *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming.* New York : ACM, 1990. p. 61-78.

—. 1995. Monads for Functional Programming. *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text.* Berlin : Springer-Verlag, 1995. p. 24--52.

—. 1992. The essence of functional programming. *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* New York : ACM, 1992. p. 1-14.

Win, Bart de, et al. 2002. On the importance of the separation-of-concerns principle in secure software engineering. *Workshop on the Application of Engineering Principles to System Security Desig.* Boston : Applied Computer Security Associates (ACSA), 2002.