Facultad de Ciencias Exactas, Ingeniería y Agrimensura - UNR
Departamento de Ciencias de la Computación

Formalización de mónadas concurrentes en Agda: el caso de la mónada Delay

Tesina de Grado

Autora:
Bini, Valentina María
Legajo: B-5926/9

Director:
Rivas, Exequiel

Índice general

Intro

Parte I

Preliminares

Capítulo 1

Introducción a Agda

Agda es un lenguaje de programación funcional desarrollado inicialmente por Ulf Norell en la Universidad de Chalmers como parte de su tesis doctoral [Nor07] que se caracteriza por tener tipos dependientes. A diferencia de otros lenguajes donde hay una separación clara entre el mundo de los tipos y el de los valores, en un lenguaje con tipos dependientes estos universos están más entremezclados. Los tipos pueden contener valores arbitrarios (lo que los hace *depender* de ellos) y pueden aparecer también como argumentos o resultados de funciones.

El hecho de que los tipos puedan contener valores, permite que se puedan escribir propiedades de ciertos valores como tipos. Los elementos de estos tipos son pruebas de que la propiedad que representan es verdadera. Esto hace que los lenguajes con tipos dependientes puedan ser utilizados como una lógica. Esta fue la idea principal de la teoría de tipos desarrollada por Martin Löf, en la cual está basado el desarrollo de Agda. Una característica importante de esta teoría es su enfoque constructivista, en el cual para demostrar la existencia de un objeto debemos construirlo.

Para poder utilizar a Agda como una lógica se necesita que sea consistente, y es por eso que se requiere que todos los programas sean totales, es decir que no tienen permitido fallar o no terminar. En consecuencia, Agda incluye mecanismos que comprueban la terminación de los programas.

El objetivo de esta sección es presentar una introducción a Agda, haciendo énfasis en las características necesarias para exponer la temática de esta tesina.

1.1. Tipos de datos y *Pattern Matching*

Un concepto clave en Agda es el *pattern matching* sobre tipos de datos algebraicos. Al agregar los tipos dependientes el *pattern matching* se hace aún más poderoso. Se verá este tema más en detalle en la sección 1.2. Para comenzar, en esta sección se describirán las funciones y tipos de datos con tipos simples.

Los tipos de datos se definen utilizando una declaración data, en la que se especifica el nombre y el tipo del tipo de dato a definir, así como los constructores y sus respectivos tipos. En el siguiente código se puede ver una forma de definir el tipo de los booleanos:

```
data Bool : Set where
  true : Bool
  false : Bool
```

El tipo de Bool es Set, el tipo de los tipos pequeños. Las funciones sobre Bool pueden definirse por *pattern matching*:

```
not : Bool → Bool
not true = false
not false = true
```

Las funciones en Agda no tienen permitido fallar, por lo que una función debe cubrir todos los casos posibles. Esto será constatado por el *type checker*, el cual lanzará un error si hay casos no definidos.

Otro tipo de dato que puede ser útil son los números naturales.

```
data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ
```

La suma sobre números naturales puede ser definida como una función recursiva (también utilizando *pattern matching*).

```
_+_ : ℕ → ℕ → ℕ
zero + m = m
(suc n) + m = suc (n + m)
```

Para garantizar la terminación de la función, las llamadas recursivas deben ser aplicadas a argumentos más pequeños que los originales. En este caso, _+_ pasa el chequeo de terminación ya que el primer argumento se hace más pequeño en la llamada recursiva.

Si el nombre de una función contiene guiones bajos (_), entonces puede ser utilizado como un operador en el cual los argumentos se posicionan donde están los guiones bajos. En consecuencia, la función _+_ puede ser utilizada como un operador infijo escribiendo n + m en lugar de _+_ n m.

La precedencia y asociatividad de un operador se define utilizando una declaración infix. Para mostrar esto se agregará, además de la suma, una función producto (la cual tiene más precedencia que la suma). La precedencia y asociatividad de ambas funciones podrían escribirse de la siguiente manera:

```
infixl 2 _*_
infixl 1 _+_

_*_ : ℕ → ℕ → ℕ
zero * m = zero
suc n * m = m + n * m
```

La palabra clave infixl indica que se asocia a izquierda (de igual manera existe infixr para asociar a derecha o infix si no se asocia hacia ningún lado) y el número que sigue indica la precedencia, operadores con mayor número tendrán más precedencia que operadores con menor número.

1.1.1. Tipos de datos parametrizados

Los tipos de datos pueden estar parametrizados por otros tipos de datos. El tipo de las listas de elementos de tipo arbitrario se define de la siguiente manera:

```
infixr 1 _::_
data List (A : Set) : Set where
  [] : List A
  _::_ : A → List A → List A
```

En este ejemplo el tipo List está parametrizado por el tipo A, el cual define el tipo de dato que tendrán los elementos de las listas. List ℕ es el tipo de las listas de números naturales.

1.1.2. Patrones con puntos

En algunos casos, al definir una función por *pattern matching*, ciertos patrones de un argumento fuerzan que otro argumento tenga un único valor posible que tipe correctamente. Para indicar que el valor de un argumento fue deducido por chequeo de tipos y no observado por *pattern matching*, se le agrega delante un punto (.). Para mostrar un ejemplo de uso de un patrón con punto, se considerará el siguiente tipo de dato Square definido como sigue:

```
data Square : ℕ → Set where
  sq : (m : ℕ) → Square (m * m)
```

El tipo Square n representa una propiedad sobre el número n, la cual dice dicho número es un cuadrado perfecto. Un habitante de tal tipo es una prueba de que el número n efectivamente es un cuadrado perfecto. Si se quisiera definir entonces una función root que tome un natural y una prueba de que dicho natural es un cuadrado perfecto, y devuelva su raíz cuadrada, podría realizarse de la siguiente manera:

```
root : (n : ℕ) → Square n → ℕ
root .(m * m) (sq m) = m
```

Se puede observar que al *matchear* el argumento de tipo Square n con el constructor sq aplicado a un natural m, n se ve forzado a ser igual a m * m.

1.1.3. Patrones absurdos

Otro tipo de patrón especial es el patrón absurdo. Usar un patrón absurdo al definir una función significa que no es necesario dar una definición para ese caso ya que no hay manera alguna en la que alguien pudiera dar un argumento para la función. El tipo de dato definido a continuación será de utilidad para ver un ejemplo de este tipo de patrones:

```
data Even : ℕ → Set where
  even-zero : Even zero
  even-plus2 : {n : ℕ} → Even n → Even (suc (suc n))
```

El tipo Even n representa, al igual que Square n, una propiedad sobre n. En este caso la propiedad afirma que n es un número par. Un habitante de este tipo es una prueba de que dicha proposición se cumple.

Si se quisiera definir una función que, dado un número y una prueba de que es par, devuelva el resultado de dividirlo por dos, podría realizarse de la siguiente manera:

```
half : (n : ℕ) → Even n → ℕ
half zero even-zero = zero
half (suc zero) ()
half (suc (suc n)) (even-plus2 e) = half n e
```

Se puede ver que en el caso del 1, no existe una prueba de que ese número sea par, y por lo tanto no debemos dar una definición para ese caso. Requerir la pureba de paridad nos asegura que no hay riesgo de intentar dividir por dos un número impar.

1.1.4. El constructor `with`

A veces no alcanza con hacer *pattern matching* sobre los argumentos de una función, sino que se necesita analizar por casos el resultado de alguna computación intermedia. Para esto se utiliza el constructor with.

Si se tiene una expresión e en la definición de una función f, se puede abstraer f sobre el valor de e. Al hacer esto se agrega a f un argumento extra, sobre el cual se puede hacer *pattern matching* al igual que con cualquier otro argumento.

Para proveer un ejemplo de uso del constructor with, se definirá a continuación la relación de orden _<_ sobre los números naturales.

```
_<_ : ℕ → ℕ → Bool
_<_ zero _          = true
_<_ (suc _) zero = false
_<_ (suc x) (suc y) = x < y
```

Si se quisiera definir entonces, utilizando esta función, una función min que calcule el mínimo entre dos números naturales x e y, se debería analizar cuál es el resultado de calcular $x < y$. Esto se escribe haciendo uso del constructor with como sigue:

```
min : ℕ → ℕ → ℕ
min x y with x < y
min x y | true = x
min x y | false = y
```

El argumento extra que se agrega está separado por una barra vertical y corresponde al valor de la expresión $x < y$. Se puede realizar esta abstracción sobre varias expresiones a la vez, separándolas entre ellas mediante barras verticales. Las abstracciones with también pueden anidarse. En el lado izquierdo de las ecuaciones, los argumentos abstraídos con with deben estar separados también con barras verticales.

En este caso, el valor que tome $x < y$ no cambia nada la información que se tiene sobre los argumentos x e y, por lo que volver a escribirlos no es necesario, puede reemplazarse la parte izquierda por tres puntos como se muestra a continuación:

```
min₂ : ℕ → ℕ → ℕ
min₂ x y with x < y
... | true = x
... | false = y
```

1.2. Funciones dependientes

Como se mencionó anteriormente, una de las principales características de Agda es que tiene tipos dependientes. El tipo dependiente más básico de todos son las funciones dependientes, en las cuales el tipo del resultado depende del valor del argumento. En Agda se escribe $(x : \mathsf{A}) \to \mathsf{B}$ para indicar el tipo de una función que toma un argumento x de tipo A y devuelve un resultado de tipo B, donde x puede aparecer en B. Un caso especial de esto es cuando x es un tipo en sí mismo. Se podría definir, por ejemplo:

```
identity : (A : Set) → A → A
identity A x = x
```

```
zero' : ℕ
zero' = identity ℕ zero
```

identity es una función dependiente que toma como argumento un tipo A y un elemento de A y retorna dicho elemento. De esta manera se codifican las funciones polimórficas en Agda.

A continuación se muestra un ejemplo de una función dependiente menos trivial, la cual toma una función dependiente y la aplica a cierto argumento:

```
apply : (A : Set) (B : A → Set) → ((x : A) → B x) → (a : A) → B a
apply A B f a = f a
```

1.2.1. Argumentos Implícitos

Los tipos dependientes sirven para definir funciones polimórficas. Pero en los ejemplos provistos en la sección anterior se da de forma explícita el tipo al cual cierta función polimórfica se debe aplicar. Usualmente esto es diferente. En general se espera que el tipo sobre el cual se va a aplicar una función polimórfica sea inferido por el *type checker*. Para solucionar este problema, Agda utiliza un mecanismo de *argumentos implícitos*.

Para declarar un argumento implícito de una función, se utilizan llaves en lugar de paréntesis. $\{x : A\} \rightarrow B$ significa lo mismo que $(x : A) \rightarrow B$, excepto que cuando se utiliza una función de este tipo el verificador de tipos intenta inferir el argumento por su cuenta.

Con esta nueva sintaxis puede definirse una nueva versión de la función identidad, donde no es necesario explicitar el tipo argumento:

```
id : {A : Set} → A → A
id x = x

true' : Bool
true' = id true
```

Se puede observar que el tipo argumento es implícito tanto cuando la función se aplica como cuando es definida. No hay restricciones sobre cuáles o cuántos argumentos pueden ser implícitos, así como tampoco hay garantías de que estos puedan ser efectivamente inferidos por el *type checker*.

Para dar explícitamente un argumento implícito se usan también llaves. f $\{v\}$ asigna v al primer argumento implícito de f. Si se requiere explicitar un argumento que no es el primero, escribe f $\{x = v\}$, lo cual asigna v al argumento implícito llamado x. El nombre de un argumento implícito se obtiene de la declaración del tipo de la función.

Si se desea, por el contrario, que el verificador de tipos infiera un término que debería darse explícitamente, se puede reemplazar por un guión bajo. Por ejemplo:

```
one' : ℕ
one' = identity _ (suc zero)
```

1.3. Familias de Tipos de datos

Se definió en el apartado 1.1.1 el tipo de las listas de tipo arbitrario parametrizado por A. Estas listas pueden tener cualquier largo, tanto una lista vacía como una lista con un millón de elementos son de tipo List A. En ciertos casos es útil que el tipo restrinja el largo que tiene la lista, y es así como surgen las listas de largo definido, llamadas comúnmente vectores, que se definen como sigue:

```
data Vec (A : Set) : ℕ → Set where
  [] : Vec A zero
  _::_ : {n : ℕ} → A → Vec A n → Vec A (suc n)
```

El tipo de Vec A es $\mathbb{N} \to$ Set. Esto significa que Vec A es una familia de tipos indexada por los números naturales. Por lo tanto, para cada n natural, Vec A n es un tipo. Los constructores pueden construir elementos de cualquier tipo dentro de la familia. Hay una diferencia sustancial entre parámetros e índices de un tipo de dato. Se dice que Vec está parametrizado por un tipo A e indexado sobre los números naturales.

En el tipo del constructor _::_ se puede observar un ejemplo de una función dependiente. El primer argumento del constructor es un número natural n implícito, el cual es el largo de la cola. Es seguro poner n como argumento implícito ya que el verificador de tipos siempre podrá inferirlo en base al tipo del tercer argumento.

Lo que tienen de interesante las familias de tipos es lo que sucede cuando se usa *pattern matching* sobre sus elementos. Si se quisiera definir una función que devuelva la cabeza de una lista no vacía, el tipo Vec permite expresar el tipo de las listas no vacías, lo cual hace posible definir la función head de manera segura como se muestra a continuación:

```
head : {A : Set} {n : ℕ} → Vec A (suc n) → A
head (x :: xs) = x
```

La definición es aceptada por el verificador de tipos ya que, aunque no se da un caso para la lista vacía, es exhaustiva. Esto es gracias a que un elemento del tipo Vec A (suc n) sólo puede ser construído por el constructor _::_, y resulta útil ya que la función head no está correctamente definida para el caso de la lista vacía.

1.4. Sistema de Módulos

El objetivo del sistema de módulos de Agda es manejar el espacio de nombres. Un programa se estructura en diversos archivos, cada uno de los cuales tiene un módulo *top-level*, dentro del cual van todas las definiciones. El nombre del módulo principal de un archivo debe coincidir con el nombre de dicho archivo. Si se tiene, por ejemplo, un archivo llamado `Agda.agda`, al comienzo del archivo se debería encontrar la siguiente línea:

```
module Agda where
```

Dentro del módulo principal se pueden definir otros módulos. Esto se hace de la misma manera que se define el módulo *top-level*. Por ejemplo:

```
module Numbers where
        data Nat : Set where
```

```
    zero : Nat
    suc : Nat → Nat

  suc₂ : Nat → Nat
  suc₂ n = suc (suc n)
```

Para acceder a entidades definidas en otro módulo hay que anteponer al nombre de la entidad el nombre del módulo en el cual está definida. Para hacer referencia a Nat desde fuera del módulo Numbers se debe escribir Numbers.Nat:

```
one : Numbers.Nat
one = Numbers.suc Numbers.zero
```

La extensión de los módulos (excepto el módulo principal) se determina por indentación. Si se quiere hacer referencia a las definiciones de un módulo sin anteponer el nombre del módulo constantemente se puede utilizar la sentencia open, tanto localmente como en *top-level*:

```
two : Numbers.Nat
two = let open Numbers in suc one


open Numbers
two₂ : Nat
two₂ = suc one
```

Al abrir un módulo, se puede controlar qué definiciones se muestran y cuáles no, así como también cambiar el nombre de algunas de ellas. Para esto se utilizan las palabras clave using (para restringir cuáles definiciones traer), hiding (para esconder ciertas definiciones) y renaming (para cambiarles el nombre). Si se quisiera abrir el módulo Numbers ocultando la función suc_2 y cambiando los nombres del tipo y los constructores, se debería escribir:

```
open Numbers hiding (suc₂)
              renaming (Nat to natural; zero to z0; suc to successor)
```

1.4.1. Módulos Parametrizados

Los módulos pueden ser parametrizados por cualquier tipo de dato. En caso de que se quiera definir un módulo para ordenar listas, por ejemplo, puede ser conveniente asumir que las listas son de tipo A y que tenemos una relación de orden sobre A. A continuación se presenta dicho ejemplo:

```
module Sort (A : Set)(_<_ : A → A → Bool) where
  insert : A → List A → List A
  insert y [] = y :: []
  insert y (x :: xs) with x < y
  ... | true = x :: insert y xs
  ... | false = y :: x :: xs

  sort : List A → List A
```

```
sort [] = []
sort (x :: xs) = insert x (sort xs)
```

Cuando se mira desde afuera una función definida dentro de un módulo parametrizado, la función toma como argumentos, además de los propios, los parámetros del módulo. De esta manera se podría definir:

```
sort₁ : (A : Set) (_<_ : A → A → Bool) → List A → List A
sort₁ = Sort.sort
```

También pueden aplicarse todas las funciones de un módulo parametrizado a los parámetros del módulo de una vez instanciando el módulo de la siguiente manera:

```
module SortNat = Sort ℕ _<_
```

Esto crea el módulo SortNat que contiene las funciones insert y sort, las cuales ya no tienen como argumentos los parámetros del módulo Sort, sino que directamente trabajan con naturales y la relación sobre naturales $<$.

```
sort₂ : List ℕ → List ℕ
sort₂ = SortNat.sort
```

También se puede instanciar el módulo y abrir directamente el módulo resultante sin darle un nuevo nombre, lo cual se escribe de forma simplificada como sigue:

```
open Sort ℕ _<_ renaming (insert to insertNat; sort to sortNat)
```

1.4.2. Importando módulos desde otros archivos

Se describió hasta ahora la forma de utilizar diferentes módulos dentro de un archivo, el cual tiene siempre un módulo principal. Muchas veces, sin embargo, los programas se dividen en diversos archivos y uno se ve en la necesidad de utilizar un módulo definido en un archivo diferente al actual. Cuando esto sucede, se debe *importar* el módulo correspondiente.

Los módulos se importan por nombre. Si se tiene un módulo A.B.C en un archivo `/alguna/direccion/local/A/B/C.agda`, este se importa con la sentencia import A.B.C. Para que el sistema pueda encontrar el archivo, `/alguna/direccion/local` debe estar en el *path* de búsqueda de Agda.

Al importar módulos se pueden utilizar las mismas palabras claves de control de espacio de nombres que al abrir un módulo (using, hiding y renaming). Importar un módulo, sin embargo, no lo abre automáticamente. Se puede abrir de forma separada con una sentencia open o usar la forma corta open import A.B.C.

1.5. Records

Un tipo *record* se define de forma similar a un tipo *data*, donde en lugar de constructores se tienen campos, los cuales son provistos por la palabra clave field. Por ejemplo:

```
record Point : Set where
  field x : ℕ
        y : ℕ
```

Esto declara el registro Point con dos campos naturales x e y. Para construir un elemento de Point se escribe:

```
mkPoint : ℕ → ℕ → Point
mkPoint a b = record{ x = a; y = b }
```

Si antes de la palabra clave field se agrega la palabra clave constructor, se puede dar un constructor específico para el registro, el cual permite construir de manera simplificada un elemento del mismo.

```
record Point' : Set where
  constructor _,_
  field x : ℕ
        y : ℕ

mkPoint' : ℕ → ℕ → Point'
mkPoint' a b = a , b
```

Para poder extraer los campos de un *record*, cada tipo *record* viene con un módulo con el mismo nombre. Este módulo está parametrizado por un elemento del tipo y contiene funciones de proyección para cada uno de los campos. En el ejemplo de Point se obtiene el siguiente módulo:

```
- module Point (p : Point) where
- x : Nat
- y : Nat
```

Este módulo puede utilizarse como viene o puede instanciarse a un registro en particular.

```
getX : Point → ℕ
getX = Point.x

getY : Point → ℕ
getY p = let open Point p in y
```

Es posible agregar funciones al módulo de un *record* incluyéndolas en la declaración del mismo luego de los campos.

```
record Monad (M : Set → Set) : Set₁ where
  constructor makeMonad
  field
        return : {A : Set} → A → M A
        _»=_ : {A B : Set} → M A → (A → M B) → M B

  mapM : {A B : Set} → (A → M B) → List A → M (List B)
  mapM f [] = return []
  mapM f (x :: xs) = f x »= \y →
                     mapM f xs »= \ys →
```

```
                    return (y :: ys)

mapM' : {M : Set → Set} → Monad M → {A B : Set}
        → (A → M B) → List A → M (List B)
mapM' {M} Mon f xs = Monad.mapM {M} Mon f xs
```

Como se puede ver en este ejemplo, los *records* pueden ser, al igual que los *datatype*, parametrizados. En este caso, el *record* Monad está parametrizado por M. Cuando un *record* está parametrizado, el módulo generado por él tiene los parámetros del *record* como parámetros implícitos.

1.6. Características adicionales

Agda tiene soporte para definiciones mutuamente recursivas. Estas deben ser declaradas dentro de un bloque mutual:

```
mutual
  even : ℕ → Bool
  even zero = true
  even (suc n) = odd n

  odd : ℕ → Bool
  odd zero = false
  odd (suc n) = even n
```

1.7. Coinducción

Capítulo 2

Mónadas Concurrentes

2.1. Definiciones previas

2.1.1. Categorías

Una **categoría** $\mathscr{C}$ consiste de:

- una clase de **objetos**: **ob** $\mathscr{C}$;
- una clase de **morfismos** o **flechas**: **mor** $\mathscr{C}$;
- dos funciones de clase:
  - dom : **mor** $\mathscr{C} \to$ **ob** $\mathscr{C}$ (dominio),
  - $codom$: **mor** $\mathscr{C} \to$ **ob** $\mathscr{C}$ (codominio).

  Para cada par de objetos A, B en **ob** $\mathscr{C}$ se denomina $Hom(A, B)$ al conjunto de flechas o morfismos de A a B, es decir:

$$Hom(A, B) := \{f \in \mathbf{mor}\ \mathscr{C} : dom(f) = A, codom(f) = B\}$$

- Y para cada $A, B, C \in$ **ob** C una operación

$$\circ : Hom(A, B) \times Hom(B, C) \to Hom(A, C)$$

  llamada **composición** con las siguientes propiedades:

  - Se denota $\circ(f, g) = g \circ f$.
  - **Asociatividad**: para cada $A, B, C, D \in$ **ob** $\mathscr{C}$ y $f, g, h \in$ **mor** $\mathscr{C}$ tales que $f \in Hom(A, B)$, $g \in Hom(B, C)$ y $h \in Hom(C, D)$, $\quad h \circ (g \circ f) = (h \circ g) \circ f$.
  - Para cada $A \in$ **ob**$\mathscr{C}$ existe un **morfismo identidad** $id_A \in Hom(A, A)$ tal que
    - $\star$ $\forall B,\ \forall f \in Hom(A, B), f \circ id_A = f$,
    - $\star$ $\forall C,\ \forall g \in Hom(C, A, id_A \circ g = g$.

Ejemplos

Categoría Set La categoría **Set** es aquella tal que:

- **ob Set** = conjuntos
- **mor Set** = funciones.

Categoría $\mathbb{1}$ La categoría $\mathbb{1}$ es aquella tal que:

- **ob** $\mathbb{1} = \{\star\}$
- **mor** $\mathbb{1} = \{\mathrm{id}_\star\}$.

Objetos iniciales y terminales

Un objeto $\mathbf{0} \in \mathbf{ob}\mathscr{C}$ se dice **inicial** si $\forall A \in \mathbf{ob}\mathscr{C}, \exists! \mathbf{0} \to A$.

Un objeto $\mathbf{1} \in \mathbf{ob}\mathscr{C}$ se dice **terminal** si $\forall A \in \mathbf{ob}\mathscr{C}, \exists! A \to \mathbf{1}$.

Ejemplo En **Set** $\emptyset$ es el único objeto inicial y los conjuntos de un elemento $\{x\}$ son los objetos terminales.

2.1.2. Funtores

Sean $\mathscr{C}$ y $\mathscr{D}$ dos categorías. Un **funtor** $F : \mathscr{C} \to \mathscr{D}$ asigna:

- a cada objeto $A \in \mathbf{ob}\ \mathscr{C}$, un objeto $F(A) \in \mathbf{ob}\ \mathscr{D}$;
- a cada morfismo $f : A \to B \in \mathbf{mor}\ \mathscr{C}$, un morfismo $F(f) : F(A) \to F(B) \in \mathbf{mor}\ \mathscr{D}$ tal que:
  - para todo $A \in \mathbf{ob}\ \mathscr{C}$, $F(id_A) = id_{F(A)}$;
  - para todos $f, g \in \mathbf{mor}\ \mathscr{C}$ tales que tenga sentido la composición $g \circ f$, se tiene $F(g \circ f) = F(g) \circ F(f)$.

2.1.3. Transformaciones Naturales

Sean $F, G : \mathscr{C} \to \mathscr{D}$ dos funtores (entre las mismas categorías). Una **transformación natural** $\eta : F \to G$ asigna a cada $A \in \mathbf{ob}\ \mathscr{C}$ un morfismo $\eta_A : F(A) \to G(A)$ tal que para todo $f \in Hom(A, B)$ se cumple que:

$$\eta_B \circ F(f) = G(f) \circ \eta_A$$

2.1.4. Monoides

Un **monoide** es un conjunto M dotado de una operación asociativa $M \times M \to M$, $(m, n) \to mn$ tal que existe un elemento neutro:

$$\exists e \in M, \forall m \in M, (em = me = m).$$

2.2. Introducción a las mónadas

Se considerarán dos variantes de la definición de mónadas. La primera es la definición clásica y la segunda define a una mónada como un sistema de extensión o 3-tupla Kleisli. La primera es muy utilizada en la literatura ya que es la definición matemática y está definida en torno a transformaciones naturales, pero la segunda es más fácil de utilizar desde una perspectiva computacional. Como ambas definiciones son equivalentes [Mog91], se utilizará una u otra según sea conveniente.

2.2.1. Definición clásica de Mónadas

Dada una categoría $\mathscr{C}$, una mónada sobre $\mathscr{C}$ es una tupla (T, μ, η), donde:

- $T : \mathscr{C} \to \mathscr{C}$ es un funtor,
- $\eta : Id \to T$ y $\mu : T \cdot T \to T$ son transformaciones naturales

- y se cumplen las siguientes identidades:

$$\mu_X \circ T\mu_X = \mu_X \circ \mu_{TX}, \qquad \mu_X \circ T\eta_X = id_{TX}, \qquad \mu_X \circ \eta_{TX} = id_{TX}$$

Ejemplos de mónadas sobre la categoría Set

Mónada *Error* Sea $T : \mathbf{Set} \to \mathbf{Set}$ el funtor $TX = X + E$, donde E es un conjunto de errores. Intuitivamente un elemento de TX puede ser un elemento de X (un valor) o un error perteneciente a E. Luego se definen η y μ como siguen:

- Para cada conjunto X, se define $\eta_X : IdX \to TX$ como $\eta_X(x) = inl(x)$.
- Para cada conjunto X, se define $\mu_X : TTX \to TX$ como $\mu_X(inl(tx)) = tx$ si $tx \in X + E$ y $\mu_X(inr(e)) = inr(e)$ si $e \in E$. Es decir que si se tiene un error se propaga el error y si se tiene un elemento de TX se devuelve dicho elemento.

Mónada *State* Sea $T : \mathbf{Set} \to \mathbf{Set}$ el funtor $TX = (X \times S)^S$, donde S es un conjunto no vacío de estados. Intuitivamente, TX es una computación que toma un estado y retorna el valor resultante junto con el estado modificado. Luego se definen η y μ como sigue:

- Para cada conjunto X, se define $\eta_X : IdX \to TX$ como $\eta_X(x) = (\lambda s : S.\langle x, s\rangle)$.
- Para cada conjunto X, se define $\mu_X : TTX \to TX$ como $\mu_X(f) = (\lambda s : S.\ \mathrm{let}\ \langle f', s'\rangle = f(s)\ \mathrm{in}\ f'(s'))$, es decir que $\mu_X(f)$ es la computación que, dado un estado s, primero computa el par computación-estado $f(s) = \langle f', s'\rangle$ y luego retorna el par valor-estado $f'(s') = \langle x, s''\rangle$.

2.2.2. Definición alternativa de Mónadas

Una **3-tupla Kleisli** sobre una categoría $\mathscr{C}$ es una tupla $(T, \eta, \_^*)$, donde

- $T : \mathbf{ob}\,\mathscr{C} \to \mathbf{ob}\,\mathscr{C}$,
- para cada $A \in \mathbf{ob}\,\mathscr{C}$, $\eta_A : A \to TA$,
- para cada $f : A \to TB$, $f^* : TA \to TB$,
- y se cumplen las siguientes ecuaciones:
  - $\eta_A^* = id_{TA}$
  - $\eta_A; f^* = f$ para cada $f : A \to TB$
  - $f^*; g^* = (f; g^*)^*$ para cada $f : A \to TB$ y $g : B \to TC$.

Intuitivamente η_A es la inclusión de valores en computaciones (lo que en programación funcional usualmente se conoce como *return*) y f^* es la extensión de una función f de valores a computaciones a una función de computaciones a computaciones, la cual primero evalúa una computación y luego aplica f al valor resultante (lo que generalmente se conoce como *bind* o $\gg\!=$).

Ejemplos definidos como 3-tupla Kleisli

Mónada *Error* Tomando el funtor descripto en la versión clásica:

- Para cada conjunto X, se define $\eta_X : IdX \to TX$ como $\eta_X(x) = inl(x)$ al igual que en la versión clásica.
- Para cada función $f : X \to TY$, se define $f^* : TX \to TY$ como $f^*(inl(x)) = f(x)$ si $x \in X$ y $f^*(inr(e)) = inr(e)$ si $e \in E$.

Mónada *State* Tomando el funtor descripto en la versión clásica:

- Para cada conjunto X, se define $\eta_X : IdX \to TX$ como $\eta_X(x) = (\lambda s : S.\langle x, s\rangle)$ al igual que en la primera versión.
- Para cada función $f : X \to TY$, se define $f^* : TX \to TY$ como $f^*(g) = (\lambda s : S.\ \text{let}\ \langle x, s'\rangle = g(s)\ \text{in}\ f(x)(s'))$.

2.2.3. Fortaleza de funtores y mónadas

Un **funtor** $F : \mathscr{C} \to \mathscr{C}$ es **fuerte** si viene equipado con una transformación natural $\sigma_{X,Y} : FX \times Y \to F(X \times Y)$, de manera que se cumplen las siguientes ecuaciones:

$$\pi_1 = F(\pi_1) \circ \sigma_{X,\mathbf{1}}, \quad \sigma \circ (\sigma \times \text{id}) \circ \alpha = F(\alpha) \circ \sigma$$

donde π_1 y π_2 son las proyecciones del producto cartesiano y $\alpha = \langle\langle \pi_1, \pi_1 \circ \pi_2\rangle, \pi_2 \circ \pi_2\rangle$ representa su asociatividad.

Una **mónada** (T, μ, η) sobre $\mathscr{C}$ es **fuerte** si el funtor subyacente T es fuerte y la fortaleza es compatible con μ y η:

$$\eta_{A\times B} = \sigma_{A,B} \circ (\eta_A \times \text{id}), \quad \sigma_{A,B} \circ (\mu_A \times \text{id}) = \mu_{A\times B} \circ T\sigma_{A,B} \circ \sigma_{TA,B}$$

Hay una definición similar de fortaleza $\bar{\sigma}_{X,Y} : X \times FY \to F(X \times Y)$ que actúa sobre el lado derecho, pero como el producto cartesiano es simétrico, se puede obtener de la fortaleza izquierda como $\bar{\sigma} = F\gamma \circ \sigma \circ \gamma$, donde $\gamma = \langle \pi_2, \pi_1\rangle$ intercambia los elementos del producto cartesiano.

2.2.4. Estructuras monoidales

Un **funtor monoidal** es un funtor $F : \mathscr{C} \to \mathscr{C}$ equipado con una estructura monoidal (m, e), donde $m : FX \times FY \to F(X \times Y)$ es una transformación natural y $e : \mathbf{1} \to F\mathbf{1}$ es un morfismo tal que los diagramas de coherencia estándar conmutan. Además, si la estructura monoidal es compatible con γ, entonces el funtor monoidal es simétrico.

Una **mónada monoidal** es una mónada (T, μ, η) que tiene una estructura monoidal (m, e) en su funtor subyacente T tal que $e = \eta_{\mathbf{1}}$ y las estructuras monoidal y monádica son compatibles:

$$\eta_{A\times B} = m_{A,B} \circ (\eta_A \times \eta_B), \quad m_{A,B} \circ (\mu_A \times \mu_B) = \mu_{A\times B} \circ Tm_{A,B} \circ m_{TA\times TB}.$$

2.2.5. Mónadas conmutativas

Dada una mónada fuerte (T, μ, η), T como funtor puede ser equipado con dos estructuras monoidales canóncas:

$$\begin{array}{ll} \phi : TA \times TB \to T(A \times B) & \psi : TA \times TB \to T(A \times B) \\ \phi = \mu \circ T\bar{\sigma} \circ \sigma & \psi = \mu \circ T\sigma \circ \bar{\sigma} \end{array}$$

y $e = \eta_1 : \mathbf{1} \to T\mathbf{1}$ en ambos casos.

Se dice que una mónada es **conmutativa** cuando estas dos estructuras coinciden.

En esta tesina se utilizará la categoría **Set**, la cual es la categoría de conjuntos y funciones, y las mónadas que se presenten serán sobre esta categoría. El objeto terminal

$\mathbf{1} = \{\star\}$ es un conjunto unitario. Una consecuencia particular de esto es que cualquier funtor F y mónada (T, μ, η) sobre esta categoría son fuertes, y cada uno admite una única fortaleza posible σ $(\bar{\sigma})$.

Por ejemplo, la mónada del conjunto partes $\mathcal{P}$ (y su variante finita $\mathcal{P}_f$) tiene fortaleza $\sigma(X, y) = X \times \{y\}$. En general, la fórmula de fortaleza de un funtor sobre **Set** puede ser expresada como $\sigma(v, y) = F(\lambda x : X.(x, y))(v)$. Cuando la mónada es conmutativa, hay sólo una estructura monoidal posible. En consecuencia, si una mónada es monoidal entonces es conmutativa [Koc70].

Parte II

Formalización de Mónadas Concurrentes

Bibliografía

Bibliografía

[Koc70] Kock, Anders: *Strong functors and monoidal monads.* Archiv der Mathematik, 23:113–120, 1970.

[Mog91] Moggi, Eugenio: *Notions of computation and monads.* Inf. Comput., 93(1):55–92, 1991.

[Nor07] Norell, Ulf: *Towards a practical programming language based on dependent theory.* Tesis de Doctorado, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.