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

I Preliminares **3**

1. Introducción a Agda **5**

1.1. Tipos de datos y *Pattern Matching* 5

1.2. Funciones dependientes 8

1.3. Familias de Tipos de datos 10

1.4. Sistema de Módulos 10

1.5. Records 12

1.6. Características adicionales 14

2. Mónadas Concurrentes **17**

2.1. Teoría de Categorías 17

2.2. Introducción a las mónadas 20

2.3. Mónadas Concurrentes 23

3. La Mónada *Delay* **29**

3.1. Introducción a la Coinducción 29

3.2. Coinducción en Agda 31

3.3. Mónada *Delay* 33

II Formalización de Mónadas Concurrentes **35**

Bibliografía **37**

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 simples (se profundizará esto en la sección 1.6.1). 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. $\{x : A\} \to B$ significa lo mismo que $(x : A) \to B$, 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 : ℕ
  y : ℕ
```

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

1.6.1. Universos

La paradoja de Russell implica que la colección de todos los conjuntos no es en sí misma un conjunto. Si existiera tal conjunto U, entonces uno podría formar el subconjunto $A \subseteq U$ de todos los conjuntos que no se contienen a sí mismos. Luego se deduciría que $A \in A \iff A \notin A$, lo cual es una contradicción.

Por razones similares, no todos los tipos de Agda son de tipo Set. Por ejemplo, se tiene que Bool : Set y Nat : Set, pero no es cierto que Set : Set. Sin embargo, es necesario y conveniente que Set tenga un tipo, es por eso que en Agda se le da el tipo Set_1:

$\mathsf{Set} : \mathsf{Set}_1$

Las expresiones de tipo Set_1 se comportan en gran medida como las de tipo Set, por ejemplo, pueden ser utilizadas como tipo de otras cosas. Sin embargo, los habitantes de Set_1 son potencialmente *más grandes*. Cuando se tiene $\mathsf{A} : \mathsf{Set}_1$, entonces se dice a veces que A es un *conjunto grande*. Sucesivamente, se tiene que:

$\mathsf{Set}_1 : \mathsf{Set}_2$

$\mathsf{Set}_2 : \mathsf{Set}_3$

etcétera. Un tipo cuyos habitantes son tipos se llama **universo**. Agda provee un número infinito de universos Set, Set_1, Set_2, Set_3, ..., cada uno de los cuales es un habitante del siguiente. Set es en sí mismo una abreviación de Set_0. El subíndice n es el **nivel** del universo Set_n. Agda provee también un tipo primitivo especial Level, cuyos habitantes son los posibles niveles de los universos. De hecho, la notación Set_n es una abreviación para Set n, donde n : Level.

Si bien no hay un número de niveles específico, se sabe que existe un nivel más bajo lzero, y que para cada nivel n existe algún nivel mayor lsuc n. Por lo tanto, el conjunto de niveles es infinito. Además, puede tomarse la cota superior mínima (o supremo) $n \sqcup m$ de dos niveles. En resumen, las siguientes operaciones son las únicas operaciones que Agda provee sobre niveles:

lzero : Level

lsuc : (n : Level) → Level

_⊔_ : (n m : Level) → Level

1.6.2. Inducción-Recursión

Una característica fundamental de Agda que la distingue de otros lenguajes similares es el soporte para *inducción-recursión* (en inglés *induction-recursion*). En la teoría de tipos intuicionista, la inducción-recursión es una propiedad que permite declarar simultáneamente un tipo y una función sobre dicho tipo, haciendo posible la creación de tipos más grandes que los tipos inductivos como, por ejemplo, los universos. En una definición inductiva, se dan reglas para generar habitantes de un tipo y luego pueden definirse funciones de ese tipo por inducción en dichos habitantes. En inducción-recursión se permite que las reglas que generan los habitantes de un tipo hagan referencia a la función que a su vez es definida por inducción en los habitantes del tipo. A continuación se muestra un ejemplo de una definición inductiva-recursiva para ilustrar mejor esta característica.

```
mutual
  data U : Set where
    sig : (A : U) → (El A → U) → U
    pi : (A : U) → (El A → U) → U

  El : U → Set
  El (sig A B) = Σ (El A) (λ a → El (B a))
  El (pi A B) = (a : (El A)) → (El (B a))
```

Como se ve en el ejemplo, la manera de hacer una declaración inductiva-recursiva es mediante la palabra clave mutual. Esta palabra clave puede utilizarse también para realizar definiciones de funciones mutuamente recursivas.

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

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

1.6.3. Coinducción

Agda tiene varios soportes diferentes para coinducción. Se describirán algunos de ellos más adelante en la sección 3.2, junto con sus características principales y su utilidad.

Capítulo 2

Mónadas Concurrentes

En este capítulo se hará una introducción teórica sobre mónadas concurrentes. Se definirán al comienzo algunos conceptos previos de la teoría de categorías sobre la cual se definen las mónadas. En la segunda sección se hará una introducción a las mónadas en sí mismas, en la cual se darán varias definiciones y ejemplos. Por último, en la tercera, se introducirán los conceptos específicos necesarios para definir una mónada concurrente.

2.1. Teoría de Categorías

La teoría de categorías fue desarrollada por Eilenberg y MacLane [EM45] en 1945. Esta teoría busca axiomatizar de forma abstracta diversas estructuras matemáticas como una sola, tales como los grupos y los espacios topológicos, mediante el uso de objetos y morfismos. A su vez, esta axiomatización se realiza de una manera nueva sin incluir las nociones de elemento o pertenencia, es decir, sin utilizar conjuntos. Con el concepto de categoría se pretende capturar la esencia de una clase de objetos matemáticos que se relacionan entre sí mediante aplicaciones, poniendo énfasis en la relación entre los objetos y no en la pertenencia como en la teoría de conjuntos.

Definición 2.1 (Categoría)**.** 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:

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

- Y para cada $A, B, C \in$ **ob** $\mathscr{C}$ 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

$\star$ $\forall B,\ \forall f \in Hom(A,B),\ f \circ id_A = f$,
$\star$ $\forall C,\ \forall g \in Hom(C,A),\ id_A \circ g = g$.

A continuación se presentan algunos ejemplos de categorías que pueden ser de utilidad para comprender mejor el concepto.

Ejemplo 2.2 (Categoría **Set**)**.** La categoría **Set** es aquella tal que:

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

Ejemplo 2.3 (Categoría $\mathbb{1}$)**.** La categoría $\mathbb{1}$ es aquella tal que:

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

Dentro de los objetos de una categoría, hay dos clases especiales de objetos: iniciales y terminales. Estos se definen como sigue:

Definición 2.4 (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 2.5. En **Set**, $\emptyset$ es el único objeto inicial y los conjuntos de un elemento $\{x\}$ son los objetos terminales.

En la categoría **Set**, se sabe que el producto cartesiano entre dos objetos (conjuntos) $A \times B$ es el conjunto de los pares (a,b) tales que $a \in A$ y $b \in B$. Para definir el concepto de producto cartesiano entre dos objetos A y B de una categoría cualquiera, es necesario caracterizar a $A \times B$ sin hacer referencia a sus elementos.

Definición 2.6 (Producto)**.** El **producto** de dos objetos A y B en una categoría $\mathscr{C}$ es una terna $(A \times B, \pi_A, \pi_B)$ donde:

- $\pi_A \in Hom(A \times B, A)$,
- $\pi_B \in Hom(A \times B, B)$
- y para todo objeto C y para todo par de morfismos $f : C \to A$, $g : C \to B$, existe un único mofismo $\langle f, g\rangle : C \to A \times B$ tal que:
  - $f = \pi_A \circ \langle f, g\rangle$
  - $g = \pi_B \circ \langle f, g\rangle$

Suponiendo que existen los productos $A \times B$ y $C \times D$ y que se tienen dos morfismos $f : A \to C$ y $g : B \to D$, se puede definir un morfismo $f \times g : A \times B \to C \times D$ tal que $f \times g = \langle f \circ \pi_A, g \circ \pi_B\rangle$.

De forma dual a la definición de producto, se puede definir la noción de coproducto entre dos objetos A y B de una categoría arbitraria como sigue:

Definición 2.7 (Coproducto)**.** El **coproducto** de dos objetos A, B de una categoría $\mathscr{C}$ es una terna $(A + B, \iota_A, \iota_B)$ donde:

- $\iota_A \in Hom(A, A + B)$,
- $\iota_B \in Hom(B, A + B)$
- y para todo objeto C y para todo par de morfismos $f : A \to C$, $g : B \to C$ existe un único morfismo $[f, g] : A + B \to C$ tal que se cumplen las siguientes

ecuaciones:
- $f = [f, g] \circ \iota_A$
- $g = [f, g] \circ \iota_B$

Una última relación que puede establecerse entre dos objetos de una categoría es el exponencial. En **Set**, el exponencial de dos conjuntos A y B es el conjunto B^A de todas las funciones que van de A en B, es decir que toman un elemento de A y devuelven un elemento de B. Esta noción puede generalizarse a una categoría arbitraria que tenga productos cartesianos.

Definición 2.8 (Exponencial)**.** Sea $\mathscr{C}$ una categoría con productos binarios y sean $A, B \in \mathbf{ob}\ \mathscr{C}$. Un objeto B^A es un **exponencial** si existe un morfismo $\varepsilon : B^A \times A \to B$ tal que para todo morfismo $g : C \times A \to B$ existe un único morfismo $\tilde{g} : C \to B^A$ tal que $g = \varepsilon \circ (\tilde{g} \times id_A)$.

Si se quisiera construir una categoría cuyos objetos son categorías, se necesitaría contar con morfismos entre categorías. Estos existen y se llaman funtores, son en cierta manera una generalización del concepto de función de conjuntos para categorías. Un funtor permite construir una nueva categoría a partir de otra dada.

Definición 2.9 (Funtor)**.** 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 que $F(g \circ f) = F(g) \circ F(f)$.

Se dice que un funtor es un **endofuntor** si la categoría de salida y la de llegada son la misma, es decir, $F : \mathscr{C} \to \mathscr{C}$.

Siguiendo con la misma lógica, uno podría construir morfismos entre funtores. Es decir, algún tipo de construcción matemática que lleve de un funtor dado a otro. Este concepto se denomina transformación natural y se define como sigue:

Definición 2.10 (Transformación Natural)**.** 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:

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

Para cerrar la sección, se introducirá la noción de monoide. Los monoides son un tipo de estructura algebraica abstracta introducida por primera vez por Arthur Cayley [].

Definición 2.11 (Monoide)**.** 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:

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

El elemento neutro de un monoide es único. Por esa razón, en general el elemento neutro es considerado una constante, es decir, una operación 0-aria (sin argumentos). Se utilizará esta representación en la formalización de los monoides.

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

Se define a continuación el concepto de mónada de la manera clásica dentro de la teoría de categorías.

Definición 2.12 (Mónada)**.** 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:

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

A continuación se presentan algunos ejemplos de mónadas clásicas que son ampliamente utilizadas en computación.

Ejemplo 2.13 (Mónada *Error*)**.** Sea T : **Set** $\to$ **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 pertenenciente 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.

Ejemplo 2.14 (Mónada *State*)**.** Sea T : **Set** $\to$ **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 $\mu_X(f) = (\lambda s : S.$ let $\langle f', s'\rangle = f(s)$ 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

Se define ahora la noción de sistema de extensión, también llamado 3-tupla Kleisli. Esta definición también parte de la teoría de categorías pero no es la más utilizada en

la literatura. Sin embargo, como se explica más adelante, es la que más se acerca a la forma de utilizar las mónadas en los lenguajes de programación funcional.

Definición 2.15 (Sistema de extensión)**.** Un **sistema de extensión** 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:
  - $\eta_A^* = id_{TA}$
  - $f^* \circ \eta_A = f$ para cada $f : A \to TB$
  - $g^* \circ f^* = (g^* \circ f)^*$ 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 que va de valores a computaciones a una función que va 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 $\gg\!=$).

A continuación se muestra cómo quedan definidos los ejemplos vistos para la definición clásica como sistemas de extensión para que se comprenda mejor el paralelismo entre ambas definiciones.

Ejemplo 2.16 (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 $e \in E$.

Ejemplo 2.17 (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. Funtores, mónadas y producto cartesiano

La fortaleza de los funtores es una forma de compatibilidad entre funtores y productos. En adelante se trabajará con funtores y mónadas que son fuertes respecto del producto cartesiano. A continuación se definen las nociones de funtor fuerte y mónada fuerte.

Definición 2.18 (Funtor fuerte)**.** 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:

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

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.

Definición 2.19 (Mónada fuerte)**.** 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:

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

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.

Otra forma en la que un funtor puede ser compatible con el producto cartesiano es si es un funtor monoidal. A continuación se definen los conceptos de funtor monoidal y mónada monoidal.

Definición 2.20 (Funtor monoidal)**.** 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 las siguientes ecuaciones se cumplen:

$$\pi_1 = F(\pi_1) \circ m_{A,\mathbf{1}} \circ (id_{FA} \times e), \qquad \pi_2 = F(\pi_2) \circ m_{\mathbf{1},A} \circ (e \times id_{FA}),$$

$$F(\alpha) \circ m_{X\times Y,Z} \circ (m_{X,Y} \times id_{FZ}) = m_{X,Y\times Z} \circ (id_{FX} \times m_{Y,Z}) \circ \alpha$$

Además, si la estructura monoidal es compatible con γ, entonces el funtor monoidal es simétrico.

Definición 2.21 (Mónada monoidal)**.** 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:

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

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

$$\phi : TA \times TB \to T(A \times B) \qquad \psi : TA \times TB \to T(A \times B)$$
$$\phi = \mu \circ T\bar{\sigma} \circ \sigma \qquad \psi = \mu \circ T\sigma \circ \bar{\sigma}$$

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].

2.3. Mónadas Concurrentes

La teoría de concurrencia está compuesta por una amplia variedad de modelos basados en diferentes conceptos. Hoare et al. [HHM+11] se plantearon si es posible tener un tratamiento comprensible de la concurrencia en el cual la memoria compartida, el pasaje de mensajes y los modelos de intercalación e independencia de computaciones puedan ser vistos como parte de la misma teoría con el mismo núcleo de axiomas. Con esta motivación crearon un modelo simple de concurrencia basado en estructuras algebraicas, dos de las cuales resultan interesantes para este trabajo: bimonoides ordenados y monoides concurrentes. Más tarde, Rivas y Jaskelioff [RJ19] extendieron este modelo al nivel de funtores y mónadas, dando lugar a las mónadas concurrentes. En las siguientes secciones se detallarán las características principales de cada uno de estos modelos.

2.3.1. Ley de intercambio

Ya estaba establecido que la composición secuencial y concurrente son estructuras monoidales, donde la concurrencia es además conmutativa. La pregunta que surge luego es cómo estas operaciones se relacionan entre sí. Se podría pensar en un principio que la ley de intercambio $(p * r); (q * s) = (p; q) * (r; s)$ de 2-categorías o bi-categorías debería cumplirse. Sin embargo, la presencia de esta ley implicaría que ambas estructuras monoidales coinciden, derivando en que las operaciones de secuenciación y concurrencia son la misma. Esto se puede ver aplicando el argumento Eckmann-Hilton.

Teorema 2.22 (argumento Eckmann-Hilton)**.** *Sea X un conjunto con dos operaciones binarias $;$ y $*$ tal que $e_;$ es el elemento neutro de $;$, e_* es el elemento neutro de $*$ y la ley de intercambio $(a * b); (c * d) = (a; c) * (b; d)$ se cumple. Entonces, ambas operaciones $;$ y $*$ coinciden, y ambas son conmutativas y asociativas.*

Demostración. Primero se muestra que ambos elementos neutros coinciden:

$$e_; = e_;; e_; = (e_* * e_;); (e_; * e_*) = (e_*; e_;) * (e_;; e_*) = e_* * e_* = e_*$$

Como los neutros coinciden, se lo puede llamar simplemente e. Se muestra ahora que ambas operaciones coinciden:

$$a; b = (e * a); (b * e) = (e; b) * (a; e) = b * a = (b; e) * (e; a) = (b * e); (e * a) = b; a$$

Usando el mismo argumento se puede ver también que la operación es conmutativa. La prueba de asociatividad es análoga. □

Como solución a esto, surge la idea de considerar un orden en los procesos, de manera que pueda debilitarse la ley de intercambio. En [HMSW11] se introduce una generalización del álgebra de Kleene para programas secuenciales [Koz94], llamada Álgebra Concurrente de Kleene. Esta es un álgebra que mezcla primitivas de composición concurrente $(*)$ y secuencial $(;)$, cuya característica principal es la presencia de una versión ordenada de la ley de intercambio de 2-categorías o bi-categorías.

$$(p * r); (q * s) \sqsubseteq (p; q) * (r; s)$$

Esta ley intuitivamente tiene sentido, por ejemplo, en un modelo de concurrencia de intercalación. Si se tiene una traza $t = t_1; t_2$ donde t_1 es una intercalación de dos trazas t_p y t_r, y t_2 de t_q y t_s, entonces t es también una intercalación de $t_p; t_q$ y $t_r; t_s$.

2.3.2. Dos modelos

Se introducirán a continuación dos modelos utilizados por Hoare et al. [HHM+11] para desarrollar su teoría, los cuales servirán de ejemplo en las secciones que siguen.

Modelo de Trazas Sea A un conjunto. Luego $Trazas_A$ es el conjunto de secuencias finitas de elementos de A. El conjunto partes $\mathcal{P}(Trazas_A)$ será el conjunto soporte del modelo. Se tienen las siguientes operaciones binarias sobre $\mathcal{P}(Trazas_A)$:

1. $T_1 * T_2$ es el conjunto de las intercalaciones de las trazas de T_1 y T_2.
2. $T_1; T_2$ es el conjunto de las concatenaciones entre las trazas de T_1 y T_2.

El conjunto $\{\epsilon\}$ funciona como elemento neutro para ambas operaciones ; y $*$, donde ϵ es la secuencia vacía. El orden está dado por la inclusión de conjuntos.

Modelo de Recursos Sea $(\Sigma, \bullet, u)$ un monoide parcial conmutativo, dado por una operación parcial binaria $\bullet$ y elemento neutro u. La igualdad significa que ambos lados están definidos y son iguales, o que ninguno está definido. El conjunto partes $\mathcal{P}(\Sigma)$ tiene una estructura de monoide ordenado conmutativo $(*, \text{emp})$ definido por:

$$p * q = \{\sigma_0 \bullet \sigma_1 | (\sigma_0, \sigma_1) \in dom(\bullet) \wedge \sigma_0 \in p \wedge \sigma_1 \in q\}$$
$$emp = \{u\}$$

El conjunto de funciones monótonas $\mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ es el conjunto soporte del modelo. Estas funciones representan transformadores de predicados. Las operaciones se definen mediante las siguientes fórmulas, donde F_i itera sobre los transformadores de predicados e Y_i se utiliza para iterar sobre los subconjuntos de Σ:

$$(F * G)Y = \bigcup \{FY_1 * GY_2 | Y_1 * Y_2 \subseteq Y\}$$
$$\texttt{nothing}\ Y = Y \cap \text{emp}$$
$$(F; G)Y = F(G(Y))$$
$$\texttt{skip}\ Y = Y$$

La idea es que se comienza con una postcondición Y, luego se la separa en dos afirmaciones separadas Y_1 e Y_2 y se aplica la regla de concurrencia hacia atrás para obtener una precondición $FY_1 * GY_2$ para la composición paralela de F y G. Se realilza la unión de todas estas descomposiciones de manera de obtener la precondición más débil posible.

El orden del modelo está dado por el orden reverso punto a punto, es decir $F \sqsubseteq G$ significa que $\forall X \subseteq \mathcal{P}(\Sigma), FX \supseteq GX$. Según esta definición, el elemento más pequeño es la función $\lambda X.\Sigma$, la cual se corresponde con el transformador de precondición más débil para la divergencia.

2.3.3. Monoides concurrentes

Como se va a utilizar un orden combinado con estructuras alegbraicas, se necesita una noción de compatibilidad de las operaciones con el orden. Sea $(A, \sqsubseteq)$ un orden parcial, entonces una operación $\oplus : A \times A \to A$ es *compatible* con el orden si $a \sqsubseteq b$ y $c \sqsubseteq d$ implica que $a \oplus c \sqsubseteq b \oplus d$. Se define primero una aproximación a la noción de monoide concurrente, el cual tiene dos estructuras monoidales y un orden compatible con ellas, pero no incluye ninguna relación especial entre ellas.

Definición 2.23 (Bimonoide ordenado)**.** Un **bimonoide ordenado** es un conjunto parcialmente ordenado $(A, \sqsubseteq)$ junto con dos estructuras monoidales $(A, *, \texttt{nothing})$ y $(A, ;, \texttt{skip})$ tal que $*$ y ; son compatibles con $\sqsubseteq$ y $*$ es conmutativa.

Podría ser tentador requerir que ambos elementos neutros de un bimonoide ordenado sean iguales, pero, por ejemplo, en el Modelo de Recursos no lo son. El Modelo de Recursos es un ejemplo de un bimonoide ordenado que no es un monoide concurrente. A continuación se define la noción de monoide concurrente.

Definición 2.24 (Monoide concurrente)**.** Un **monoide concurrente** es un bimonoide ordenado tal que los neutros coinciden, es decir $\texttt{nothing} = \texttt{skip}$, y la siguiente ley de intercambio se cumple:

$$(a * b); (c * d) \sqsubseteq (a; c) * (b; d)$$

En esta estructura no hay reducción de la operación $*$ a una intercalación del operador ;. El orden une a ambas estructuras sin reducir una a la otra. En el Modelo de Trazas ambos elementos neutros coinciden y la ley de intercambio se cumple, por lo que, además de bimonoide ordenado, es un monoide concurrente.

2.3.4. Generalización a nivel de funtores y mónadas

Para elevar la teoría descripta en la sección anterior al nivel de funtores, se modela el operador de secuenciación ; como la multiplicación monádica y su elemento neutro `skip` como la unidad monádica. Sólo faltan tres elementos: la operación de mezcla $*$, su elemento neutro `nothing` y el orden $\sqsubseteq$. La operación de mezcla se modela como una familia de transformaciones naturales $m_{A,B} : TA \times TB \to T(A \times B)$. Como se necesita que $*$ sea un monoide conmutativo, se requiere que T tenga una estructura de funtor monoidal simétrica, incluyendo un morfismo unidad $e : \mathbf{1} \to T\mathbf{1}$ que representa `skip`.

En cuanto al orden, lo que se necesita es un orden sobre computaciones que mida el grado de secuencialidad en ellas. Esto es una estructura de orden $\sqsubseteq_I$ sobre TI para cada conjunto I, sujeto a algún tipo de compatibilidad con el resto de la estructura que modela computaciones. Se define a continuación la noción de funtor ordenado, la cual establece una condición de compatibilidad de un funtor con un orden parcial, siguiendo la idea de Hughes y Jacobs [HJ04]:

Definición 2.25 (Funtor ordenado)**.** Un **orden** para un **funtor** F es una asignación de un orden parcial $\sqsubseteq_I$ en FI para cada conjunto I, tal que para cada morfismo $f : I \to J$, el morfismo $Ff : FI \to FJ$ es una función monótona respecto de $\sqsubseteq_I$ y $\sqsubseteq_J$.

A continuación se define la compatibilidad para la estructura de mónada. Para esto se utiliza una familia de estructuras ordenadas, lo cual es una instancia de lo que definieron Katsumata y Sato como una mónada ordenada [KS13].

Definición 2.26 (Mónada ordenada)**.** Un **orden** para una **mónada** (T, μ, η) es una asignación de un orden $\sqsubseteq_I$ sobre TI para cada conjunto I, tal que la categoría Kleisli de T se enriquece en la categoría de órdenes con el orden correspondiente a $\mathcal{K}\ell(T)(A, B)$ definido por $f \sqsubseteq_{A,B} g$ si y sólo si $\forall a : \mathbf{1} \to A, f \circ a \sqsubseteq_B g \circ a$.

Esta noción de orden sólo se relaciona con la estructura monádica. En comparación

a los bimonoides ordenados, se corresponde con la condición de que ; sea compatible con $\sqsubseteq$. La compatibilidad entre $*$ y $\sqsubseteq$ se postula como sigue:

Definición 2.27 (Funtor monoidal ordenado)**.** Sea T un funtor con una estructura monoidal (m, e) y una asignación de un orden $\sqsubseteq_I$ sobre TI para cada conjunto I. Se dice que el orden es compatible con (m, e) si $v \sqsubseteq_A v'$ y $w \sqsubseteq_B w'$ implican que $m \circ \langle v, w\rangle \sqsubseteq_{A\times B} m \circ \langle v', w'\rangle$ para cada $v, v' : \mathbf{1} \to TA$ y $w, w' : \mathbf{1} \to TB$.

A partir de todas estas definiciones previas se definen las variantes monádicas de los bimonoides ordenados y monoides concurrentes como sigue:

Definición 2.28 (Mónada monoidal ordenada)**.** Una mónada $(T, \eta, \_^*)$ es una mónada monoidal ordenada si está dotada de una estructura de funtor monoidal simétrico

$$m_{X,Y} : TX \times TY \to T(X \times Y), \qquad e : \mathbf{1} \to T\mathbf{1}$$

y una relación de orden $\sqsubseteq$ sobre T compatible con (m, e).

Aclaración: se escribe $f \star g$ para representar $m \circ (f \times g)$.

Definición 2.29 (Mónada concurrente)**.** Una mónada monoidal ordenada T es una mónada concurrente si $e = \eta_{\mathbf{1}} : \mathbf{1} \to T\mathbf{1}$ y se cumple la siguiente ley de intercambio:

$$(h \star i) \bullet (f \star g) \sqsubseteq (h \bullet f) \star (i \bullet g)$$

Esta definición puede probarse mostrando que las mónadas conmutativas (como los monoides conmutativos) son mónadas concurrentes (como los monoides concurrentes).

Ejemplo 2.30. Sea (T, μ, η) una mónada conmutativa. Entonces T tiene una única estructura de mónada monoidal $(m, \eta_{\mathbf{1}})$, y se puede definir la estructura de orden por el orden diagonal. La ley de intercambio se reduce a las condiciones de mónada monoidal.

Rivas y Jaskelioff [RJ19] también muestran que estas estructuras al nivel de mónada efectivamente generalizan aquellas al nivel de los monoides probando el siguiente lema.

Lema 2.31. *Sea (T, μ, η) una mónada monoidal ordenada (mónada concurrente). Entonces $T\mathbf{1}$ es un bimonoide ordenado (monoide concurrente).*

Como en el caso de los monoides, hay dos estructuras de bimonoide ordenado sobre $T\mathbf{1}$, que resultan de la simetría de los axiomas de los bimonoides ordenados. En esta estructura, como en los monoides, se puede también ir en la dirección contraria: dado un bimonoide ordenado, este puede ser elevado a una mónada monoidal ordenada.

Lema 2.32. *Sea $(A, \sqsubseteq, *, \texttt{nothing}, ;, \texttt{skip})$ un bimonoide ordenado. Este puede convertirse en una mónada monoidal ordenada con funtor soporte $T_A X = A \times X$, operaciones y orden. Más aún, si $\texttt{nothing} = \texttt{skip}$ (es decir que es un monoide concurrente), entonces $e = \eta_{\mathbf{1}}$ (es decir que es una mónada concurrente).*

Este resultado será utilizado más adelante en este trabajo para dar una representación alternativa de la mónada delay.

El ejemplo del Modelo de Trazas puede ser generalizado a una mónada concurrente parametrizando el conjunto sobre el cual se toman las trazas.

Ejemplo 2.33. La mónada $Tr_L(X) = \mathcal{P}_f(Trazas_{L\times X})$ es concurrente. La estructura de orden se define como la inclusión de conjuntos al igual que antes.

Capítulo 3

La Mónada *Delay*

Como se discutió previamente, la teoría de tipos de Martin-Löf es un lenguaje de programación funcional rico con tipos dependientes y, a su vez, un sistema de lógica constructiva. Sin embargo, esto trae una limitación respecto de los lenguajes de programación funcional estándar, ya que obliga a que todas las computaciones deban terminar. Esta restricción tiene dos razones principales: hacer que el chequeo de tipos de los tipos dependientes sea decidible y representar pruebas como programas (una prueba que no termina es inconsistente).

El tipo de dato *delay* fue introducido por Capretta [Cap05] con el objetivo de facilitar la representación de la no-terminación de funciones en la teoría de tipos de Martin-Löf. Lo que busca es explotar los tipos coinductivos para modelar computaciones infinitas. Los habitantes del tipo *delay* son valores "demorados", los cuales pueden no terminar y, por lo tanto, no retornar un valor nunca. El tipo de dato *delay* es una mónada y constituye una alternativa constructiva de la mónada *maybe*.

Se introducirá primero la noción de coinducción, junto con los soportes para coinducción que Agda proporciona. Luego se presentará la definición de la mónada *delay* y sus principales características.

3.1. Introducción a la Coinducción

El principio de inducción está bien establecido en el área de las matemáticas y las ciencias de la computación. En esta última, se utiliza principalmente para razonar sobre tipos de datos definidos inductivamente, tales como listas finitas, árboles finitos y números naturales. La coinducción es el pincipio dual de la inducción y puede ser utilizado para razonar sobre tipos de datos definidos coinductivamente, tales como flujos de datos, trazas infinitas o árboles infinitos, pero no está tan difundido ni se comprende tan bien en general.

Para ilustrar mejor el concepto de coinducción, se utilizarán algunos ejemplos presentados por Kozen y Silva [KS17] con el objetivo de promover este principio como una herramienta útil y hacerlo tan familiar e intuitivo como la inducción.

A continuación se considera el ejemplo del tipo `Lista de` A de listas finitas sobre un alfabeto A, definido inductivamente por:

- `nil` $\in$ `Lista de` A
- si $a \in A$ y $\ell \in$ `Lista de` A, entonces a `::` $\ell \in$ `Lista de` A.

El tipo de dato definido es la solución mínima a la ecuación:

$$\texttt{Lista de}\ A = \texttt{nil} + A \times \texttt{Lista de}\ A \tag{3.1}$$

Es decir que es el mínimo conjunto tal que se cumplen las condiciones listadas más arriba. Esto significa que uno puede definir funciones con dominio `Lista de` A de

manera única por inducción estructural. El tipo de las listas finitas e infinitas sobre A se define coinductivamente como la solución máxima de la ecuación 3.1. Esto significa que es el máximo conjunto tal que se cumplen ambas condiciones.

Formalmente, el tipo de las listas finitas sobre A es un álgebra inicial para una signatura que consiste en una constante (`nil`) y un constructor binario (`::`). El tipo de las listas finitas e infinitas sobre A forman la coálgebra final de la signatura (`nil`, `::`). Se definen a continuación los conceptos de álgebra, coálgebra, álgebra inicial y coálgebra final:

Definición 3.1 (Álgebra de un funtor)**.** Dado un endofuntor F sobre una categoría $\mathscr{C}$, un **álgebra** de F es un objeto X de $\mathscr{C}$ junto con un morfismo $\alpha : FX \to X$. Dadas dos álgebras $(X, \alpha : FX \to X)$, $(Y, \beta : FY \to Y)$ de F, $m : X \to Y$ es un morfismo de álgebras si se cumple la siguiente ecuación:

$$m \circ \alpha = \beta \circ F(m)$$

Las álgebras de F junto con sus morfismos forman una categoría llamada F-álgebras.

Definición 3.2 (Coálgebra)**.** Una **coálgebra** para un endofuntor F sobre una categoría $\mathscr{C}$ es un objeto A junto con un morfismo $u : A \to FA$. Dadas dos coálgebras $(A, \eta : A \to FA)$, $(B, \theta : B \to FB)$, $f : A \to B$ es un morfismo de coálgebras si respeta la estructura coalgebraica:

$$\theta \circ f = F(f) \circ \eta$$

Las coálgebras de F junto con sus morfismos generan una categoría llamada F-coálgebras.

Definición 3.3 (Álgebra inicial)**.** Un **álgebra inicial** para un endofuntor F sobre una categoría $\mathscr{C}$ es un objeto inicial en la categoría de las F-álgebras.

Definición 3.4 (Coálgebra final)**.** Una **coálgebra final** para un endofuntor F sobre una categoría $\mathscr{C}$ es un objeto terminal en la categoría de las F-coálgebras.

Formalmente, los tipos coinductivos se definen como elementos de una coálgebra final para un endofuntor dado en la categoría **Set**.

Ejemplo 3.5 (Flujo de datos infinitos)**.** El conjunto A^ω de flujos de datos (o *streams* en inglés) infinitos sobre un alfabeto A es (el conjunto soporte de) la coálgebra final del funtor $FX = A \times X$.

Ejemplo 3.6 (Cadenas infinitas)**.** El conjunto A^∞ de las cadenas finitas e infinitas sobre un alfabeto A es (el conjunto soporte de) la coálgebra final del funtor $FX = \mathbb{1} + A \times X$.

Mientras que los tipos inductivos se definen mediante sus constructores, los tipos coinductivos usualmente se presentan junto con sus destructores. Por ejemplo, los flujos de datos o *streams* admiten dos operaciones $hd : A^\omega \to A$ y $tl : A^\omega \to A^\omega$, los cuales representan la función *head* que devuelve el primer elemento del *stream* y la

función *tail* que devuelve la cola del *stream*. La existencia de los destructores es una consecuencia del hecho de que A^ω es una coálgebra para el funtor $FX = A \times X$. Todas estas coálgebras vienen equipadas con una función estructural $\langle obs, cont \rangle : X \to A \times X$; para A^ω se tiene que $obs = hd$ y $cont = tl$.

Las pruebas por coinducción tienen un paso coinductivo (análogo al paso inductivo) pero no caso base. Aunque esto parezca incorrecto o genere cierta desconfianza en dichas pruebas, cualquier dificultad que haga que la propiedad a demostrar no se cumpla se manifiesta en el intento de probar el paso coinductivo.

3.2. Coinducción en Agda

Se describirán a continuación los dos soportes de coinducción en Agda que se utilizaron en esta Tesina. El primero se basa en una notación particular, la notación musical, la cual permite manejar términos potencialmente infinitos. A pesar de ser una notación práctica e intuitiva, este soporte tiene algunos problemas con el chequeo de terminación de Agda, lo que limita bastante las propiedades que pueden demostrarse usándolo. Es por eso que se utilizó luego otro enfoque, basado en tipos de tamaño limitado (*sized types* en inglés), el cual ayuda al chequeo de terminación de Agda haciendo un seguimiento de la profundidad de las estructuras de datos mediante la definición de límites en la profundidad.

3.2.1. Notación Musical

Para mostrar la notación musical se utilizará como ejemplo el conjunto de los números *conaturales*. Así como las coálgebras son el dual de las álgebras, los números conaturales son el dual de los números naturales y se definen en Agda como sigue:

```
data Coℕ : Set where
  zero : Coℕ
  suc : ∞ Coℕ → Coℕ
```

El operador *delay* (∞) se utiliza para etiquetar ocurrencias coinductivas. El tipo ∞ A se interpretea como una computación suspendida o demorada de tipo A. Este operador viene equipado con funciones *delay* y *force*:

```
♯_ : ∀ {a} {A : Set a} → A → ∞ A
♭ : ∀ {a} {A : Set a} → ∞ A → A
```

La función *delay* (♯_) toma un valor de tipo A y lo devuelve suspendido dentro de un valor de tipo ∞ A. Por el contrario, la función *force* (♭), toma un valor de tipo ∞ A y lo desencapsula devolviendo un valor de tipo A.

Los valores de tipos coinductivos pueden ser construidos usando corecursión, la cual no debe necesariamente terminar, pero sí ser productiva. Por ejemplo, el infinito puede ser difinido como se muestra a continuación.

```
inf : Coℕ
inf = suc (♯ inf)
```

Como aproximación a la productividad, en el chequeo de terminación se pide que en la definición de funciones corecursivas las llamadas recursivas aparezcan bajo la

aplicación directa de un construcor coinductivo. Esta restricción en general hace que programar con tipos coinductivos sea incómodo, y es por eso que se buscan técnicas alternativas para asegurar que las definiciones corecursivas estén bien definidas.

3.2.2. *Sized Types*

Agda tiene un soporte nativo para *sized types*. Estos son tipos que cuentan con un índice que indica el número de desencapsulamientos que pueden realizarse sobre los habitantes de este tipo. Estos índices, llamados tamaños o *sizes*, asisten al chequeo de terminación evaluando que las definiciones corecursivas estén bien definidas.

En Agda existe un tipo Size de tamaños y un tipo Size< i cuyos habitantes son los tamaños estrictamente menores a i. Si se tiene un tamaño j : Size< i, este es forzado a ser j : Size. La relación de orden de los tamaños es transitiva, lo que implica que si se tiene que j : Size< i y k : Size< j, entonces k : Size< i. La relación de orden es, además, bien fundada, lo cual se usa para definir funciones corecursivas productivas. Existe también una operación sucesor de tamaños ↑ y un tamaño "infinito" ∞ tal que para cada tamaño i, i : Size< ∞. Finalmente, un *sized type* es un tipo indexado por Size.

Para ejemplificar el uso de los *sized types*, se definen a continuación los conaturales utilizando esta técnica.

```
mutual

  data Conat (i : Size) : Set where
    zero : Conat i
    suc : Conat′ i → Conat i

  record Conat′ (i : Size) : Set where
    coinductive
    field
      force : {j : Size< i} → Conat j

open Conat′ public
```

Ambos tipos, Conat y Conat′ están indexados por un tamaño i. Este índice debe entenderse como una cota superior del número de veces que puede aplicarse force. Más precisamente, cuando se aplica force a un n' : Conat′ i el valor resultante es es un n : Conat j de una profundidad estrictamente menor $j < i$. Un caso especial es el valor $\infty n'$: Conat′ ∞ de índice infinito, cuyo resultado de aplicar force es ∞n : Conat ∞, el cual también tiene índice infinito. De esta manera los tamaños establecen productividad en las definiciones recursivas. Al final, sólo interesan los valores n : Conat ∞ de índice infinito.

Si una función corecursiva en Conat i sólo se llama a sí misma con índices menores $j < i$, se garantiza la productividad y, por lo tanto, está bien definida. En la siguiente definición del valor infty se muestran los argumentos implícitos de tamaño explícitamente de manera que se evidencie cómo aseguran la productividad:

```
infty : ∀ {i} → Conat′ i
force (infty {i}) {j} = suc (infty {j})
```

3.3. Mónada *Delay*

El tipo de dato *delay* consituye una mónada fuerte, lo cual hace posible manejar computaciones que posiblemente no terminen como si fuera cualquier otra computación con efectos laterales. A continuación se presenta su definición formal siguiendo el estilo utilizado en [CUV19].

Definición 3.7 (*Delay*)**.** Sea X un tipo. Cada habitante de $\mathbf{D}X$ es una computación posiblemente infinita que, si termina, retorna un valor de tipo X. Se define $\mathbf{D}X$ como un tipo coinductivo mediante las siguientes reglas:

$$\frac{}{\texttt{now}\ x : \mathbf{D}X} \qquad \frac{c : \mathbf{D}X}{\texttt{later}\ c : \mathbf{D}X}$$

Sea R una relación de equivalencia sobre X. La relación se eleva a una relación de equivalencia $\sim_R$ sobre $\mathbf{D}X$ que se denomina R-bisemejanza fuerte (R-*strong bisimilarity* en inglés).

Definición 3.8 (R-bisemejanza fuerte)**.** Dada una relación de equivalencia R sobre X, se define la relación $\sim_R$ sobre $\mathbf{D}X$ coinductivamente mediante las siguientes reglas:

$$\frac{p : x_1\ R\ x_2}{\texttt{now}_\sim\ p : \texttt{now}\ x_1 \sim_R \texttt{now}\ x_2} \qquad \frac{p : c_1 \sim_R c_2}{\texttt{later}_\sim\ p : \texttt{later}\ c_1 \sim_R \texttt{later}\ c_2}$$

La $\equiv$-bisemejanza fuerte se denomina simplemente bisemejanza fuerte y se denota $\sim$.

En algunos casos, uno está interesado en la terminación de las computaciones y no exactamente en el tiempo exacto en el cual terminan. Es deseable entonces tener una relación que considere iguales dos computaciones si terminan con valores iguales, aunque una tarde más en terminar que la otra. Es decir, que identifique computaciones que sólo difieren en una cantidad finita de aplicaciones del constructor `later`. Esta relación se llama R-bisemejanza débil (R-*weak bisimilarity* en inglés) y se define en términos de *convergencia*. Esta última es una relación binaria entre $\mathbf{D}X$ y X que relaciona computaciones que terminan con sus valores de terminación.

Definición 3.9 (Convergencia)**.** La relación de **convergencia** denotada con $\downarrow$ entre $\mathbf{D}X$ y X se define inductivamente mediante las siguientes reglas:

$$\frac{p : x_1 \equiv x_2}{\texttt{now}_\downarrow\ p : \texttt{now}\ x_1 \downarrow x_2} \qquad \frac{p : c \downarrow x}{\texttt{later}_\downarrow\ p : \texttt{later}\ c \downarrow x}$$

Definición 3.10 (R-bisemejanza débil)**.** Dada una relación de equivalencia R sobre X, se define la relación $\approx_R$ sobre $\mathbf{D}X$ coinductivamente mediante las siguientes reglas:

$$\frac{p_1 : c_1 \downarrow x_1 \quad p_2 : x_1\ R\ x_2 \quad p_3 : c_2 \downarrow x_2}{\downarrow_\approx\ p_1\ p_2\ p_3 : c_1 \approx_R c_2} \qquad \frac{p : c_1 \approx_R c_2}{\texttt{later}_\approx\ p : \texttt{later}\ c_1 \approx_R \texttt{later}\ c_2}$$

La $\equiv$-bisemejanza débil se denomina simplemente bisemejanza débil y se denota $\approx$. En este caso, se modifica el primer constructor por simplicidad:

$$\frac{p_1 : c_1 \downarrow x \quad p_2 : c_2 \downarrow x}{\downarrow_{\approx} \; p_1 \; p_2 : c_1 \approx c_2}$$

El tipo *delay* $\mathbf{D}$ es una mónada fuerte. La unidad η es el constructor `now`, mientras que la multiplicación μ es la "concatenación" de constructores `later`:

$$\begin{aligned}
&\mu : \mathbf{D}(\mathbf{D}X) \to \mathbf{D}X \\
&\mu \; (\texttt{now} \; c) = c \\
&\mu \; (\texttt{later} \; c) = \texttt{later} \; (\mu \; c)
\end{aligned}$$

Parte II

Formalización de Mónadas Concurrentes

Bibliografía

Bibliografía

[Cap05] Capretta, Venanzio: *General Recursion via Coinductive Types.* Logical Methods in Computer Science, Volume 1, Issue 2, Julio 2005.

[CUV19] Chapman, James, Tarmo Uustalu y Niccolò Veltri: *Quotienting the Delay Monad by Weak Bisimilarity.* Mathematical Structures in Computer Science, 29(1):67–92, 2019.

[EM45] Eilenberg, Samuel y Saunders MacLane: *General Theory of Natural Equivalences.* Transactions of the American Mathematical Society, 58(2):231–294, 1945.

[HHM+11] Hoare, C. A. R., Akbar Hussain, Bernhard Möller, Peter W. O'Hearn, Rasmus Lerchedahl Petersen y Georg Struth: *On Locality and the Exchange Law for Concurrent Processes.* En Katoen, Joost Pieter y Barbara König (editores): *CONCUR 2011 - Concurrency Theory - 22nd International Conference, CONCUR 2011, Aachen, Germany, September 6-9, 2011. Proceedings*, volumen 6901 de *Lecture Notes in Computer Science*, páginas 250–264. Springer, 2011.

[HJ04] Hughes, Jesse y Bart Jacobs: *Simulations in coalgebra.* Theoretical Computer Science, 327(1):71–108, 2004, ISSN 0304-3975. Selected Papers of CMCS '03.

[HMSW11] Hoare, Tony, Bernhard Möller, Georg Struth y Ian Wehrman: *Concurrent Kleene Algebra and its Foundations.* The Journal of Logic and Algebraic Programming, Aug 2011.

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

[Koz94] Kozen, D.: *A Completeness Theorem for Kleene Algebras and the Algebra of Regular Events.* Information and Computation, 110(2):366–390, 1994.

[KS13] Katsumata, Shin-ya y Tetsuya Sato: *Preorders on Monads and Coalgebraic Simulations.* FOSSACS'13, página 145–160, Berlin, Heidelberg, 2013. Springer-Verlag, ISBN 9783642370748.

[KS17] Kozen, Dexter y Alexandra Silva: *Practical coinduction.* Mathematical Structures in Computer Science, 27(7):1132–1152, 2017.

[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.

[RJ19] Rivas, Exequiel y Mauro Jaskelioff: *Monads with merging.* https://inria.hal.science/hal-02150199, working paper or preprint, Junio 2019.