

*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**  
LICENCIATURA EN CIENCIAS DE LA COMPUTACIÓN

**AUTORA:**

*Bini, Valentina María*

**DIRECTOR:**

*Rivas, Exequiel*



Universidad  
Nacional  
de Rosario

Facultad de  
Ciencias Exactas,  
Ingeniería y Agrimensura





# Índice general

<b>I</b>	<b>Preliminares</b>	<b>3</b>
<b>1.</b>	<b>Introducción a Agda</b>	<b>5</b>
1.1.	Tipos de datos y <i>Pattern Matching</i> . . . . .	5
1.2.	Tipos dependientes . . . . .	9
1.3.	Familias de Tipos de datos . . . . .	10
1.4.	Sistema de Módulos . . . . .	11
1.5.	Records . . . . .	13
1.6.	Características adicionales . . . . .	15
<b>2.</b>	<b>Mónadas Concurrentes</b>	<b>17</b>
2.1.	Teoría de Categorías . . . . .	17
2.2.	Introducción a las mónadas . . . . .	19
2.3.	Mónadas Concurrentes . . . . .	22
<b>3.</b>	<b>La Mónada <i>Delay</i></b>	<b>27</b>
3.1.	Introducción a la Coinducción . . . . .	27
3.2.	Números Conaturales . . . . .	30
3.3.	Mónada <i>Delay</i> . . . . .	31
3.4.	Coinducción en Agda . . . . .	33
<b>II</b>	<b>Formalización de Mónadas Concurrentes: el Caso de la Mónada <i>Delay</i></b>	<b>35</b>
<b>4.</b>	<b>Formalización de Mónadas Concurrentes en Agda</b>	<b>37</b>
4.1.	Formalización en Agda: los monoides . . . . .	37
4.2.	Formalización a nivel de Funtores y Mónadas . . . . .	38
4.3.	Formalización de Monoides Concurrentes . . . . .	42
4.4.	Formalización de Mónadas Concurrentes . . . . .	44
<b>5.</b>	<b>El caso de la Mónada <i>Delay</i></b>	<b>49</b>
5.1.	Definición del tipo <i>delay</i> con notación musical . . . . .	49
5.2.	Prueba de que el tipo <i>delay</i> es una mónada . . . . .	55
5.3.	Prueba de que el tipo <i>delay</i> es un functor monoidal . . . . .	58

5.4. ¿El tipo <i>delay</i> es una mónada concurrente? . . . . .	61
5.5. Reduciendo el problema a los conaturales . . . . .	71
5.6. Cambio de paradigma: <i>sized types</i> . . . . .	71
<b>Bibliografía</b>	<b>76</b>

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 *depend* 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 bloque de 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 sobre 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 definen 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 del operador, 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 en uno de los casos del *pattern matching* al definir una función significa que no es necesario dar una definición para ese caso ya que no es posible dar un argumento para la función que caiga dentro de ese caso. 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 prueba 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. Tipos 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  $(x : A) \rightarrow B$  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
```

Existen otros tipos dependientes además de las funciones. Uno muy utilizado son los pares dependientes, los cuales consisten en un par ordenado o tupla de dos elementos en la cual el tipo del segundo elemento depende del valor del primero. Estos pares son llamados usualmente  $\Sigma$ -types (*sigma types*) y pueden definirse de dos maneras. La primera, utilizando una declaración `data`, se muestra a continuación. La otra se define utilizando un tipo `record` y se verá en la sección 1.5.1. Se reserva el nombre  $\Sigma$  para esta segunda definición ya que es la más estándar y la más utilizada, su código fuente se encuentra en el módulo `Agda.Builtin.Sigma`. Para la versión que se presentará a continuación se utilizará el nombre alternativo  $\Sigma'$ .

```
infixr 4 _'_ _
data Σ' (A : Set) (B : A → Set) : Set where
  _'_ : (a : A) → (b : B a) → Σ' A B
```

Como se puede observar, el tipo del segundo elemento es  $B$  y depende de un valor de tipo  $A$  que se le pase como argumento. Para construir un par de este tipo se utiliza el constructor `_'_`. Este toma un valor  $a$  de tipo  $A$  y luego un valor  $b$  de tipo  $B a$ , por lo cual queda evidenciado que el tipo del valor  $b$  depende del valor de  $a$ .

### 1.2.1. Argumentos Implícitos

Los tipos dependientes sirven, entre otras cosas, para definir funciones polimórficas. 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, se 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  $\mathbb{N} \rightarrow \text{Set}$ . 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 construido por el constructor `_::_`, y resulta útil ya que la función `head` no está correctamente definida para el caso de la lista vacía.

En algunos casos puede ser necesario dar un único tipo que englobe a todas las listas de todos los largos posibles. Esto puede expresarse mediante la unión disjunta de los vectores de cada posible largo. Expresar la unión disjunta de familias de tipos de datos es un uso muy común de los  $\Sigma$ -types. Escribir  $\Sigma' \mathbb{N} (\text{Vec } A)$  (o lo que es equivalente,  $\Sigma' \mathbb{N} (\lambda n \rightarrow \text{Vec } A \ n)$ ) es conceptualmente lo mismo que escribir `List A`.

Un ejemplo en el cual es necesaria la unión disjunta de todos los vectores es para definir la función `filterVec` que, dado un vector de cierto largo  $n$  y un predicado, filtra todos los elementos del vector que cumplen la condición dada. En este caso no es posible saber de antemano qué largo tendrá el vector resultante, puesto que se desconoce cuántos elementos del vector original cumplirán la condición. Es por esto que se define como sigue:

```
filterVec : {A : Set} {n : ℕ} → (A → Bool) → Vec A n → Σ' ℕ (Vec A)
filterVec _ [] = zero , []
filterVec f (x :: xs) with filterVec f xs | f x
... | length , filtered | true = (suc length) , (x :: filtered)
... | length , filtered | false = length , filtered
```

## 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 `suc2` 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 en la dirección `/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 habitante 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 tipos `record` pueden ser, al igual que los tipos `data`, 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.

### 1.5.1. Campos con tipos dependientes

A la hora de definir un `record`, el tipo de un campo puede depender de los valores de todos los campos anteriores. Esto hace que el orden en que se introducen los campos no siempre pueda ser arbitrario. A continuación se muestra la definición de los  $\Sigma$ -types como un tipo `record`, en la cual el tipo del segundo campo depende del valor del primero.

```
record  $\Sigma$  (A : Set) (B : A → Set) : Set where
  constructor _,_
  field
    fst : A
    snd : B fst

open  $\Sigma$  public
```

## 1.6. Características adicionales

Antes de finalizar este capítulo, se describirán algunas características adicionales específicas de Agda que son importantes para comprender la potencia del lenguaje.

### 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 `Set1`:

```
Set : Set1
```

Las expresiones de tipo `Set1` 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 `Set1` son potencialmente *más grandes*. Cuando se tiene  $A : \text{Set}_1$ , entonces se dice a veces que  $A$  es un *conjunto grande*. Sucesivamente, se tiene que:

```
Set1 : Set2
```

```
Set2 : Set3
```

etcétera. Un tipo cuyos habitantes son tipos se llama **universo**. Agda provee un número infinito de universos `Set`, `Set1`, `Set2`, `Set3`, ..., cada uno de los cuales es un habitante del siguiente. `Set` es en sí mismo una abreviación de `Set0`. El subíndice  $n$  es el **nivel** del universo `Setn`. Agda provee también un tipo primitivo especial `Level`, cuyos habitantes son los posibles niveles de los universos. De hecho, la notación `Setn` es una abreviación para `Set n`, donde  $n : \text{Level}$ .

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 3.4, 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 presentada 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**  $\mathcal{C}$  consiste de:

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

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

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

- Y para cada  $A, B, C \in \mathbf{ob} \mathcal{C}$  una operación

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

llamada **composición** con las siguientes propiedades:

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

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 **1**). La categoría **1** es aquella tal que:

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

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  $0 \in \text{ob } \mathcal{C}$  se dice **inicial** si  $\forall A \in \text{ob } \mathcal{C}, \exists ! 0 \rightarrow A$ . Un objeto  $1 \in \text{ob } \mathcal{C}$  se dice **terminal** si  $\forall A \in \text{ob } \mathcal{C}, \exists ! A \rightarrow 1$ .

**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  $\mathcal{C}$  es una terna  $(A \times B, \pi_A, \pi_B)$  donde:

- $\pi_A \in \text{Hom}(A \times B, A)$ ,
- $\pi_B \in \text{Hom}(A \times B, B)$
- y para todo objeto  $C$  y para todo par de morfismos  $f : C \rightarrow A, g : C \rightarrow B$ , existe un único morfismo  $\langle f, g \rangle : C \rightarrow A \times B$  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  $f : A \rightarrow C$  y  $g : B \rightarrow D$ , se puede definir un morfismo  $f \times g : A \times B \rightarrow C \times D$  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  $\mathcal{C}$  es una terna  $(A + B, \iota_A, \iota_B)$  donde:

- $\iota_A \in \text{Hom}(A, A + B)$ ,
- $\iota_B \in \text{Hom}(B, A + B)$
- y para todo objeto  $C$  y para todo par de morfismos  $f : A \rightarrow C, g : B \rightarrow C$  existe un único morfismo  $[f, g] : A + B \rightarrow C$  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  $\mathcal{C}$  una categoría con productos binarios y sean  $A, B \in \mathbf{ob} \mathcal{C}$ . Un objeto  $B^A$  es un **exponencial** si existe un morfismo  $\varepsilon : B^A \times A \rightarrow B$  tal que para todo morfismo  $g : C \times A \rightarrow B$  existe un único morfismo  $\tilde{g} : C \rightarrow B^A$  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  $\mathcal{C}$  y  $\mathcal{D}$  dos categorías. Un **funtor**  $F : \mathcal{C} \rightarrow \mathcal{D}$  asigna:

- a cada objeto  $A \in \mathbf{ob} \mathcal{C}$ , un objeto  $F(A) \in \mathbf{ob} \mathcal{D}$ ;
- a cada morfismo  $f : A \rightarrow B \in \mathbf{mor} \mathcal{C}$ , un morfismo  $F(f) : F(A) \rightarrow F(B) \in \mathbf{mor} \mathcal{D}$  tal que:
  - para todo  $A \in \mathbf{ob} \mathcal{C}$ ,  $F(id_A) = id_{F(A)}$ ;
  - para todos  $f, g \in \mathbf{mor} \mathcal{C}$  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,  $F : \mathcal{C} \rightarrow \mathcal{C}$ .

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  $F, G : \mathcal{C} \rightarrow \mathcal{D}$  dos funtores (entre las mismas categorías). Una **transformación natural**  $\eta : F \rightarrow G$  asigna a cada  $A \in \mathbf{ob} \mathcal{C}$  un morfismo  $\eta_A : F(A) \rightarrow G(A)$  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  $M \times M \rightarrow M$ ,  $(m, n) \rightarrow mn$  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 [Man76], 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  $\mathcal{C}$ , una **mónada** sobre  $\mathcal{C}$  es una tupla  $(T, \mu, \eta)$ , donde:

- ▶  $T : \mathcal{C} \rightarrow \mathcal{C}$  es un funtor,
- ▶  $\eta : Id \rightarrow T$  y  $\mu : T \cdot T \rightarrow T$  son transformaciones naturales
- ▶ y se cumplen las siguientes identidades:

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

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

**Ejemplo 2.13** (Mónada *Error*). Sea  $T : \mathbf{Set} \rightarrow \mathbf{Set}$  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  $\eta$  y  $\mu$  como siguen:

- ▶ Para cada conjunto  $X$ , se define  $\eta_X : IdX \rightarrow TX$  como  $\eta_X(x) = inl(x)$ .
- ▶ Para cada conjunto  $X$ , se define  $\mu_X : TTX \rightarrow TX$  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  $T : \mathbf{Set} \rightarrow \mathbf{Set}$  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  $\eta$  y  $\mu$  como sigue:

- ▶ Para cada conjunto  $X$ , se define  $\eta_X : IdX \rightarrow TX$  como  $\eta_X(x) = (\lambda s : S. \langle x, s \rangle)$ .
- ▶ Para cada conjunto  $X$ , se define  $\mu_X : TTX \rightarrow TX$  como  $\mu_X(f) = (\lambda s : S. \text{let } \langle f', s' \rangle = f(s) \text{ in } f'(s'))$ , 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  $\mathcal{C}$  es una tupla  $(T, \eta, \_*)$ , donde

- ▶  $T : \mathbf{ob} \mathcal{C} \rightarrow \mathbf{ob} \mathcal{C}$ ,
- ▶ para cada  $A \in \mathbf{ob} \mathcal{C}$ ,  $\eta_A : A \rightarrow TA$ ,



- para cada  $f : A \rightarrow TB$ ,  $f^* : TA \rightarrow TB$ ,
- y se cumplen las siguientes ecuaciones:
  - $\eta_A^* = id_{TA}$
  - $f^* \circ \eta_A = f$  para cada  $f : A \rightarrow TB$
  - $g^* \circ f^* = (g^* \circ f)^*$  para cada  $f : A \rightarrow TB$  y  $g : B \rightarrow TC$ .

Intuitivamente  $\eta_A$  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 descrito en la versión clásica:

- Para cada conjunto  $X$ , se define  $\eta_X : IdX \rightarrow TX$  como  $\eta_X(x) = inl(x)$  al igual que en la versión clásica.
- Para cada función  $f : X \rightarrow TY$ , se define  $f^* : TX \rightarrow TY$  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 descrito en la versión clásica:

- Para cada conjunto  $X$ , se define  $\eta_X : IdX \rightarrow TX$  como  $\eta_X(x) = (\lambda s : S. \langle x, s \rangle)$  al igual que en la primera versión.
- Para cada función  $f : X \rightarrow TY$ , se define  $f^* : TX \rightarrow TY$  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  $F : \mathcal{C} \rightarrow \mathcal{C}$  es **fuerte** si viene equipado con una transformación natural  $\sigma_{X,Y} : FX \times Y \rightarrow F(X \times Y)$ , de manera que se cumplen las siguientes ecuaciones:

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

donde  $\pi_1$  y  $\pi_2$  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**  $(T, \mu, \eta)$  sobre  $\mathcal{C}$  es **fuerte** si el funtor subyacente  $T$  es fuerte y la fortaleza es compatible con  $\mu$  y  $\eta$ :

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

Hay una definición similar de fortaleza  $\bar{\sigma}_{X,Y} : X \times FY \rightarrow F(X \times Y)$  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  $F : \mathcal{C} \rightarrow \mathcal{C}$  equipado con una estructura monoidal  $(m, e)$ , donde  $m : FX \times FY \rightarrow F(X \times Y)$  es una transformación natural y  $e : \mathbf{1} \rightarrow F\mathbf{1}$  es un morfismo tal que las siguientes ecuaciones se cumplen:

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

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

Además, si la estructura monoidal es compatible con  $\gamma$ , entonces el funtor monoidal es simétrico.

**Definición 2.21** (Mónada monoidal). Una **mónada monoidal** es una mónada  $(T, \mu, \eta)$  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  $(T, \mu, \eta)$ ,  $T$  como funtor puede ser equipado con dos estructuras monoidales canónicas:

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

y  $e = \eta_{\mathbf{1}} : \mathbf{1} \rightarrow T\mathbf{1}$  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  $(T, \mu, \eta)$  sobre esta categoría son fuertes, y cada uno admite una única fortaleza posible  $\sigma$  ( $\bar{\sigma}$ ).

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<sup>+</sup>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.  $\square$

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<sup>+</sup>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  $\epsilon$  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:

$$p * q = \{\sigma_0 \bullet \sigma_1 \mid (\sigma_0, \sigma_1) \in \text{dom}(\bullet) \wedge \sigma_0 \in p \wedge \sigma_1 \in q\}$$

$$\text{emp} = \{u\}$$

El conjunto de funciones monótonas  $\mathcal{P}(\Sigma) \rightarrow \mathcal{P}(\Sigma)$  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  $\Sigma$ :

$$(F * G)Y = \bigcup \{FY_1 * GY_2 \mid Y_1 * Y_2 \subseteq Y\}$$

$$\text{nothing } Y = Y \cap \text{emp}$$

$$(F; G)Y = F(G(Y))$$

$$\text{skip } Y = Y$$

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 realiza 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 algebraicas, se necesita una noción de compatibilidad de las operaciones con el orden. Sea  $(A, \sqsubseteq)$  un orden parcial, entonces una operación  $\oplus : A \times A \rightarrow A$  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  $(A, ;, \text{skip})$  y  $(A, *, \text{nothing})$  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  $\text{nothing} = \text{skip}$ , 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 bimonoides ordenados, es un monoide concurrente.

### 2.3.4. Generalización a nivel de funtores y mónadas

Para elevar la teoría descrita 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  $m_{A,B} : TA \times TB \rightarrow T(A \times B)$ . Como se necesita que  $*$  sea un monoide conmutativo, se requiere que  $T$  tenga una estructura de funtor monoidal simétrica, incluyendo un morfismo unidad  $e : \mathbf{1} \rightarrow T\mathbf{1}$  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  $f : I \rightarrow J$ , el morfismo  $Ff : FI \rightarrow FJ$  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**  $(T, \mu, \eta)$  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  $\mathcal{Kl}(T)(A, B)$  definido por  $f \sqsubseteq_{A,B} g$  si y sólo si  $\forall a : \mathbf{1} \rightarrow A, f \circ a \sqsubseteq_B g \circ a$ .

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  $v, v' : \mathbf{1} \rightarrow TA$  y  $w, w' : \mathbf{1} \rightarrow TB$ .

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  $(T, \eta, \mu)$  es una mónada monoidal ordenada si está dotada de una estructura de funtor monoidal simétrico

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

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

*Aclaración: se escribe  $f \star g$  para representar  $m \circ (f \times g)$  y  $f \bullet g$  para denotar  $f^* \circ g$ .*

**Definición 2.29** (Mónada concurrente). Una mónada monoidal ordenada  $T$  es una mónada concurrente si  $e = \eta_1 : \mathbf{1} \rightarrow T\mathbf{1}$  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  $(T, \mu, \eta)$  una mónada conmutativa. Entonces  $T$  tiene una única estructura de mónada monoidal  $(m, \eta_1)$ , 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  $(T, \mu, \eta)$  una mónada monoidal ordenada (mónada concurrente). Entonces  $T\mathbf{1}$  es un bimonioide ordenado (monioide concurrente).*

Como en el caso de los monoides, hay dos estructuras de bimonioide 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 bimonioide ordenado, este puede ser elevado a una mónada monoidal ordenada.

**Lema 2.32.** *Sea  $(A, \sqsubseteq, *, \text{nothing}, ;, \text{skip})$  un bimonioide ordenado. Este puede convertirse en una mónada monoidal ordenada con funtor soporte  $T_A X = A \times X$ , operaciones y orden. Más aún, si  $\text{nothing} = \text{skip}$  (es decir que es un monioide concurrente), entonces  $e = \eta_1$  (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  $Tr_L(X) = \mathcal{P}_f(\text{Trazas}_{L \times X})$  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 y un conjunto coinductivo muy especial: los números conaturales. Luego se presentará la definición de la mónada *delay* y sus principales características. Finalmente, se describirán algunos de los distintos soportes para coinducción que Agda proporciona.

### 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 principio 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. Es por esta razón que esta sección comenzará primero con un repaso de lo conocido: la inducción, para luego introducir la coinducción haciendo un paralelismo entre ambas. A continuación se desarrolla la noción de definición inductiva.

**Definición 3.1** (Definición inductiva). Sean:

- $U$  un conjunto que se denominará **universo**.
- $B$  un subconjunto no vacío de  $U$  que tendrá el nombre de **base**.
- $K$  un conjunto no vacío de funciones llamado **constructor**. Cada función  $f \in K$  tiene cierta aridad  $ar(f) = n \in \mathbb{N}$ , de manera que  $f : U^n \rightarrow U$ .

Un conjunto  $A$  está **definido inductivamente** por  $B$ ,  $K$  y  $U$  si es el mínimo conjunto que satisface que:

- $B \subseteq A$

- si  $a_1, \dots, a_n \in A$  y  $f(a_1, \dots, a_n) = a$ , entonces  $a \in A$ , donde  $f \in K$ .

Cuando esto sucede se dice que  $A$  fue generado por la base  $B$  y las reglas de inducción  $f \in K$ .

Que  $A$  sea el mínimo conjunto que cumple estas condiciones implica que si se tiene otro conjunto  $A'$  que satisface las mismas condiciones, entonces  $A \subseteq A'$ . De la mano con las definiciones inductivas viene el principio de inducción estructural, el cual sirve para demostrar propiedades sobre conjuntos definidos inductivamente.

**Teorema 3.2** (Principio de inducción estructural). *Sea  $A \subseteq U$  definido inductivamente por la base  $B$  y el constructor  $K$ . Sea  $P$  una propiedad que verifica:*

- $\forall x \in B, P(x)$  es verdadera.
- Para cada  $f \in K$ , si  $f(a_1, \dots, a_{ar(f)}) = a$  y  $P(a_i)$  se cumple para cada  $i = 1, \dots, ar(f)$ , entonces  $P(a)$  se cumple.

*Entonces se cumple  $P(x) \forall x \in A$ .*

Como se mencionó previamente, el principio de coinducción es el principio dual de la inducción. Asimismo, las definiciones coinductivas son duales a las definiciones inductivas. Un conjunto se define coinductivamente de la siguiente manera:

**Definición 3.3** (Definición coinductiva). Sean:

- $U$  un conjunto que se denominará **universo**.
- $B$  un subconjunto de  $U$  que tendrá el nombre de **base**.
- $K$  un conjunto no vacío de funciones llamado **constructor**. Cada función  $f \in K$  tiene cierta aridad  $ar(f) = n \in \mathbb{N}$ , de manera que  $f : U^n \rightarrow U$ .

Un conjunto  $A$  está **definido coinductivamente** por  $B, K$  y  $U$  si es el máximo conjunto que satisface que:

- $B \subseteq A$
- si  $a_1, \dots, a_n \in A$  y  $f(a_1, \dots, a_n) = a$ , entonces  $a \in A$ , donde  $f \in K$ .

Se puede observar que ambas definiciones son muy similares, la diferencia sustancial está en las palabras *mínimo* y *máximo*. Para ver más clara la diferencia, en el caso de una definición inductiva, se puede pensar que el conjunto a definir comienza siendo el conjunto vacío  $\emptyset$  y se van añadiendo elementos iterativamente de acuerdo con las reglas. En una definición coinductiva, por el contrario, se puede imaginar que el conjunto comienza siendo el conjunto de todos los objetos posibles (el universo  $U$ ), e iterativamente se van removiendo los objetos que contradicen las reglas. El hecho de que en este caso  $A$  sea el máximo conjunto que cumple las condiciones, implica que si hay otro conjunto  $A'$  que también las satisface, entonces  $A' \subseteq A$ .

Una vez que se tienen conjuntos definidos coinductivamente, se pueden probar propiedades sobre ellos mediante el principio de coinducción, el cual fue introducido por primera vez por Milner y Tofte en 1991 [MT91].

**Teorema 3.4** (Principio de coinducción). *Sea  $A \subseteq U$  definido coinductivamente por la base  $B$  y el constructor  $K$ . Sea  $P$  una propiedad que verifica:*

- Para cada  $f \in K$ , si  $f(a_1, \dots, a_{ar(f)}) = a$  y  $P(a_i)$  se cumple para cada  $i = 1, \dots, ar(f)$ ,



entonces  $P(a)$  se cumple.

Entonces se cumple  $P(x) \forall x \in A$ .

Al igual que antes, el principio es bastante similar al inductivo, excepto que no se pide que la propiedad se cumpla para los casos base. De hecho, un conjunto definido coinductivamente podría no tener un conjunto base ( $B = \emptyset$ ), es el caso, por ejemplo, de los flujos de datos (*streams*) infinitos. Aún teniendo un conjunto base, no quiere decir que todos los elementos del conjunto se construyan a partir de elementos de la base. Esta característica hace parecer que el paso coinductivo no está bien fundado y muchas veces genera dudas sobre la veracidad de las pruebas por coinducción. Sin embargo, cualquier dificultad que haga que la propiedad a demostrar no se cumpla se manifiesta en el intento de probar el paso coinductivo.

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:

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

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

$$\text{Lista de } A = \text{nil} + A \times \text{Lista de } A \quad (3.1)$$

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. En este ejemplo se evidencia que, a pesar de que el conjunto tiene un caso base, los elementos no necesariamente se construyen a partir de él, como es el caso de las listas de longitud infinita.

Si bien Milner y Tofte [MT91] fueron los primeros en introducir la coinducción, ellos lo hicieron en base a puntos fijos mínimos y máximos. Existe otro trasfondo teórico sobre el cual se puede definir formalmente la coinducción, el cual se basa en álgebras iniciales y coálgebras finales. Este enfoque es el que será utilizado en este trabajo y fue con el que trabajaron Jacobs y Rutten [JR12]. Se definen a continuación los conceptos de álgebra, coálgebra, álgebra inicial y coálgebra final:

**Definición 3.5** (Álgebra de un funtor). Dado un endofunctor  $F$  sobre una categoría  $\mathcal{C}$ , un **álgebra** de  $F$  es un objeto  $X$  de  $\mathcal{C}$  junto con un morfismo  $\alpha : FX \rightarrow X$ . Dadas dos álgebras  $(X, \alpha : FX \rightarrow X)$ ,  $(Y, \beta : FY \rightarrow Y)$  de  $F$ ,  $m : X \rightarrow Y$  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 3.6** (Coálgebra de un funtor). Una **coálgebra** para un endofunctor  $F$  sobre una categoría  $\mathcal{C}$  es un objeto  $A$  junto con un morfismo  $u : A \rightarrow FA$ . Dadas dos coálgebras  $(A, \eta : A \rightarrow FA)$ ,  $(B, \theta : B \rightarrow FB)$ ,  $f : A \rightarrow B$  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 3.7** (Álgebra inicial). Un **álgebra inicial** para un endofunctor  $F$  sobre una categoría  $\mathcal{C}$  es un objeto inicial en la categoría de las  $F$ -álgebras.

**Definición 3.8** (Coálgebra final). Una **coálgebra final** para un endofunctor  $F$  sobre una categoría  $\mathcal{C}$  es un objeto terminal en la categoría de las  $F$ -coálgebras.

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`, `::`). Los tipos coinductivos se definen como elementos de una coálgebra final para un endofunctor dado en la categoría **Set**.

Se muestran a continuación algunos ejemplos más presentados por Kozen y Silva en [KS17].

**Ejemplo 3.9** (Flujo de datos infinitos). El conjunto  $A^\omega$  de flujos de datos infinitos sobre un alfabeto  $A$  es (el conjunto soporte de) la coálgebra final del funtor  $FX = A \times X$ .

**Ejemplo 3.10** (Cadenas infinitas). El conjunto  $A^\infty$  de las cadenas finitas e infinitas sobre un alfabeto  $A$  es (el conjunto soporte de) la coálgebra final del funtor  $FX = 1 + A \times X$ .

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  $hd : A^\omega \rightarrow A$  y  $tl : A^\omega \rightarrow A^\omega$ , 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  $A^\omega$  es una coálgebra para el funtor  $FX = A \times X$ . Todas estas coálgebras vienen equipadas con una función estructural  $\langle obs, cont \rangle : X \rightarrow A \times X$ ; para  $A^\omega$  se tiene que  $obs = hd$  y  $cont = tl$ .

## 3.2. Números Conaturales

Un conjunto coinductivo muy importante es el de los números conaturales, también llamados conúmeros. Como su nombre lo indica, es el conjunto dual al de los números naturales. Para introducir este conjunto, al igual que en la sección anterior, se partirá de lo conocido, siguiendo el estilo utilizado por Mike Gordon [Gor17].

La inducción matemática es la inducción estructural aplicada a la estructura de los números naturales  $(\mathbb{N}, 0, S)$ , donde  $\mathbb{N}$  es un conjunto,  $0 \in \mathbb{N}$  es una constante u operador de aridad 0 (conjunto base  $B = \{0\}$ ) y  $S : \mathbb{N} \rightarrow \mathbb{N}$  es una función de aridad 1 ( $K = \{S\}$ ). La estructura de los números naturales está caracterizada por los cinco axiomas de Peano:

- $0 \in \mathbb{N}$
- $\forall n \in \mathbb{N}, S(n) \in \mathbb{N}$
- $\forall n \in \mathbb{N}, S(n) \neq 0$
- $\forall m \in \mathbb{N}, \forall n \in \mathbb{N}, S(m) = S(n) \Rightarrow m = n$
- $\forall M, 0 \in M \wedge (\forall n \in M, S(n) \in M) \Rightarrow \mathbb{N} \subseteq M$

La estructura  $(\mathbb{N}, 0, S)$  es una instancia de la clase de estructuras  $(\mathcal{A}, z, s)$ , donde  $\mathcal{A}$  es un conjunto,  $z \in \mathcal{A}$  y  $s : \mathcal{A} \rightarrow \mathcal{A}$ . Estas estructuras se denominan estructuras de Peano. Para cada una de ellas existe una única función  $f : \mathbb{N} \rightarrow \mathcal{A}$  tal que  $f(0) = z$  y  $\forall n \in \mathbb{N}, f(S(n)) = s(f(n))$ . Una estructura de Peano es un álgebra (pueden llamarse también álgebras de Peano) y  $(\mathbb{N}, 0, S)$  es un álgebra inicial en la categoría de las álgebras de Peano. De manera dual, la estructura de los conaturales es una cóalgebra final en la categoría de las cóalgebras de Peano.

El conjunto de los números conaturales  $\bar{\mathbb{N}}$  puede definirse como el máximo conjunto que cumple que  $0 \in \bar{\mathbb{N}}$  y si  $n \in \bar{\mathbb{N}}$  entonces  $S(n) \in \bar{\mathbb{N}}$  (las mismas condiciones que los números naturales). Sin embargo, como se mencionó anteriormente, los conjuntos coinductivos suelen definirse por sus destructores en lugar de sus constructores. Esto hace que la definición sea más intuitiva y se comprenda mejor.

El dual del constructor unario  $S$  es la función predecesor  $P$  definida como  $P(n) = n - 1$ , donde  $P$  es una función parcial definida sólo para los números distintos de 0, por lo cual  $\text{Dom}(P) = \{n \mid n > 0\}$ . Los constructores de aridad 0 representan elementos distinguidos del conjunto soporte, por lo que no es evidente cuáles son sus destructores correspondientes ya que no hay una componente del constructor que retornar. Para enfrentar esto, los destructores correspondientes a constructores 0-arios retornan un valor arbitrario que representa “sin componentes”. Este puede ser denotado como  $\star$  o  $()$ . El destructor dual al constructor 0-ario 0 es la función parcial  $is0 : \{0\} \rightarrow \{\star\}$ , por lo que necesariamente  $is0(0) = \star$  y  $\text{Dom}(is0) = \{0\}$ .

El dual de un álgebra de Peano  $(\mathcal{A}, z, s)$  es una cóalgebra de Peano  $(\mathcal{C}, isz, p)$ , donde  $isz$  y  $p$  son destructores:  $isz : \mathcal{C} \rightarrow \{\star\}$  y  $p : \mathcal{C} \rightarrow \mathcal{C}$ . Si una cóalgebra tiene más de un destructor, entonces todos sus destructores son funciones parciales. Los dominios de  $isz$  y  $p$  forman una partición de  $\mathcal{C}$ , por lo que si  $x \in \mathcal{C}$  entonces o bien  $x \in \text{Dom}(isz)$  o bien  $x \in \text{Dom}(p)$ , pero no ambos. Los números conaturales son la cóalgebra de Peano  $(\bar{\mathbb{N}}, is0, P)$ , con la propiedad de que para cualquier cóalgebra de Peano  $(\mathcal{C}, isz, p)$  hay una única función  $g : \mathcal{C} \rightarrow \bar{\mathbb{N}}$  tal que para cada  $x \in \mathcal{C}$ :

- si  $x \in \text{Dom}(isz)$  entonces  $g(x) \in \text{Dom}(is0)$  e  $is0(g(x)) = isz(x)$ ;
- si  $x \in \text{Dom}(p)$  entonces  $g(x) \in \text{Dom}(P)$  y  $P(g(x)) = g(p(x))$ .

La cóalgebra  $(\bar{\mathbb{N}}, is0, P)$  es entonces una cóalgebra terminal o final en la categoría de las cóalgebras de Peano.

Intuitivamente, un número conatural puede ser o bien el sucesor de otro (está en el dominio de la función  $P$  y se puede obtener su predecesor) o bien 0 (está en el dominio de  $is0$ ), de manera que se evidencia el paralelismo entre ambas definiciones. La diferencia con los naturales es que, en este caso, un conatural puede ser sucesor de otro “infinitamente”, puesto que no hay condición alguna que indique que al aplicar la función predecesor sucesivamente debe llegarse obligatoriamente a 0. Esto da lugar a un valor especial  $\infty$  tal que  $P(\infty) = \infty$ . El conjunto soporte  $\bar{\mathbb{N}}$  sería entonces de alguna manera equivalente a  $\mathbb{N} \cup \{\infty\}$ .

### 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 por Chapman et al [CUV19].

**Definición 3.11** (*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:

$$\frac{}{\text{now } x : \mathbf{D}X} \quad \frac{c : \mathbf{D}X}{\text{later } c : \mathbf{D}X}$$

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 3.12** ( $R$ -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:

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

Siendo  $\equiv$  la igualdad proposicional, 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 3.13** (Convergencia). La relación de **convergencia** denotada con  $\downarrow$  entre  $\mathbf{D}X$  y  $X$  se define inductivamente mediante las siguientes reglas:

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

**Definición 3.14** ( $R$ -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:

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

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  $\eta$  es el constructor **now**, mientras que la multiplicación  $\mu$  es la “concatenación” de constructores **later**:

$$\begin{aligned} \mu : \mathbf{D}(\mathbf{D}X) &\rightarrow \mathbf{D}X \\ \mu (\text{now } c) &= c \\ \mu (\text{later } c) &= \text{later } (\mu c) \end{aligned}$$

### 3.4. 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.

#### 3.4.1. Notación Musical

Para mostrar la notación musical se utilizará como ejemplo el conjunto de los números *conaturales*, los cuales se definen en Agda como sigue:

```
data CoN : Set where
  zero : CoN
  suc :  $\infty$  CoN  $\rightarrow$  CoN
```

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

```
#_ :  $\forall \{a\} \{A : \text{Set } a\} \rightarrow A \rightarrow \infty A$ 
b :  $\forall \{a\} \{A : \text{Set } a\} \rightarrow \infty A \rightarrow A$ 
```

La función *delay* ( $\#_$ ) toma un valor de tipo A y lo devuelve suspendido dentro de un valor de tipo  $\infty$  A. Por el contrario, la función *force* (*b*), toma un valor de tipo  $\infty$  A y lo desencapsula devolviendo un valor de tipo A.

Los valores de tipos coinductivos pueden ser contruidos usando corecursión, la cual no debe necesariamente terminar, pero sí ser productiva. Por ejemplo, el infinito puede ser definido 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 constructor 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.

#### 3.4.2. 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  $j : \text{Size}< i$ , este es forzado a ser también de tipo `Size`. La relación de orden de los tamaños es transitiva, lo que implica que si se tiene que  $j : \text{Size}< i$  y  $k : \text{Size}< j$ , entonces  $k : \text{Size}< i$ . La relación de orden es, además, bien fundada, lo cual se usa para definir funciones corecursivas productivas. Existe también una operación sucesor de tamaños  $\uparrow$  y un tamaño “infinito”  $\infty$  tal que para cada tamaño  $i$ ,  $i : \text{Size}< \infty$ . 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  $n' : \text{Conat}' i$ , el valor resultante es un  $n : \text{Conat} j$  de una profundidad estrictamente menor  $j < i$ . Un caso especial es el valor  $\infty n' : \text{Conat}' \infty$  de índice infinito, cuyo resultado de aplicar `force` es  $\infty n : \text{Conat} \infty$ , el cual también tiene índice infinito. De esta manera los tamaños establecen productividad en las definiciones recursivas. Al final, sólo interesan los valores  $n : \text{Conat} \infty$  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})
```

## Parte II

# Formalización de Mónadas Concurrentes: el Caso de la Mónada Delay





## Capítulo 4

# Formalización de Mónadas Concurrentes en Agda

Formalizar la matemática consiste en representar las estructuras y pruebas matemáticas en un sistema axiomático formal de manera que la correctitud de dichas pruebas pueda ser verificada mecánicamente. Esto quiere decir que el proceso de verificación es algorítmico y puede ser realizado por una computadora sin recurrir a la creatividad o la intuición. El proceso de formalización es complejo, es por esto que existen herramientas especializadas tales como asistentes de pruebas para llevarlo a cabo de manera más simple y práctica.

Como se mencionó anteriormente, Agda es, además de un lenguaje de programación funcional con tipos dependientes, un asistente de pruebas. Por el isomorfismo de Curry-Howard, se pueden representar proposiciones lógicas mediante tipos. Una proposición se demuestra escribiendo un programa del tipo correspondiente. Cuando se formaliza una estructura algebraica en Agda, lo que se hace es definir un tipo que la represente. De esta manera, al generar un habitante de dicho tipo, se genera una instancia de la estructura algebraica representada.

En este capítulo se mostrará el modo de formalizar estructuras algebraicas en Agda comenzando con un ejemplo simple: los monoides. Luego se agregará complejidad mostrando formalizaciones a nivel de funtores y mónadas, al igual que se fue escalando en los capítulos anteriores. Finalmente, se dará la formalización de los monoides y mónadas concurrentes.

### 4.1. Formalización en Agda: los monoides

Como se introdujo en la definición 2.11, un monoide consiste en un conjunto junto con una operación binaria asociativa tal que en el conjunto exista un elemento que actúe como neutro respecto de la operación. En Agda, se formaliza este concepto definiendo un tipo `record` que lo representa. No hay una única forma de definir este tipo, por lo que al hacerlo se tomaron varias decisiones. A continuación se muestra cómo quedó la última versión de esta definición y se explican las principales decisiones tomadas junto con el significado y propósito de cada uno de los campos.

```
record Monoid (M : Set) : Set₁ where
  constructor
  makeMonoid
  field
    _≅ₘ_ : M → M → Set
    eqₘ   : IsEquivalence _≅ₘ_
    zeroₘ : M
    _+ₘ_  : M → M → M
    idl   : (x : M) → (zeroₘ +ₘ x) ≅ₘ x
```

```

idr      : (x : M) → (x +m zerom) ≅m x
assoc    : (x : M) (y : M) (z : M) → (x +m (y +m z)) ≅m ((x +m y) +m z)

```

open Monoid public

Como se puede observar, el `record Monoid` tiene un parámetro  $M$  de tipo `Set`. Este representa el conjunto soporte del monoide. Se podría discutir si este debería ser un parámetro o un campo del `record` pero, como indica Norell en su tesis [Nor07], es más fácil convertir un parámetro en un campo que al revés, por lo que en general se pone como parámetro a menos que se necesite que sea un campo por alguna razón.

El primer campo que aparece es `_≅m_`, el cual es una función que toma dos argumentos de tipo  $M$  y devuelve un elemento de tipo `Set`. Este campo es una relación binaria entre elementos del conjunto  $M$  y, como se indica en el segundo campo, debe ser una relación de equivalencia. En efecto, el campo `eqm` es una prueba de que la relación `_≅m_` es una relación de equivalencia, es decir que es reflexiva, simétrica y transitiva. Esto se prueba dando una instancia del `record IsEquivalence` provisto por la librería `Relation.Binary.Structures`. La razón de que se pidan estos dos campos es que para los siguientes se requiere una noción de igualdad, ya que algunos de ellos son ecuaciones entre elementos del conjunto  $M$  que deben cumplirse para que tal conjunto sea un monoide. La noción de igualdad tradicional (igualdad proposicional en Agda) no siempre es suficiente ya que, como se vio en la sección 3.3, a veces nos interesan otros tipos de equivalencias como, por ejemplo, la bisemejanza en el caso del tipo `delay`. Esto sucede con frecuencia cuando se trabaja con tipos coinductivos ya que una prueba de igualdad tradicional en general sería infinita. Es por esto que se decidió, tanto para esta como para las estructuras que siguen, que la noción de igualdad sea un campo, de manera que puedan definirse las estructuras con diferentes nociones de igualdad según sea conveniente.

El tercer campo, `zerom`, es un elemento particular del conjunto  $M$ , el cual será el elemento neutro de la operación binaria. Esta última se introduce en el cuarto campo, `_+m_`, como una función que toma dos elementos de  $M$  y devuelve un nuevo elemento del mismo conjunto.

Los últimos tres campos son los más interesantes y son los que hacen que, al dar un habitante del tipo `Monoid`, quede demostrado que tal habitante es efectivamente un monoide. El primero de ellos, `idl`, representa la prueba de que `zerom` es un neutro a izquierda respecto de la operación `+m`. El tipo de este campo representa una proposición que indica que para todo elemento  $x : M$ , se cumple la ecuación `(zerom +m x) ≅m x`. El siguiente, `idr`, es análogo a `idl` y representa la prueba de que `zerom` es neutro a derecha respecto de la operación `+m`. Por último, `assoc` representa la prueba de que la operación `+m` es asociativa. Esto es, para cualesquiera elementos dados  $x, y$  y  $z$  del conjunto  $M$ , se cumple que `(x +m (y +m z)) ≅m ((x +m y) +m z)`. Estas pruebas se dan en torno a la noción de igualdad introducida en el primer campo.

## 4.2. Formalización a nivel de Funtores y Mónadas

Siguiendo con el camino hacia la formalización de los monoides y mónadas concurrentes, se expondrán en esta sección las formalizaciones de las estructuras de funtor monoidal y mónada. La elección de estas estructuras se debe a que cada una de ellas aporta un ingrediente que luego aparecerá en la formalización de las mónadas concurrentes. La formalización de mónadas, por su parte, agrega la estructura monádica, `bind` y `return`, junto con las leyes de mónada. Por otro lado, el funtor monoidal introduce la operación `merge` y sus propiedades, que también aparecen en la formalización de mónadas concurrentes puesto que, como se expone en la definición 2.29,

estas son mónadas cuyo funtor subyacente tiene una estructura monoidal.

### 4.2.1. Formalización de Mónada

La formalización de las mónadas está dada, al igual que la de los monoides, por un **record** parametrizado. Sin embargo, en este caso el parámetro no es simplemente un conjunto sino que se trata de un funtor  $M : \text{Set} \rightarrow \text{Set}$ .

```
record Monad (M : Set → Set) : Set1 where
  constructor
  makeMonad
  field
    _≅m_ : ∀ {A} → M A → M A → Set
    eqm   : ∀ {A} → IsEquivalence (_≅m_ {A})
    return : ∀ {A : Set} → A → M A
    _>=>_   : ∀ {A B : Set} → M A → (A → M B) → M B
    law1   : ∀ {A B : Set} → (x : A) (f : A → M B)
              → (((return x) >=> f) ≅m (f x))
    law2   : ∀ {A} → (t : M A) → (t >=> return) ≅m t
    law3   : ∀ {A B C : Set} → (t : M A) (f : A → M B) (g : B → M C)
              → ((t >=> f) >=> g) ≅m (t >=> (λ x → f x >=> g))

open Monad public
```

Como se adelantó en la sección anterior, el primer campo de esta definición es una noción de igualdad. A diferencia de la que se pedía en **Monoid**, esta debe estar definida para elementos del conjunto  $M A$ , para cualquier  $A$ . De igual manera, la prueba de que esta relación es de equivalencia debe darse para un  $A$  arbitrario. La arbitrariedad de  $A$  se debe a que, como se puede ver en los siguientes campos, el funtor  $M$  se aplica sobre diversos conjuntos y se necesita la igualdad definida para todos los conjuntos sobre los que  $M$  se aplique. Esta necesidad se evidencia sobre todo al dar instancias de **Monad**.

El tercer campo es la función **return** usual de las mónadas. Esta toma un elemento de algún conjunto  $A$  arbitrario y lo encapsula en la mónada aplicando el funtor  $M$ . En la definición formal de las mónadas (ambas versiones) esta función representa el morfismo  $\eta : A \rightarrow M A$ . El campo que sigue, **\_>=>\_**, representa el operador *bind* de mónadas que toma un elemento de  $M A$  y una función que toma el resultado  $A$  y genera un elemento de  $M B$  y devuelve un  $M B$ . Esta función representa la secuenciación de computaciones y está definida en el estilo de la definición de mónadas como sistemas de extensión. En la definición 2.15 se define el operador **\_\*** que toma una función de tipo  $A \rightarrow M B$  y devuelve una función de tipo  $M A \rightarrow M B$ . El tipo del operador es por tanto **\_\*** :  $(A \rightarrow M B) \rightarrow M A \rightarrow M B$ . Si se da vuelta el orden de sus argumentos se obtiene el tipo de la función *bind*: **\_>=>\_** :  $M A \rightarrow (A \rightarrow M B) \rightarrow M B$ .

Los últimos tres campos representan las leyes de mónadas. Estos están dados por tipos que representan proposiciones. El primero de ellos, **law<sub>1</sub>**, representa la primera ley de mónadas que dice que para cualquier elemento  $x$  de un conjunto  $A$  y cualquier función  $f$  que dado un elemento de  $A$  y genere una computación de tipo  $M B$ , se debe cumplir que hacer el *bind* de **return**  $x$  con  $f$  debe dar el mismo resultado que aplicar  $f$  a  $x$ . Esta ley se corresponde con la siguiente ecuación de la definición formal:  $f^* \circ \eta_A = f$ . Cuando se da una instancia de **Monad** y se asigna al campo **law<sub>1</sub>** un habitante de este tipo, se está demostrando que las expresiones asignadas a los campos

`return` y `_>>=_` cumplen con la primera ley de mónadas respecto de la igualdad provista por el primer campo del `record`.

De la misma manera, `law2` representa la proposición que indica que se cumple la segunda ley de mónadas. Esta postula que dada una computación  $t$  de tipo  $M A$ , hacer `bind` de  $t$  con `return` es lo mismo que aplicar sólo  $t$ . Esta segunda ley se corresponde con la ecuación que indica que  $\eta_A^* = id_{MA}$ . El término  $\eta_A^*$  corresponde a la acción de aplicar a alguna computación  $t$  el operador `bind` seguido de `return`, mientras que del otro lado se tiene  $id_{MA}$  que aplicado a alguna computación arbitraria  $t$  da como resultado  $t$ .

Finalmente, el último campo, `law3`, representa la asociatividad del operador `_>>=_`. Esta denota que, dadas una computación  $t$  de tipo  $M A$  y dos funciones  $f$  y  $g$  que van de  $A$  en  $M B$  y de  $B$  en  $M C$  respectivamente, es lo mismo aplicar el `bind` a  $t$  con  $f$  y luego al resultado obtenido aplicarle el `bind` con  $g$ , que aplicarlo a  $t$  con la función que toma un  $x$  de tipo  $A$  y devuelve la aplicación de  $f x >>= g$ . Esta última ley se corresponde con la ecuación que indica que  $g^* \circ f^* = (g^* \circ f)^*$ .

#### 4.2.2. Formalización de Funtor Monoidal

Como se indica en la definición 2.20, un funtor monoidal es un funtor que cuenta con una estructura monoidal de manera que se cumplen ciertas ecuaciones de congruencia. Su formalización queda definida como un `record` que, al igual que `Monad`, está parametrizado por un funtor  $M : \text{Set} \rightarrow \text{Set}$ .

```
record MonoidalFunctor (M : Set → Set) : Set1 where
  constructor
  makeMonoidalFunctor
  field
    _≅m_ : ∀ {A} → M A → M A → Set
    eqm   : ∀ {A} → IsEquivalence (_≅m_ {A})
    unit   : M ⊤
    merge  : ∀ {A B : Set} → M A → M B → M (A × B)
    fmap   : ∀ {A B : Set} → (A → B) → M A → M B
    idr    : ∀ {A : Set} → (a : M A)
      → (merge a unit) ≅m (fmap (λ a → (a , tt)) a)
    idl    : ∀ {B : Set} → (b : M B)
      → (merge unit b) ≅m (fmap (λ b → (tt , b)) b)
    assoc  : ∀ {A B C : Set} → (a : M A) (b : M B) (c : M C)
      → (fmap (λ {((a , b) , c) → (a , (b , c))}) (merge (merge a b) c))
      ≅m (merge a (merge b c))
    - comm : ∀ {A B : Set} → (a : M A) (b : M B)
    - → merge a b ≅m fmap swap (merge b a)
```

`open MonoidalFunctor public`

Los primeros dos campos del `record` son iguales a los de `Monad` y brindan la noción de igualdad junto con la prueba de que tal noción es una relación de equivalencia.

Los dos campos que siguen conforman la estructura monoidal que el funtor requiere para ser un funtor monoidal. La transformación natural  $m : MA \times MB \rightarrow M(A \times B)$  está dada por el

campo **merge**. A diferencia de como se describe en la definición formal (2.20), en este caso **merge** no toma un elemento de  $M A \times M B$ , sino que toma ambos elementos por separado, es decir que se encuentra curriificada.

El otro ingrediente que se necesita para dar la estructura monoidal del funtor es el morfismo  $e : \mathbf{1} \rightarrow M \mathbf{1}$ . En la sección 2.2.3 se mencionó que en esta tesina se trabajaría siempre con la categoría **Set** considerando como objeto terminal el conjunto unitario  $\mathbf{1} = \{\star\}$ . Este conjunto tiene una representación propia en Agda y está dada por el tipo de dato  $\top$  definido en el módulo **Data.Unit.Base**. Este se define como un **record** sin campos con un único constructor llamado **tt**. En la formalización de funtor monoidal, en lugar de representar al morfismo  $e$  con una función de tipo  $\top \rightarrow M \top$ , este se representa directamente como un habitante del tipo  $M \top$  llamado **unit**. Esto es porque sólo hay un habitante de tipo  $\top$ , por lo que sólo habrá un habitante del tipo  $M \top$  y darlo como función sería redundante, ya que siempre habría que escribir **unit tt** puesto que no habría otro posible argumento para la función.

El siguiente campo de **MonoidalFuntor** es **fmap**. Como se describió en la definición 2.9, un funtor  $F$  de  $\mathcal{C}$  en  $\mathcal{D}$  no sólo asigna un objeto de  $\mathcal{D}$  a cada objeto de  $\mathcal{C}$ , sino que también asigna a cada morfismo  $f : A \rightarrow B \in \mathbf{mor} \mathcal{C}$  un morfismo  $F(f) : FA \rightarrow FB$ . El parámetro  $M$  sólo representa la asignación de objetos del funtor, asigna a cada objeto de **Set** otro objeto de **Set**. La asignación de morfismos no está definida. El rol de **fmap** es representar esta asignación, es decir que es un mapeo que, dada una función que va de  $A$  en  $B$ , devuelve otra función que va de  $M A$  en  $M B$ . Este campo es necesario para poder definir las ecuaciones que la estructura monoidal debe cumplir, las cuales están dadas en los últimos tres campos del **record**.

Las primeras dos, **idr** e **idl**, son análogas puesto que representan las pruebas de que **unit** es neutro, a derecha e izquierda respectivamente, respecto de la operación **merge**. Por esta razón se explicará en profundidad sólo una de ellas, la elegida será **idr**. Volviendo a la definición formal, se muestra a continuación la ecuación correspondiente a **idr** junto con su versión vista como un diagrama conmutativo, el cual puede resultar más facil de comprender a la vista.

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

$$\begin{array}{ccc} FA \times \mathbf{1} & \xrightarrow{id_{FA} \times e} & FA \times F\mathbf{1} \\ \downarrow \pi_1 & & \downarrow m_{A, \mathbf{1}} \\ FA & \xleftarrow{F(\pi_1)} & F(A \times \mathbf{1}) \end{array}$$

Teniendo en cuenta que ya no se tiene  $e$  que pasa de  $\mathbf{1}$  a  $F\mathbf{1}$ , sino que se tiene directamente un elemento de  $F\mathbf{1}$ , la esquina superior izquierda del diagrama desaparece, quedando como resultado el diagrama 4.1. Si luego se reemplaza  $F$  por  $M$ ,  $\mathbf{1}$  por  $\top$  y  $m_{A, \mathbf{1}}$  por **merge**, se obtiene el diagrama 4.2.

$$\begin{array}{ccc} & FA \times F\mathbf{1} & \\ & \swarrow \pi_1 \quad \downarrow m_{A, \mathbf{1}} & \\ FA & \xleftarrow{F(\pi_1)} & F(A \times \mathbf{1}) \end{array} \quad (4.1)$$

$$\begin{array}{ccc} & MA \times M\top & \\ & \swarrow \pi_1 \quad \downarrow \text{merge} & \\ MA & \xleftarrow{M(\pi_1)} & M(A \times \top) \end{array} \quad (4.2)$$

El segundo diagrama da lugar a la ecuación:  $M(\pi_1) \circ \text{merge} \cong_m \pi_1$ . Si en lugar de utilizar  $\pi_1$  se considera su inversa  $\pi_1^{-1}$ , el diagrama y su ecuación correspondiente pasan a quedar como sigue:

$$\begin{array}{ccc} & MA \times M\top & \\ \pi_1^{-1} \nearrow & \downarrow \text{merge} & \\ MA & \xrightarrow{M(\pi_1^{-1})} & M(A \times \top) \end{array} \quad (4.3)$$

$$\text{merge} \circ \pi_1^{-1} \cong_m M(\pi_1^{-1}) \quad (4.4)$$

Ahora, como en realidad **merge** no toma un producto cartesiano sino ambos elementos por separado, no es necesario utilizar  $\pi_1^{-1}$  antes de **merge**. Si se considera por último que  $\pi_1^{-1}$  se representa en Agda como  $(\lambda a \rightarrow (a, \text{tt}))$  y que la asignación de morfismos del funtor  $M$  está dada por el campo **fmap**, se obtiene finalmente la ecuación **merge**  $\cong_m$  **fmap**  $(\lambda a \rightarrow (a, \text{tt}))$ , la cual al agregarle los argumentos correspondientes forma la ecuación del campo **idr**:

$$(\text{merge } a \text{ unit}) \cong_m (\text{fmap}(\lambda a \rightarrow (a, \text{tt})) a).$$

Queda por analizar el campo **assoc**. Como su nombre lo indica, este representa la prueba de que la operación **merge** es asociativa. En la definición formal se requiere la siguiente ecuación en la cual  $\alpha$  representa la asociatividad del producto cartesiano.

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

$\alpha$  se representa en Agda como la asignación  $(\lambda \{((a, b), c) \rightarrow (a, (b, c))\})$ . Luego  $F(\alpha)$  en se escribe en Agda como: **fmap**  $(\lambda \{((a, b), c) \rightarrow (a, (b, c))\})$  siguiendo la representación dada.

Por otro lado,  $m_{X \times Y, Z} \circ (m_{X, Y} \times id_{FZ})$  se traduce en la formalización propuesta aplicada a sus argumentos como **(merge (merge a b) c)**. La aplicación **(merge a b)** se corresponde con  $m_{X, Y}$  e  $id_{FZ}$  se refleja simplemente en la variable  $c$  ya que aplicar la función identidad sobre ella da el mismo resultado. No es necesario generar la función producto de estas dos puesto que **merge** toma sus argumentos por separado. Finalmente, la segunda aplicación de **merge** se corresponde con  $m_{X \times Y, Z}$ .

En el otro lado de la ecuación, de manera análoga,  $m_{X, Y \times Z} \circ (id_{FX} \times m_{Y, Z})$  se corresponde con las siguientes aplicaciones: **(merge a (merge b c))**. Quedaría por agregar la aplicación de  $\alpha$  al principio, pero esta no es necesaria en este lado de la ecuación puesto que **merge** toma sus argumentos por separado, por lo que no hay ningún producto cartesiano que haya que dar vuelta.

Uniendo todas las partes traducidas y tomando la composición de funciones simplemente como la aplicación de una función al resultado de otra, se obtiene la ecuación presente en **assoc**:

$$(\text{fmap } (\lambda \{((a, b), c) \rightarrow (a, (b, c))\})) (\text{merge } (\text{merge } a \text{ b}) \text{ c}) \cong_m (\text{merge } a \text{ (merge b c)}).$$

En la definición formal hay una condición extra para los funtores monoidales que dice que si la estructura monoidal es además compatible con  $\gamma$ , la cual intercambia los elementos del producto cartesiano, entonces el funtor monoidal es simétrico. Si se piensa al funtor monoidal como un mecanismo para encapsular efectos de computaciones, entonces que el funtor sea simétrico significaría que el orden de los efectos no importa, es decir que los efectos conmutan. Esta condición extra se refleja en el campo final que se encuentra comentado, **comm**, el cual postula la conmutatividad del operador **merge**. Esta proposición indica que para cualesquiera computaciones  $a$  y  $b$  de tipo  $M A$  y  $M B$  respectivamente, la aplicación **merge a b** da el mismo resultado que la aplicación **merge b a** seguida de un **swap** que representa a la función  $\gamma$ . Agregando este último campo se obtiene entonces la formalización de los funtores monoidales simétricos.

### 4.3. Formalización de Monoides Concurrentes

Como paso previo a la formalización de las mónadas concurrentes, se presenta en esta sección la formalización de los monoides concurrentes. Estos son una mezcla entre los monoides y los funtores monoidales e introducen un ingrediente nuevo que luego estará presente también en la formalización de las mónadas concurrentes: la ley de intercambio. Esta formalización está dada

también por un tipo `record` parametrizado, donde el parámetro es, al igual que en el caso de los monoides, un conjunto.

```
record ConcurrentMonoid (M : Set) : Set1 where
  constructor
  makeConcurrentMonoid
  field
    _≅m_      : M → M → Set
    eqm       : IsEquivalence _≅m_
    _≲m_      : M → M → Set
    porderm   : IsPartialOrder _≅m_ _≲m_
    zerom     : M
    _+_m      : M → M → M
    scomp≲m : (x y z w : M) → x ≲m z → y ≲m w → (x +m y) ≲m (z +m w)
    sidl       : (x : M) → (zerom +m x) ≅m x
    sidr       : (x : M) → (x +m zerom) ≅m x
    sassoc     : (x : M) (y : M) (z : M) → (x +m (y +m z)) ≅m ((x +m y) +m z)
    maxm      : M → M → M
    mcomp≲m : (x y z w : M) → x ≲m z → y ≲m w → (maxm x y) ≲m (maxm z w)
    midl       : (x : M) → (maxm zerom x) ≅m x
    midr       : (x : M) → (maxm x zerom) ≅m x
    massoc     : (x : M) (y : M) (z : M) → ((maxm (maxm x y) z) ≅m (maxm x (maxm y z)))
    mcomm      : (x y : M) → (maxm x y) ≅m (maxm y x)
    ichange    : (x : M) (y : M) (z : M) (w : M)
                  → (maxm (x +m y) (z +m w)) ≲m ((maxm x z) +m (maxm y w))

open ConcurrentMonoid public
```

En la definición 2.23 se presenta a un bimonoides ordenado como un conjunto parcialmente ordenado  $(A, \sqsubseteq)$  junto con dos estructuras monoidales  $(A, ;, \text{skip})$  y  $(A, *, \text{nothing})$  tales que  $;$  y  $*$  son compatibles con  $\sqsubseteq$  y  $*$  es conmutativa. Luego, en la definición 2.24 se define a un monoides concurrente como un bimonoides ordenado tal que los neutros coinciden y se cumple la ley de intercambio.

Al igual que en las demás estructuras, los primeros dos campos de `Concurrent Monoid` representan la noción de igualdad junto con la prueba de que esta es una relación de equivalencia. De manera similar, el tercer y cuarto campo introducen una relación de orden junto con una prueba de que esta es un orden parcial, es decir que es reflexiva, antisimétrica y transitiva. Esta relación se corresponde con el orden opuesto al orden  $\sqsubseteq$  del conjunto que se pide en la definición formal, es decir que  $x \preceq_m y \Leftrightarrow y \sqsubseteq x$ . La decisión de tomar el orden opuesto se debe a los modelos con los que se trabajó en esta tesina, en el siguiente capítulo se comprenderá mejor esta decisión al dar instancias de esta estructura. La relación de orden es necesaria, sobre todo, debido a la presencia de la ley de intercambio, la cual está dada, a diferencia de las demás propiedades, como una desigualdad. Se requiere entonces que exista un orden sobre el conjunto  $M$  y este debe ser un orden parcial respecto de la noción de igualdad previamente introducida. Esto significa que las propiedades de reflexividad, antisimetría y transitividad están demostradas respecto de  $\_ \equiv_m \_$ . La prueba `porderm` de que la relación  $\_ \preceq_m \_$  es un orden parcial está dada por un habitante de `IsPartialOrder`, el cual es un tipo `record` definido en el módulo `Relation.Binary.Structures`.

Los siguientes dos campos,  $\_ +_m \_$  y `zerom`, representan la primera estructura monoidal del bimonoides ordenado, es decir, el operador  $;$  y su neutro `skip`. La prueba de que el operador es



compatible con la relación de orden se introduce a continuación en el campo  $\text{scomp}_{\lesssim_m}$ . Este indica que si se tienen cuatro valores  $x, y, z$  y  $w$  de tipo  $M$ , tales que  $x \lesssim_m z$  e  $y \lesssim_m w$ , entonces la suma de los menores es menor o igual a la suma de los mayores:  $(x +_m y) \lesssim_m (z +_m w)$ .

Seguidamente, se presentan las leyes o propiedades de la primera estructura monoidal:  $\text{sidl}$  y  $\text{sidr}$  son las pruebas de que  $\text{zero}_m$  es neutro de la operación  $+_m$ , a izquierda y derecha respectivamente, y  $\text{sassoc}$  es la prueba de que dicha operación es asociativa. Estos tres campos son iguales a los que aparecen en la formalización de monoide, la única diferencia es que se antepone la letra  $s$  para indicar que estas propiedades pertenecen a la suma.

A continuación se introduce otra operación binaria sobre el conjunto  $M$ :  $\text{max}_m$ . Esta operación junto con  $\text{zero}_m$  constituyen la segunda estructura monoidal del conjunto. El neutro es el mismo en ambas estructuras debido a que es un monoide concurrente. Se puede pensar a  $\text{max}_m$  como una versión simplificada de la operación  $\text{merge}$  de los funtores monoidales. Al igual que para la suma, se da la prueba de que este segundo operador es también compatible con la relación de orden. Esta prueba está representada por el campo  $\text{mcomp}_{\lesssim_m}$ , cuyo tipo es muy similar a  $\text{scomp}_{\lesssim_m}$ , donde la operación pasa a ser  $\text{max}_m$  en lugar de la suma. Los campos que siguen son pruebas de que la estructura  $(\text{max}_m, \text{zero}_m)$  es monoidal:  $\text{midl}$  y  $\text{midr}$  representan pruebas de que  $\text{zero}_m$  es neutro respecto de  $\text{max}_m$ , a izquierda y derecha respectivamente, y  $\text{massoc}$  representa la prueba de que dicha operación es asociativa. Se antepone la  $m$  delante para indicar que estas propiedades pertenecen a  $\text{max}_m$ . Se agrega para esta operación una propiedad extra: la conmutatividad. Esta es requerida para que el monoide sea concurrente y está representada por el campo  $\text{mcomm}$  que indica que, dados dos valores  $x$  e  $y$  de tipo  $M$ , siempre se cumple que  $(\text{max}_m x y) \cong_m (\text{max}_m y x)$ .

El último campo de esta estructura representa la ley de intercambio. El tipo del campo  $\text{ichange}$  se obtiene de forma casi directa de la ecuación de la ley de intercambio introducida en la definición 2.24:  $(x * z); (y * w) \sqsubseteq (x; y) * (z; w)$ . Si se reemplazan los operadores  $*$  y  $;$  por  $\text{max}_m$  y  $+_m$ , respectivamente, y se invierte el orden de la desigualdad como se explicó previamente, se llega a la siguiente ecuación:  $(\text{max}_m (x +_m y) (z +_m w)) \lesssim_m ((\text{max}_m x z) +_m (\text{max}_m y w))$ .

## 4.4. Formalización de Mónadas Concurrentes

Se introduce a continuación la formalización de las mónadas concurrentes. Esta estará dada, como en las estructuras algebraicas anteriores, por un tipo  $\text{record}$  parametrizado. Como en el caso de la formalización de mónadas, el parámetro del  $\text{record}$   $\text{ConcurrentMonad}$  será una asignación  $M : \text{Set} \rightarrow \text{Set}$ , la cual representa la asignación de objetos a objetos de un funtor  $M : \text{Set} \rightarrow \text{Set}$ .

$\text{record ConcurrentMonad } (M : \text{Set} \rightarrow \text{Set}) : \text{Set}_1 \text{ where}$

  constructor

$\text{makeConcurrentMonad}$

  field

$\_ \cong_m \_ : \forall \{A\} \rightarrow M A \rightarrow M A \rightarrow \text{Set}$   
      $\text{eq}_m : \forall \{A\} \rightarrow \text{IsEquivalence } (\_ \cong_m \_ \{A\})$   
      $\_ \lesssim_m \_ : \forall \{A\} \rightarrow M A \rightarrow M A \rightarrow \text{Set}$   
      $\text{porder}_m : \forall \{A\} \rightarrow \text{IsPartialOrder } (\_ \cong_m \_ \{A\}) (\_ \lesssim_m \_ \{A\})$   
      $\text{return} : \forall \{A : \text{Set}\} \rightarrow A \rightarrow M A$   
      $\_ \gg= \_ : \forall \{A B : \text{Set}\} \rightarrow M B \rightarrow (B \rightarrow M A) \rightarrow M A$   
      $\text{bcomp}_{\lesssim_m} : \forall \{A B : \text{Set}\} \rightarrow (a_1 a_2 : M A) \rightarrow (f_1 f_2 : A \rightarrow M B) \rightarrow a_1 \lesssim_m a_2$   
        $\rightarrow (\forall (a : A) \rightarrow (f_1 a) \lesssim_m (f_2 a)) \rightarrow (a_1 \gg= f_1) \lesssim_m (a_2 \gg= f_2)$



$$\begin{aligned}
\text{monad}_1 & : \forall \{A \ B : \text{Set}\} \rightarrow (x : B) (f : B \rightarrow M A) \\
& \quad \rightarrow (((\text{return } x) \gg= f) \cong_m (f x)) \\
\text{monad}_2 & : \forall \{A\} \rightarrow (t : M A) \rightarrow (t \gg= \text{return}) \cong_m t \\
\text{monad}_3 & : \forall \{A \ B \ C : \text{Set}\} \rightarrow (t : M C) (f : C \rightarrow M B) (g : B \rightarrow M A) \\
& \quad \rightarrow ((t \gg= f) \gg= g) \cong_m (t \gg= (\lambda x \rightarrow f x \gg= g)) \\
\text{unit} & : M \top \\
\text{merge} & : \forall \{A \ B : \text{Set}\} \rightarrow M A \rightarrow M B \rightarrow M (A \times B) \\
\text{mcomp}_{\lesssim_m} & : \forall \{A \ B : \text{Set}\} \rightarrow (a_1 \ a_2 : M A) \rightarrow (b_1 \ b_2 : M B) \rightarrow a_1 \lesssim_m a_2 \rightarrow b_1 \lesssim_m b_2 \\
& \quad \rightarrow (\text{merge } a_1 \ b_1) \lesssim_m (\text{merge } a_2 \ b_2) \\
\text{id}_r & : \forall \{A : \text{Set}\} \rightarrow (a : M A) \\
& \quad \rightarrow (\text{merge } a \ \text{unit}) \cong_m (a \gg= (\lambda a \rightarrow \text{return } (a, \text{tt}))) \\
\text{id}_l & : \forall \{B : \text{Set}\} \rightarrow (b : M B) \\
& \quad \rightarrow (\text{merge } \text{unit} \ b) \cong_m (b \gg= (\lambda b \rightarrow \text{return } (\text{tt}, b))) \\
\text{assoc} & : \forall \{A \ B \ C : \text{Set}\} \rightarrow (a : M A) (b : M B) (c : M C) \\
& \quad \rightarrow ((\text{merge } (\text{merge } a \ b) \ c) \gg= (\lambda \{(a, b), c\} \rightarrow \text{return } (a, (b, c)))) \\
& \quad \quad \quad \cong_m (\text{merge } a \ (\text{merge } b \ c)) \\
\text{comm} & : \forall \{A \ B : \text{Set}\} \rightarrow (a : M A) (b : M B) \\
& \quad \rightarrow (\text{merge } a \ b) \cong_m ((\text{merge } b \ a) \gg= (\lambda \{(a, b)\} \rightarrow \text{return } (b, a))) \\
\text{ichange} & : \forall \{A \ B \ C \ D : \text{Set}\} \rightarrow (a : M A) (b : M B) (f : A \rightarrow M C) (g : B \rightarrow M D) \\
& \quad \rightarrow (\text{merge } (a \gg= f) (b \gg= g)) \\
& \quad \quad \quad \lesssim_m ((\text{merge } a \ b) \gg= (\lambda \{(a, b)\} \rightarrow (\text{merge } (f a) (g b))))
\end{aligned}$$

open ConcurrentMonad public

Siguiendo el ejemplo de las mónadas, el primer campo de esta estructura es una noción de igualdad sobre el conjunto  $M A$ , para un  $A$  arbitrario. De igual manera, el segundo campo es la prueba de que la relación anterior es de equivalencia, y también debe darse para un  $A$  arbitrario. Como se explicó anteriormente, es necesario que el conjunto  $A$  pueda tomar diferentes valores ya que el funtor  $M$  se aplicará a distintos conjuntos a lo largo de la estructura. Es por esta misma razón que los dos campos que siguen también deben definirse para cualquier conjunto  $A$ .

En la definición 2.29 se expone que para que una mónada sea concurrente debe ser primero una mónada monoidal ordenada. La definición 2.28, por su lado, indica que para que una mónada sea una mónada monoidal ordenada debe contar con una estructura de funtor monoidal simétrico  $(m, e)$  y una relación de orden compatible con la misma. Es así como, de manera similar a la formalización de monoides, el tercer campo introduce una relación de orden entre elementos de  $M A$ , para algún  $A$  dado. A continuación, en el cuarto campo, se da la prueba de que dicha relación es un orden parcial, es decir que es reflexiva, antisimétrica y transitiva. Estas pruebas se dan en torno a la noción de igualdad definida previamente. Al igual que en los monoides concurrentes, el orden considerado es el opuesto al utilizado en la teoría, de manera que  $x \lesssim_m y \Leftrightarrow y \sqsubseteq x$ .

Los campos siguientes introducen la estructura monádica. Estos están dados, al igual que en la formalización de mónada, siguiendo la definición de mónadas como sistemas de extensión (definición 2.15). La función `return` toma un elemento de un conjunto  $A$  arbitrario y lo encapsula en la mónada, devolviendo un habitante del tipo  $M A$ , mientras que el operador `>>=` sirve para secuenciar dos computaciones.

Antes de pasar a las leyes de las mónadas, se introduce la compatibilidad del operador `>>=` con la relación de orden. Esta propiedad está dada por el campo `bcomplesssim_m`. En la definición 2.26, se define una mónada ordenada como una mónada que tiene un orden  $\sqsubseteq_I$  sobre  $TI$  para cada

conjunto  $I$ , donde este orden enriquece a los morfismos  $A \rightarrow TB$  con un orden  $\sqsubseteq_{A,B}$  tal que  $f \sqsubseteq_{A,B} g$  si y sólo si  $\forall a : \mathbf{1} \rightarrow A, f \circ a \sqsubseteq_B g \circ a$ . En Agda, como ya se mencionó anteriormente, no tiene sentido considerar una función que toma elementos del conjunto  $\mathbf{1}$  ( $\top$  en Agda) puesto que este tiene un único elemento. Luego, en lugar de tomar morfismos  $a : \mathbf{1} \rightarrow A$ , se toman directamente valores  $a : A$ . Se considera entonces que una función  $f_1 : A \rightarrow MB$  es menor o igual a una función  $f_2 : A \rightarrow MB$  si y sólo si para todo  $a : A$  se cumple que  $(f_1 \ a) \lesssim_m (f_2 \ a)$ . La compatibilidad del operador  $\gg=$  con el orden  $\lesssim_m$  se define entonces de la siguiente manera: dados dos valores  $a_1$  y  $a_2$  de tipo  $MA$  y dos funciones  $f_1$  y  $f_2$  de tipo  $A \rightarrow MB$ , si se tiene que  $a_1 \lesssim_m a_2$  y  $\forall (a : A) \rightarrow (f_1 \ a) \lesssim_m (f_2 \ a)$ , entonces  $(a_1 \gg= f_1) \lesssim_m (a_2 \gg= f_2)$ .

Los campos `monad1`, `monad2` y `monad3` representan las tres leyes de mónadas y son iguales a los campos `law1`, `law2` y `law3` de `Monad`. Los primeros dos indican que `return` es neutro a derecha e izquierda de  $\gg=$  y el último indica la asociatividad de dicho operador.

Una vez establecida la estructura monádica, se pasa a introducir la estructura monoidal que se requiere en la definición 2.28 para que la mónada sea una mónada monoidal ordenada. El elemento neutro es un morfismo  $e : \mathbf{1} \rightarrow M\mathbf{1}$ . Como se mencionó anteriormente, el objeto terminal  $\mathbf{1} = \{\star\} \in \mathbf{ob \ Set}$  se representa en Agda como el tipo  $\top$ , cuyo único habitante es `tt`. Al ser un conjunto de un único elemento, al igual que en el caso de la formalización de funtor monoidal,  $e$  se define como un campo `unit` de tipo  $M \top$  en lugar de como una función de tipo  $\top \rightarrow M \top$  ya que hay un único argumento posible para la función. Según la definición 2.29, para que la mónada sea una mónada concurrente, el morfismo  $e$  debe ser igual a  $\eta_1 : \mathbf{1} \rightarrow M\mathbf{1}$ . Considerando  $\mathbf{1} = \top$ , luego  $e$  debería ser igual a `return { $\top$ }` :  $\top \rightarrow M \top$ . Como  $\top$  tiene un único habitante, se obtiene que `unit` debe ser igual a `return tt` :  $M \top$ . **Cómo defino `unit = return tt` dentro de la estructura?**

Continuando con la estructura monoidal, el siguiente campo es la función `merge`, la cual representa la transformación natural  $m$ . Este campo está definido de igual manera que el `merge` de `MonoidalFunctor`. En dicha estructura, a continuación del `merge` se introduce el campo `fmap`, el cual representa la asignación de morfismos a morfismos del funtor  $M$ . En el caso de las mónadas concurrentes este campo no es necesario puesto que puede construirse tal asignación a partir del operador  $\gg=$ . Sea  $f : A \rightarrow B$  una función, luego  $(\lambda ma \rightarrow ma \gg= (\lambda a \rightarrow \text{return } f \ a))$  es una función de tipo  $MA \rightarrow MB$ . De esta manera la asignación de morfismos a morfismos del funtor queda definida. **Comentario: demostrar equivalencia? Se demostraría para un caso particular igual, no? Porque no tengo una definición de `fmap` concreta acá**

El campo que sigue, `mcompm`, representa la compatibilidad del operador `merge` con la relación de orden  $\lesssim_m$ . Dados  $a_1$  y  $a_2$  de tipo  $MA$  y  $b_1$  y  $b_2$  de tipo  $MB$ , si se tiene que  $a_1 \lesssim_m a_2$  y  $b_1 \lesssim_m b_2$ , luego se debe cumplir que  $(\text{merge } a_1 \ b_1) \lesssim_m (\text{merge } a_2 \ b_2)$ .

Es el turno ahora de las propiedades que deben cumplirse para que `merge` y `unit` formen una estructura monoidal. Estas propiedades son tres y están definidas de manera similar a las de la formalización de funtor monoidal, salvo por el uso del campo `fmap`. `idr` representa la proposición que indica que `unit` es un neutro a derecha del operador `merge`. En el record `MonoidalFunctor` esta propiedad tiene tipo  $\forall \{A : \mathbf{Set}\} \rightarrow (a : MA) \rightarrow (\text{merge } \text{unit } a) \cong_m (\text{fmap } (\lambda a \rightarrow (a, \text{tt})) \ a)$ . Reemplazando `fmap`  $(\lambda a \rightarrow (a, \text{tt}))$  por la función propuesta anteriormente, donde se sustituye  $f$  por  $(\lambda a \rightarrow (a, \text{tt}))$ , quedaría el siguiente tipo:

$$\forall \{A : \mathbf{Set}\} \rightarrow (a : MA) \rightarrow (\text{merge } \text{unit } a) \cong_m ((\lambda ma \rightarrow ma \gg= (\lambda a \rightarrow \text{return } (\lambda a \rightarrow (a, \text{tt})) \ a)) \ a)$$

Si se reduce la aplicación  $(\lambda a \rightarrow (a, \text{tt})) \ a$ , se obtiene un tipo un poco más reducido:

$$\forall \{A : \text{Set}\} \rightarrow (a : M A) \rightarrow (\text{merge unit } a) \cong_m ((\lambda ma \rightarrow ma \gg= (\lambda a \rightarrow \text{return } (a, \text{tt}))) a)$$

Si se reduce finalmente la aplicación  $(\lambda ma \rightarrow ma \gg= (\lambda a \rightarrow \text{return } (a, \text{tt}))) a$ , se obtiene el tipo del campo **idr** en la estructura:

$$\forall \{A : \text{Set}\} \rightarrow (a : M A) \rightarrow (\text{merge unit } a) \cong_m (a \gg= (\lambda a \rightarrow \text{return } (a, \text{tt})))$$

De manera análoga, si se reemplaza **fmap** por su versión construida con *bind* en el tipo del campo **idl** de **MonoidalFunctor** y luego se reducen las aplicaciones, se obtiene el tipo del campo **idl** de **ConcurrentMonad**. Este representa la propiedad que postula que **unit** es neutro a izquierda del operador **merge**.

La última de estas tres propiedades, **assoc**, representa la asociatividad del operador **merge**. En la estructura de funtor monoidal esta ley estaba definida con el siguiente tipo:

$$\forall \{A B C : \text{Set}\} \rightarrow (a : M A) (b : M B) (c : M C) \rightarrow \\ (\text{fmap } (\lambda \{((a, b), c) \rightarrow (a, (b, c))\}) (\text{merge } (\text{merge } a b) c)) \cong_m (\text{merge } a (\text{merge } b c))$$

Reemplazando **fmap**  $(\lambda \{((a, b), c) \rightarrow (a, (b, c))\})$  por la función equivalente a **fmap**, donde la *f* sería sustituida por la función  $(\lambda \{((a, b), c) \rightarrow (a, (b, c))\})$ , el tipo queda como sigue:

$$\forall \{A B C : \text{Set}\} \rightarrow (a : M A) (b : M B) (c : M C) \rightarrow \\ ((\lambda ma \rightarrow ma \gg= (\lambda a \rightarrow \text{return } (\lambda \{((a, b), c) \rightarrow (a, (b, c))\}) a)) (\text{merge } (\text{merge } a b) c)) \\ \cong_m (\text{merge } a (\text{merge } b c))$$

La aplicación  $(\lambda \{((a, b), c) \rightarrow (a, (b, c))\}) a$  se puede reducir, pero para esto es necesario hacer *pattern matching* en la variable *a* de manera que se pueda acceder a sus componentes, ya que esta es un elemento de un producto cartesiano. El resultado de abrir la variable *a* y luego reducir la aplicación es el tipo:

$$\forall \{A B C : \text{Set}\} \rightarrow (a : M A) (b : M B) (c : M C) \rightarrow \\ ((\lambda ma \rightarrow ma \gg= (\lambda \{((a, b), c) \rightarrow \text{return } (a, (b, c))\})) (\text{merge } (\text{merge } a b) c)) \\ \cong_m (\text{merge } a (\text{merge } b c))$$

Se puede reducir ahora el lado izquierdo de la ecuación, reemplazando la variable *ma* por  $(\text{merge } (\text{merge } a b) c)$ . El resultado de hacer este reemplazo es el siguiente tipo, el cual es el tipo del campo **assoc** de **ConcurrentMonad**:

$$\forall \{A B C : \text{Set}\} \rightarrow (a : M A) (b : M B) (c : M C) \rightarrow \\ ((\text{merge } (\text{merge } a b) c) \gg= (\lambda \{((a, b), c) \rightarrow \text{return } (a, (b, c))\})) \cong_m (\text{merge } a (\text{merge } b c))$$

Como se mencionó anteriormente, para que la mónada sea una mónada monoidal ordenada, esta debe contar con una estructura de funtor monoidal simétrico. Para que sea simétrico, además de las leyes que se probaron sobre la estructura monoidal, debe haber una propiedad extra: la conmutatividad del operador **merge**. Esta propiedad se representa con el campo **comm**. El tipo de este campo se construye de manera similar al anterior, donde de un lado se tiene simplemente  $(\text{merge } a b)$ , y del otro se aplica  $(\text{merge } b a)$  seguido de una aplicación de  $\gg=$  con una función que da vuelta el orden del producto cartesiano.

Queda por analizar el último campo de la estructura, el cual postula la relación que deben tener los operadores  $\gg=$  y **merge** para que la mónada sea concurrente. Este se denomina **ichange**

y representa la ley de intercambio. Sean  $a' : \mathbf{1} \rightarrow MA$ ,  $b' : \mathbf{1} \rightarrow MB$ ,  $f : A \rightarrow MC$  y  $g : B \rightarrow MD$ . Comentario: puse que  $a'$  y  $b'$  sean  $\mathbf{1} \rightarrow MX$  porque en la formalización en realidad son directamente  $MX$ , pero en la teoría la ley se da con todas funciones. Cierra todo pero queda medio sin justificar por qué inicialmente los tomo así y no como funciones cualesquiera, como podría mejorar esto? En la definición 2.29, se presenta la ley de intercambio de la siguiente manera:  $(f \star g) \bullet (a' \star b') \sqsubseteq (f \bullet a') \star (g \bullet b')$ , donde  $f \star g = m \circ (f \times g)$  y  $f \bullet g = f^* \circ g$ . Esta ley se puede reescribir entonces como:

$$(m \circ (f \times g))^* \circ (m \circ (a' \times b')) \sqsubseteq (m \circ ((f^* \circ a') \times (g^* \circ b')))$$

Como ya sucedió varias veces a lo largo del proceso de formalización, se tienen en este caso dos funciones que tienen como dominio el conjunto terminal  $\mathbf{1}$ . De igual manera que en las demás ocasiones, estas serán representadas en la formalización de manera simplificada, en el caso de  $a'$  se tomará un  $a : M A$ , y en el caso de  $b'$  se tomará un  $b : M B$ . Haciendo estos cambios, la fórmula se reescribe como se muestra a continuación:

$$(m \circ (f \times g))^* (m (a , b)) \sqsubseteq (m \circ ((f^* a) \times (g^* b)))$$

La transformación natural  $m$  se reemplaza en la formalización dada con el operador **merge**, el cual no toma un par ordenado sino dos elementos por separado. Haciendo este reemplazo se obtiene la siguiente fórmula:

$$(\text{merge} \circ (f \times g))^* (\text{merge } a \ b) \sqsubseteq (\text{merge } (f^* a) (g^* b))$$

El operador  $\_*$ , por su parte, está representado por la función  $\gg=$ , donde la expresión  $h^*x$  es equivalente a escribir  $x \gg= h$ . Luego de hacer esta sustitución la fórmula queda como sigue:

$$(\text{merge } a \ b) \gg= (\text{merge} \circ (f \times g)) \sqsubseteq (\text{merge } (a \gg= f) (b \gg= g))$$

La aplicación  $(\text{merge} \circ (f \times g))$  se reescribe en Agda como una función que toma un par  $(a , b)$ , aplica la función  $f$  a  $a$  y la función  $g$  a  $b$ , y luego el operador **merge** a los dos resultados obtenidos. Es decir, se expresa de la siguiente manera:  $(\lambda \{ (a , b) \rightarrow (\text{merge } (f a) (g b)) \})$ . Haciendo esta sustitución y cambiando el orden  $\sqsubseteq$  por el campo  $\lesssim_m$  (invirtiendo el orden de la desigualdad puesto que se toma el orden opuesto), se obtiene finalmente el tipo del campo **ichange**:

$$(\text{merge } (a \gg= f) (b \gg= g)) \lesssim_m (\text{merge } a \ b) \gg= (\lambda \{ (a , b) \rightarrow (\text{merge } (f a) (g b)) \})$$

## Capítulo 5

# El caso de la Mónada Delay

Como se mencionó en el capítulo 3, el tipo *delay* fue introducido por Capretta [Cap05] para representar la posible no terminación de programas en la teoría de tipos de Martin-Löf. Sus habitantes son valores “demorados”, los cuales pueden no terminar nunca. El objetivo de este capítulo es hacer un análisis de este tipo respecto de las estructuras algebraicas previamente definidas.

Inicialmente se dará la definición de este tipo en Agda, utilizando para ello la notación musical para tipos coinductivos que fue descripta en la sección 3.4.1. Junto con el tipo se definirán diversas relaciones sobre él, entre las que se encuentran las bisemejanzas débil y fuerte y una relación de orden. La implementación del tipo *delay* utilizada fue extraída de la librería [Effect.Monad.Partiality](#).

Una vez introducido el tipo, se demostrará que este tiene estructura de mónada y de funtor monoidal. Para esto se crearán instancias de las estructuras correspondientes para el tipo *delay*. El objetivo final de este capítulo es probar o refutar que el tipo definido tiene estructura de mónada concurrente. La principal dificultad para esto subyace en la prueba de la ley de intercambio, ya que los demás ingredientes están presentes en las pruebas de mónada y funtor monoidal.

### 5.1. Definición del tipo *delay* con notación musical

El tipo *delay* se define con notación musical mediante una estructura **data** parametrizada. El parámetro será un conjunto  $A$  de tipo **Set**, el cual representa el tipo de los valores de retorno (en caso de que el programa termine). Dado entonces un  $A : \mathbf{Set}$ , el tipo  $A \perp$  representa el tipo **DA** definido en la sección 3.3.

```
data  $\_ \perp$  ( $A : \mathbf{Set}$ ) : Set where  
  now : ( $x : A$ )  $\rightarrow A \perp$   
  later : ( $x : \infty (A \perp)$ )  $\rightarrow A \perp$ 
```

El constructor **now** toma un valor  $x : A$  y genera un valor de tipo  $A \perp$ . La expresión **now**  $x$  representa un programa que simplemente retorna el valor  $x$  sin demoras. El constructor **later** toma un  $x$  de tipo  $\infty (A \perp)$ , es decir un valor de tipo  $A \perp$  suspendido o demorado. Que el valor este suspendido o demorado implica que puede ser potencialmente infinito. El constructor **later** retorna otro valor de tipo  $A \perp$ . Intuitivamente, lo que hace este constructor es “agregar una demora” al valor recibido.

Un habitante del tipo  $A \perp$  puede ser una secuencia finita de constructores **later** que finalmente retorna un valor  $x : A$ , es decir algo del estilo **later** ( $\#$  (**later** ( $\#$  ... (**now**  $x$ ) ...))); o puede ser una secuencia infinita de constructores **later** que nunca retorna. Este último es el caso del valor **never** que se define a continuación.

```

never : A ⊥
never = later (# never)

```

El tipo  $\_ \perp$  viene con dos formas de igualdad (bisemejanza débil y fuerte) y una relación de orden. La bisemejanza fuerte es más fuerte que el orden y, a su vez, este último es más fuerte que la bisemejanza débil. Las tres relaciones se definen utilizando un único tipo `data`, el cual estará indexado por un valor de tipo `Kind` que indica qué tipo de relación es. Este último se define de la siguiente manera:

```

data OtherKind : Set where
  geq weak : OtherKind

data Kind : Set where
  strong : Kind
  other : (k : OtherKind) → Kind

```

El constructor `strong` representa la bisemejanza fuerte, mientras que `other k` representa la relación de orden si  $k = \text{geq}$ , o la bisemejanza débil si  $k = \text{weak}$ . La igualdad entre tipos de igualdad, es decir entre valores del tipo `Kind`, es decidible. El operador  $\stackrel{?}{=}\text{-Kind}$  toma dos tipos de igualdad y decide si son iguales o no, dando a su vez una prueba de ello.

```

infix 4 _ $\stackrel{?}{=}\text{-Kind}$ _

_ $\stackrel{?}{=}\text{-Kind}$ _ : Decidable (_ $\equiv$ _ {A = Kind})
_ $\stackrel{?}{=}\text{-Kind}$ _ strong strong = yes P.refl
_ $\stackrel{?}{=}\text{-Kind}$ _ strong (other k) = no λ()
_ $\stackrel{?}{=}\text{-Kind}$ _ (other k) strong = no λ()
_ $\stackrel{?}{=}\text{-Kind}$ _ (other geq) (other geq) = yes P.refl
_ $\stackrel{?}{=}\text{-Kind}$ _ (other geq) (other weak) = no λ()
_ $\stackrel{?}{=}\text{-Kind}$ _ (other weak) (other geq) = no λ()
_ $\stackrel{?}{=}\text{-Kind}$ _ (other weak) (other weak) = yes P.refl

```

Como se puede ver, la definición de esta función es muy sencilla. Para los casos en que ambos tipos son el mismo la prueba es simplemente `refl`, y en los casos en que no lo son, la prueba es el patrón absurdo, puesto que no hay manera de dar una prueba de igualdad entre ellos. Esta función sirve para definir un predicado que indica si la relación es de igualdad o no. Este predicado será verdadero para `strong` y `other weak`, pero no para `other geq`. Será de utilidad tener este predicado a la hora de probar que las igualdades son relaciones de equivalencia.

```

Equality : Kind → Set
Equality k = False (k  $\stackrel{?}{=}\text{-Kind}$  other geq)

```

Una vez introducido el tipo `Kind`, se definen las relaciones propiamente dichas. Esto se realiza mediante la creación de un módulo llamado `Equality`, el cual toma como parámetros un conjunto  $A : \text{Set}$ , que será el tipo de retorno, y una relación  $\_ \sim \_ : A \rightarrow A \rightarrow \text{Set}$  que establece una noción de igualdad entre valores de tipo  $A$ .

```

module Equality {A : Set} (_ $\sim$ _ : A → A → Set) where

```

```

data Rel : Kind → A ⊥ → A ⊥ → Set where
  now  : ∀ {k x y} (x ~ y : x ~ y) → Rel k (now x) (now y)
  later : ∀ {k x y} (x ~ y : ∞ (Rel k (b x) (b y))) → Rel k (later x) (later y)
  laterr : ∀ {x y} (x ≈ y : Rel (other weak) x (b y)) → Rel (other weak) x (later y)
  laterl : ∀ {k x y} (x ~ y : Rel (other k) (b x) y) → Rel (other k) (later x) y

```

Las relaciones se definen a través del tipo `Rel`, el cual toma como parámetro un `Kind` que indica qué tipo de relación es y dos valores de tipo `A ⊥`, los cuales va a comparar mediante la relación correspondiente. El tipo `Rel` tiene cuatro constructores:

- El constructor `now` indica que, si se tiene una prueba de que dos valores  $x$  e  $y$  de tipo  $A$  están relacionados por la relación  $\sim$ , es decir que son iguales en  $A$ , entonces los términos `(now x)` y `(now y)` están relacionados en  $A ⊥$  para los tres tipos de relación posibles. Esto está dado por la utilización de la variable  $k$  que puede tomar cualquier valor de tipo `Kind` y quiere decir que `(now x)` y `(now y)` son bisemejantes tanto débil como fuertemente, y que además `(now x)` es mayor o igual a `(now y)`.
- El constructor `later` también sirve para las tres relaciones. En este caso, los  $x$  e  $y$  que recibe como parámetros implícitos son de tipo  $∞ A ⊥$  (valores de tipo  $A ⊥$  posiblemente infinitos). Este constructor pide una prueba (posiblemente infinita) de que `(b x)` y `(b y)` están relacionados por la relación  $k$  y afirma que entonces `(later x)` y `(later y)` están relacionados por la misma relación  $k$ .
- El constructor `laterr` se puede utilizar únicamente para bisemejanza débil. Dados  $x : A ⊥$  e  $y : ∞ A ⊥$  que son parámetros implícitos y una prueba de que  $x$  y `(b y)` son débilmente bisemejantes, este constructor afirma que entonces  $x$  y `(later y)` son débilmente bisemejantes también. Intuitivamente, `laterr` permite agregar un constructor `later` en el lado derecho de una bisemejanza débil.
- El constructor `laterl` es análogo a `laterr` e, intuitivamente, permite agregar un constructor `later` en el lado izquierdo de una relación que, en este caso, puede ser bisemejanza débil o desigualdad. Esto se evidencia en el uso de la variable  $k$  que tiene tipo `OtherKind` ya que luego en el tipo de relación que se le pasa a `Rel` se utiliza `(other k)`, quedando claro por qué al definir el tipo `Kind` se separa el tipo `strong` de los otros dos.

En el uso que permite cada constructor se puede ver que la relación de tipo `strong` es la más restrictiva, permitiendo únicamente agregar constructores `later` a ambos lados de la igualdad, mientras que la desigualdad `(other geq)` es un poco más débil ya que permite también agregar sólo a izquierda y la bisemejanza débil es la más débil de todas ya que permite agregar a ambos lados por separado.

Luego del tipo `Rel` se definen los operadores  $\cong$ ,  $\succsim$ ,  $\lesssim$  y  $\approx$ , los cuales representan la bisemejanza fuerte, la desigualdad (mayor o igual), la desigualdad invertida (menor o igual) y la bisemejanza débil, respectivamente.

```

infix 4 _≅_ _>= _<= _≈_
_≅_ : A ⊥ → A ⊥ → Set _
_≅_ = Rel strong

_>= _ : A ⊥ → A ⊥ → Set _
_>= _ = Rel (other geq)

_<= _ : A ⊥ → A ⊥ → Set _
_<= _ = flip _>= _

```

$\_ \approx \_ : A \perp \rightarrow A \perp \rightarrow \text{Set } \_$   
 $\_ \approx \_ = \text{Rel } (\text{other weak})$

A continuación, se definen otros operadores que tienen que ver con la noción de convergencia introducida en la definición 3.13.

$\text{infix } 4 \_ \Downarrow \_ \_ \_ \_ \_ \_$   
 $\_ \Downarrow \_ : A \perp \rightarrow \text{Kind} \rightarrow A \rightarrow \text{Set } \_$   
 $x \Downarrow [k] y = \text{Rel } k x (\text{now } y)$   
 $\_ \Downarrow \_ : A \perp \rightarrow A \rightarrow \text{Set } \_$   
 $x \Downarrow y = x \Downarrow [\text{other weak}] y$   
 $\text{infix } 4 \_ \Downarrow$   
 $\_ \Downarrow : A \perp \rightarrow \text{Set } \_$   
 $x \Downarrow = \exists \lambda v \rightarrow x \Downarrow v$   
 $\text{infix } 4 \_ \Uparrow \_ \_ \_ \_ \_ \_$   
 $\_ \Uparrow \_ : A \perp \rightarrow \text{Kind} \rightarrow \text{Set } \_$   
 $x \Uparrow [k] = \text{Rel } k x \text{ never}$   
 $\_ \Uparrow : A \perp \rightarrow \text{Set } \_$   
 $x \Uparrow = x \Uparrow [\text{other weak}]$

$x \Downarrow [k] y$  indica que  $x$  se relaciona mediante la relación  $k$  con  $(\text{now } y)$ . En el caso especial de que la relación sea bisemejanza débil, se escribe  $x \Downarrow y$  y se dice que  $x$  converge al valor  $y : A$ . Por otro lado,  $x \Downarrow$  indica que la computación  $x$  termina, es decir que existe algún valor  $v$  tal que  $x \Downarrow v$ . Por último,  $x \Uparrow [k]$  indica que  $x$  se relaciona con el valor especial **never** mediante la relación  $k$ . En el caso de que esa relación sea  $(\text{other weak})$  se escribe  $x \Uparrow$  y se dice que la computación  $x$  no termina.

Luego de dar todas las definiciones, se prueba que las tres relaciones son reflexivas y transitivas y que las bisemejanzas son simétricas mientras que la desigualdad es antisimétrica. Antes de poder demostrar esto, se prueba un conjunto de lemas que serán de utilidad para tal propósito. Todas estas pruebas se sitúan dentro de un módulo sin nombre parametrizado con el conjunto  $A$  de retorno y su relación de igualdad. Esto se hace para fijar tal conjunto y su relación y poder utilizarlos a lo largo de todas las demostraciones.

$\text{module } \_ \{A : \text{Set}\} \{ \_ \sim \_ : A \rightarrow A \rightarrow \text{Set} \} \text{ where}$   
 $\text{open Equality } \_ \sim \_ \text{ using (Rel; } \_ \cong \_ ; \_ \gtrsim \_ ; \_ \lesssim \_ ; \_ \approx \_ ; \_ \Downarrow \_ \_ ; \_ \Uparrow \_ \_ )$   
 $\text{open Equality.Rel}$   
 $\cong \Rightarrow : \forall \{k\} \{x y : A \perp\} \rightarrow x \cong y \rightarrow \text{Rel } k x y$   
 $\cong \Rightarrow (\text{now } x \sim y) = \text{now } x \sim y$   
 $\cong \Rightarrow (\text{later } x \cong y) = \text{later } (\# \cong \Rightarrow (\flat x \cong y))$   
 $\gtrsim \Rightarrow : \forall \{k\} \{x y : A \perp\} \rightarrow x \gtrsim y \rightarrow \text{Rel } (\text{other } k) x y$   
 $\gtrsim \Rightarrow (\text{now } x \sim y) = \text{now } x \sim y$



$$\begin{aligned}
\gtrRightarrow (\text{later } x \gtrsim y) &= \text{later } (\# \gtrRightarrow (\flat x \gtrsim y)) \\
\gtrRightarrow (\text{later}^l x \gtrsim y) &= \text{later}^l (\gtrRightarrow x \gtrsim y) \\
\Rightarrow \approx : \forall \{k\} \{x y : A \perp\} &\rightarrow \text{Rel } k \ x \ y \rightarrow x \approx y \\
\Rightarrow \approx \{\text{strong}\} &= \cong \Rightarrow \\
\Rightarrow \approx \{\text{other geq}\} &= \gtrRightarrow \\
\Rightarrow \approx \{\text{other weak}\} &= \text{id}
\end{aligned}$$

El lema  $\cong \Rightarrow$  demuestra que todas las relaciones incluyen a la bisemejanza fuerte, es decir que si dos términos  $x y : A \perp$  son fuertemente bisemejantes, entonces también son débilmente bisemejantes y, además,  $x \gtrsim y$ . De manera similar, el segundo lema,  $\gtrRightarrow$ , postula que la desigualdad está incluida en la bisemejanza débil. Si  $x \gtrsim y$ , entonces  $x$  e  $y$  son débilmente bisemejantes. Por último, el lema  $\Rightarrow \approx$  indica que todas las relaciones están incluidas en la bisemejanza débil. Si  $x$  e  $y$  se relacionan mediante cualquier tipo de relación  $k$ , entonces también se relacionan por la bisemejanza débil.

Las definiciones que siguen representan las operaciones inversas de los constructores  $\text{later}^r$ ,  $\text{later}^l$  y  $\text{later}$ . Así como estos constructores permiten “agregar demoras” a uno y otro lado de las ecuaciones, sus operaciones inversas permiten “quitar demoras” para ciertos tipos de relaciones.

$$\begin{aligned}
\text{later}^{r-1} : \forall \{k\} \{x : A \perp\} \{y\} &\rightarrow \\
&\text{Rel } (\text{other } k) \ x \ (\text{later } y) \rightarrow \text{Rel } (\text{other } k) \ x \ (\flat y) \\
\text{later}^{r-1} (\text{later } x \sim y) &= \text{later}^l (\flat x \sim y) \\
\text{later}^{r-1} (\text{later}^r x \approx y) &= x \approx y \\
\text{later}^{r-1} (\text{later}^l x \sim ly) &= \text{later}^l (\text{later}^{r-1} x \sim ly) \\
\text{later}^{l-1} : \forall \{x\} \{y : A \perp\} &\rightarrow \text{later } x \approx y \rightarrow \flat x \approx y \\
\text{later}^{l-1} (\text{later } x \approx y) &= \text{later}^r (\flat x \approx y) \\
\text{later}^{l-1} (\text{later}^r lx \approx y) &= \text{later}^r (\text{later}^{l-1} lx \approx y) \\
\text{later}^{l-1} (\text{later}^l x \approx y) &= x \approx y \\
\text{later}^{-1} : \forall \{k\} \{x y : \infty (A \perp)\} &\rightarrow \\
&\text{Rel } k \ (\text{later } x) \ (\text{later } y) \rightarrow \text{Rel } k \ (\flat x) \ (\flat y) \\
\text{later}^{-1} (\text{later } x \sim y) &= \flat x \sim y \\
\text{later}^{-1} (\text{later}^r lx \approx y) &= \text{later}^{l-1} lx \approx y \\
\text{later}^{-1} (\text{later}^l x \sim ly) &= \text{later}^{r-1} x \sim ly
\end{aligned}$$

La función  $\text{later}^{r-1}$  permite sacar un constructor  $\text{later}$  del lado derecho de una ecuación que tenga como operador a la desigualdad o la bisemejanza débil. Si se tiene  $\text{Rel } (\text{other } k) \ x \ (\text{later } y)$ , para algún tipo  $k : \text{OtherKind}$  y algún par de términos  $x : A \perp$  e  $y : \infty (A \perp)$ , entonces se tiene que  $x$  se relaciona por el mismo tipo de relación  $k$  con  $(\flat y)$ . La función  $\text{later}^{l-1}$ , de manera análoga, permite quitar un constructor  $\text{later}$  del lado izquierdo de la ecuación, pero sólo para el caso de la bisemejanza débil. Esto implica que si se tiene  $\text{later } x \approx y$ , entonces también se cumple que  $\flat x \approx y$ . Por último, la función  $\text{later}^{-1}$  permite quitar el constructor  $\text{later}$  de ambos lados de la ecuación. Esto puede realizarse para cualquier tipo de relación  $k : \text{Kind}$ .

Teniendo en cuenta todos estos lemas se pueden probar finalmente las propiedades que cumplen las relaciones definidas. Para poder realizar estas demostraciones se asumen las propiedades análogas para la relación subyacente  $\_ \sim \_$  sobre el conjunto de retorno  $A$ . Se realizan dichas pruebas dentro de un módulo llamado **Equivalence** que las encapsula para luego importarlas todas juntas cuando sean necesarias.

module Equivalence where

```
refl : Reflexive _ ~ _ → ∀ {k} → Reflexive (Rel k)
refl refl~ {x = now v} = now refl~
refl refl~ {x = later x} = later (# refl refl~)
```

La reflexividad se prueba para cualquier tipo de relación  $k : \text{Kind}$ . Se pide como condición una prueba de que la relación subyacente  $\_ \sim \_$  es también reflexiva. Tanto la prueba argumento como la prueba que se retorna están dadas por un término de tipo `Reflexive` definido en el módulo `Relation.Binary.Definitions`. Este es una función que toma una relación binaria  $\_ \sim \_$  y devuelve el tipo correspondiente a la prueba de que dicha relación es reflexiva, es decir  $\forall \{x\} \rightarrow x \sim x$ . De manera análoga, la prueba de simetría se da mediante un término de tipo `Symmetric` que, dada una relación, devuelve el tipo de la prueba que indica que tal relación es simétrica.

```
sym : Symmetric _ ~ _ → ∀ {k} → Equality k → Symmetric (Rel k)
sym sym~ eq (now x~y) = now (sym~ x~y)
sym sym~ eq (later x~y) = later (# sym sym~ eq (b x~y))
sym sym~ eq (laterr x~y) = laterl (sym sym~ eq x~y)
sym sym~ eq (laterl {weak} x~y) = laterr (sym sym~ eq x~y)
```

La prueba de simetría requiere, además de la prueba de que la relación subyacente  $\_ \sim \_$  es simétrica, una prueba de que el tipo de relación  $k : \text{Kind}$  es una igualdad. Esta se da mediante el predicado `Equality` definido más arriba, que es verdadero para ambas bisemejanzas y falso para la desigualdad. Puede parecer a simple vista que esta prueba no es utilizada para demostrar la simetría. Sin embargo, es la presencia de la misma la que hace que no se requiera analizar el caso  $(\text{later}^l \{ \text{geq} \} x \gtrsim y)$  y la prueba aún así sea exhaustiva. Para verlo más claramente se puede agregar este caso, donde se reemplaza la prueba `eq` por el patrón absurdo, haciendo que no sea necesario dar la definición correspondiente:

```
sym sym~ () (laterl {geq} x~y)
```

La transitividad se prueba, al igual que la reflexividad, para todos los tipos de relación  $k$ . Esta demostración es un poco más compleja y requiere de tres pruebas separadas que luego se unen. Es por esta razón que se crea un módulo privado `Trans` que contenga todas las partes de la prueba dentro, donde luego sólo se exporta públicamente la función `trans` que constituye la prueba final. Este módulo toma como parámetro una prueba  $\text{trans} \sim$  de tipo `Transitive _ ~ _` que postula la transitividad de la relación subyacente.

private

module Trans (trans~ : Transitive \_ ~ \_) where

```
now-trans : ∀ {k x y} {v : A} →
  Rel k x y → Rel k y (now v) → Rel k x (now v)
now-trans (now x~y) (now y~z) = now (trans~ x~y y~z)
now-trans (laterl x~y) y~z = laterl (now-trans x~y y~z)
now-trans x~ly (laterl y~z) = now-trans (laterr-1 x~ly) y~z
```

mutual

```
later-trans : ∀ {k} {x y : A ⊥} {z} →
```

$$\begin{aligned}
& \text{Rel } k \ x \ y \rightarrow \text{Rel } k \ y \ (\text{later } z) \rightarrow \text{Rel } k \ x \ (\text{later } z) \\
& \text{later-trans } (\text{later } x \sim y) \ ly \sim lz = \text{later } (\# \text{trans } (\flat x \sim y) (\text{later}^{-1} ly \sim lz)) \\
& \text{later-trans } (\text{later}^l x \sim y) \ y \sim lz = \text{later } (\# \text{trans } x \sim y (\text{later}^{r-1} y \sim lz)) \\
& \text{later-trans } (\text{later}^r x \sim y) \ ly \sim lz = \text{later-trans } x \sim y (\text{later}^{l-1} ly \sim lz) \\
& \text{later-trans } x \approx y \quad (\text{later}^r y \approx z) = \text{later}^r (\text{trans } x \approx y \quad y \approx z) \\
& \text{trans} : \forall \{k\} \{x \ y \ z : A \perp\} \rightarrow \text{Rel } k \ x \ y \rightarrow \text{Rel } k \ y \ z \rightarrow \text{Rel } k \ x \ z \\
& \text{trans } \{z = \text{now } v\} \ x \sim y \ y \sim v = \text{now-trans } x \sim y \ y \sim v \\
& \text{trans } \{z = \text{later } z\} \ x \sim y \ y \sim lz = \text{later-trans } x \sim y \ y \sim lz
\end{aligned}$$

open Trans public using (trans)

La transitividad postula que, dados tres términos  $x$ ,  $y$  y  $z$ , si se tienen las relaciones  $\text{Rel } k \ x \ y$  y  $\text{Rel } k \ y \ z$ , entonces se debe tener también la relación  $\text{Rel } k \ x \ z$ . Para demostrar esto, se necesitan dos pruebas auxiliares: **now-trans** que prueba esto asumiendo  $z = (\text{now } v)$  para algún  $v : A$  y **later-trans** que hace lo mismo para el caso en que  $z = (\text{later } z)$ . Esta última y la prueba principal **trans** son mutuamente recursivas y, por lo tanto, se sitúan dentro de un bloque **mutual**.

Se introduce por último la prueba de antisimetría de la relación de desigualdad. Esta postula que, dados dos términos  $x \ y : A \perp$ , si se tiene que  $x \gtrsim y$  y  $x \lesssim y$ , luego  $x \cong y$ , es decir que  $x$  e  $y$  son fuertemente bisemejantes.

$$\begin{aligned}
& \text{antisym} : \{x \ y : A \perp\} \rightarrow x \gtrsim y \rightarrow x \lesssim y \rightarrow x \cong y \\
& \text{antisym } (\text{now } x \sim y) \ (\text{now } \_) = \text{now } x \sim y \\
& \text{antisym } (\text{later } x \gtrsim y) \ (\text{later } x \lesssim y) = \text{later } (\# \text{antisym } (\flat x \gtrsim y) (\flat x \lesssim y)) \\
& \text{antisym } (\text{later } x \gtrsim y) \ (\text{later}^l x \lesssim ly) = \text{later } (\# \text{antisym } (\flat x \gtrsim y) (\text{later}^{r-1} x \lesssim ly)) \\
& \text{antisym } (\text{later}^l x \gtrsim ly) \ (\text{later } x \lesssim y) = \text{later } (\# \text{antisym } (\text{later}^{r-1} x \gtrsim ly) (\flat x \lesssim y)) \\
& \text{antisym } (\text{later}^l x \gtrsim ly) \ (\text{later}^l x \lesssim ly) = \text{later } (\# \text{antisym } (\text{later}^{r-1} x \gtrsim ly) (\text{later}^{r-1} x \lesssim ly))
\end{aligned}$$

Podría darse también la prueba de antisimetría respecto de la bisemejanza débil, pero esta es trivial puesto que, como se indica en el lema  $\gtrsim \Rightarrow$ , basta con sólo una de las relaciones de desigualdad para que  $x$  e  $y$  sean débilmente bisemejantes. Las pruebas de antisimetría en general tienen sentido cuando la igualdad está incluida en la relación de orden, y no al revés.

## 5.2. Prueba de que el tipo *delay* es una mónada

Ya introducido el tipo *delay* en Agda, se procede ahora a dar la prueba de que este tiene estructura de mónada. Esto se realizará mediante la creación de una instancia del **record Monad** definido en la sección 4.2.1, cuyo parámetro será el funtor  $\_ \perp$ . La mónada *delay* puede definirse tanto para la bisemejanza fuerte como para la bisemejanza débil. Se detalla a continuación sólo una de estas pruebas puesto que la otra es análoga.

Lo primero a definir es el operador *bind*, el cual es común a ambas pruebas y se define como sigue:

$$\begin{aligned}
& \text{bind} : A \perp \rightarrow (A \rightarrow B \perp) \rightarrow B \perp \\
& \text{bind } (\text{now } x) f = f x \\
& \text{bind } (\text{later } x) f = \text{later } (\# (\text{bind } (\flat x) f))
\end{aligned}$$

El operador **bind** debe tomar un elemento de tipo  $MA$  y una función  $A \rightarrow MB$ . Como en este caso el funtor  $M$  está dado por el operador de tipos  $\_ \perp$ , los argumentos de **bind** serán un

término  $t$  de tipo  $A \perp$  y una función  $f$  de tipo  $A \rightarrow B \perp$ . En caso de que el primer argumento esté formado por el un termino (**now**  $x$ ), el resultado será simplemente aplicar la función  $f$  a  $x$ . En caso de que el primer argumento sea algo del estilo (**later**  $x$ ), el resultado de aplicar **bind** será “meter” la aplicación hacia adentro del constructor, de manera que la definición es productiva.

La función **return** estará representada por el constructor **now**, por lo que no es necesario dar una definición de la misma. Este constructor tiene el comportamiento esperado del operador **return**, cuyo sentido usual es encapsular un valor del tipo de retorno dentro de la mónada.

Una vez definidos los operadores básicos, comienza la prueba propiamente dicha. Se analizará la prueba para la bisemejanza débil, la cual se llevará a cabo dentro de un módulo parametrizado llamado **Weak** que tendrá dos parámetros: una relación binaria  $\sim : \forall \{A\} \rightarrow A \rightarrow A \rightarrow \text{Set}$  y una prueba de que dicha relación es una relación de equivalencia. El objetivo de estos parámetros es fijar una relación de igualdad para los tipos de retorno.

```
module Weak ( _ ~ _ :  $\forall \{A\} \rightarrow A \rightarrow A \rightarrow \text{Set}$ )
  ( eq $\sim$  :  $\forall \{A\} \rightarrow \text{IsEquivalence } (\_ \sim \_ \{A\})$ ) where
```

Se demostrarán a continuación las tres leyes de las mónadas. Estas tres pruebas estarán encapsuladas dentro de otro módulo, en este caso anónimo, en el cual se fijará un tipo de retorno particular  $A$  con una relación binaria de igualdad sobre el mismo y una prueba de que dicha relación es reflexiva. Este módulo interno sirve principalmente para agrupar los argumentos comunes que tendrán las demostraciones de las tres leyes dentro de él.

```
module _ {A : Set} { _ ~ _ : A  $\rightarrow$  A  $\rightarrow$  Set} ( refl $\sim$  : Reflexive _ ~ _ ) where

  open Equality _ ~ _ using ( _  $\approx$  _ )
  open Equality.Rel
  open Equivalence using ( refl )
```

Antes de comenzar a demostrar se abren ciertos módulos del tipo *delay* que serán utilizados en las pruebas. Se abre primero el módulo **Equality**, pasándole la relación  $\sim$  sobre  $A$  como argumento y extrayendo del mismo la bisemejanza débil ( $\approx$ ). Luego se abre el tipo **data Rel** para tener acceso a sus constructores. Finalmente, se abre el módulo **Equivalence** extrayendo de él la prueba **refl** de que todas las relaciones (y en particular la bisemejanza débil) son reflexivas. Con todos estos ingredientes la prueba de la primera ley de mónadas queda como sigue:

```
left-identity : ( x : B ) ( f : B  $\rightarrow$  A  $\perp$  )  $\rightarrow$  bind ( now x ) f  $\approx$  f x
left-identity x f = refl refl $\sim$ 
```

El tipo de la prueba indica que se demuestra que la función **return**, en este caso el constructor **now**, es neutro a izquierda de **bind**. La demostración es trivial ya que la definición de la función **bind** establece la igualdad postulada. Seguidamente se expone la demostración de la segunda ley de mónadas:

```
right-identity : ( t : A  $\perp$  )  $\rightarrow$  bind t now  $\approx$  t
right-identity ( now x ) = refl refl $\sim$ 
right-identity ( later x ) = later (  $\sharp$  ( right-identity (  $\flat$  x ) ) )
```

En este caso se prueba que el constructor **now** es neutro a derecha de **bind**, es decir que para un término  $t : A \perp$ , **bind**  $t$  **now**  $\approx t$ . En el caso de que  $t$  sea de la forma (**now**  $x$ ), la definición

de `bind` indica que el resultado del lado izquierdo será la aplicación de `now` a  $x$ , quedando una prueba trivial. En caso de tener algo de la forma `(later x)`, así como la definición de `bind` es co-recursiva, también lo será la prueba.

Queda por último la prueba de la tercera ley de mónadas: la asociatividad de `bind`. Al igual que en la segunda, el caso base es trivial, mientras que el otro es simplemente co-recursivo.

```

associative : (x : C ⊥) (f : C → B ⊥) (g : B → A ⊥)
  → bind (bind x f) g ≈ bind x (λ y → bind (f y) g)
associative (now x) f g = refl refl~
associative (later x) f g = later (# associative (λ x) f g)

```

Se cierra entonces el módulo anónimo abierto más arriba y se pasa a definir los elementos faltantes para la prueba. La relación de igualdad será, para cualquier conjunto  $A$ , la relación de bisemejanza débil `_≈_` definida en el módulo `Equality` con la relación subyacente `_~_`.

```

_≈_⊥_ : ∀ {A} → A ⊥ → A ⊥ → Set
_≈_⊥_ {A} = Equality._≈_ {A} (_~_ {A})

```

La prueba de que la relación establecida es de equivalencia se construye utilizando las demostraciones de reflexividad, simetría y transitividad presentes en el módulo `Equivalence`, las cuales requieren la propiedad análoga para la relación subyacente. Las pruebas de estas propiedades análogas se extraen de la prueba `eq~`.

```

open Equivalence using (refl; sym; trans)

eq≈⊥ : ∀ {A} → IsEquivalence (_≈_⊥_ {A})
eq≈⊥ = record
  { refl = refl (IsEquivalence.refl eq~) ;
    sym = sym (IsEquivalence.sym eq~) tt ;
    trans = trans (IsEquivalence.trans eq~) }

```

Finalmente, habiendo definido todos los componentes necesarios, la instancia de `Monad` para el tipo `_⊥` queda como sigue:

```

open import Structures.Monad

delayMonad : Monad _⊥
delayMonad = makeMonad
  _≈_⊥_
  eq≈⊥
  now
  bind
  (left-identity (IsEquivalence.refl eq~))
  (right-identity (IsEquivalence.refl eq~))
  (associative (IsEquivalence.refl eq~))

```

Se puede observar que a las pruebas de las tres leyes se les pasa como argumento el parámetro requerido por el módulo anónimo en el cual están encapsuladas.

La versión de la prueba para la bisemejanza fuerte es análoga a la primera, sólo cambiando la relación de igualdad que se utiliza. El código de la misma se encuentra en el [Apéndice](#) de este capítulo.

### 5.3. Prueba de que el tipo *delay* es un funtor monoidal

En esta sección se dotará al tipo *delay* de la estructura de funtor monoidal. El objetivo es acercarse de a poco a la estructura de mónada concurrente, teniendo ya la estructura monádica, el siguiente paso es la estructura monoidal. El primer elemento a definir es entonces la operación binaria que, para el funtor  $M$ , debe tener tipo  $MA \rightarrow MB \rightarrow M(A \times B)$ . En este caso particular, como el funtor  $M$  está dado por el operador de tipos  $\_ \perp$ , la operación **merge** tendrá el siguiente tipo:  $A \perp \rightarrow B \perp \rightarrow (A \times B) \perp$ .

```
merge : A  $\perp$   $\rightarrow$  B  $\perp$   $\rightarrow$  (A  $\times$  B)  $\perp$ 
merge (now a) (now b) = now (a , b)
merge (now a) (later b) = later ( $\#$  (merge (now a) ( $\flat$  b)))
merge (later a) (now b) = later ( $\#$  (merge ( $\flat$  a) (now b)))
merge (later a) (later b) = later ( $\#$  (merge ( $\flat$  a) ( $\flat$  b)))
```

La función **merge** se define mediante *pattern matching* sobre los dos argumentos. En el caso de que ambos sean términos de la forma **(now x)**, el resultado también será un término de la misma forma, donde el valor de retorno será el par formado por los valores de retorno de los argumentos. Cuando uno de los dos es de la forma **(now x)** y el otro es de la forma **(later y)**, el resultado se construye con un constructor **later** que dentro tiene una llamada co-recursiva suspendida. El término que tenía el constructor **later** ya no lo tiene, haciendo productiva la co-recursión, mientras que el que tenía el constructor **now** queda igual. En caso de que ambos términos estén formados por un constructor **later**, también se realiza una llamada co-recursiva, donde ambos términos se achican, y por fuera sólo queda un único constructor **later** en lugar de dos. Esto se debe a que **merge**, intuitivamente, representa el paralelismo de dos computaciones, por lo que ambos **later** se fusionan en uno que realiza internamente el resto de ambas computaciones en paralelo (gracias a la llamada co-recursiva de **merge**).

El siguiente operador a definir es **fmap**, el cual representa, como se mencionó en la sección 4.2.2, la asignación de morfismos del funtor que en este caso está dado por  $\_ \perp$ . Este operador, por lo tanto, asignará, para cada función  $A \rightarrow B$ , una función  $A \perp \rightarrow B \perp$ .

```
fmap :  $\forall \{A \ B : \text{Set}\} \rightarrow (A \rightarrow B) \rightarrow (A \perp \rightarrow B \perp)$ 
fmap f (now x) = now (f x)
fmap f (later x) = later ( $\#$  (fmap f ( $\flat$  x)))
```

Dada la función  $f : A \rightarrow B$ , si el elemento de tipo  $A \perp$  recibido es de la forma **(now x)**, entonces el resultado de aplicar **fmap f** a tal elemento es un término construido también con **now**, donde el valor de retorno será la aplicación de  $f$  al valor de retorno del argumento. En caso de que el argumento tenga la forma **(later x)**, el resultado de la aplicación de **fmap f** a tal término estará formado también por un constructor **later** que dentro propaga la aplicación de **fmap f** al término **( $\flat$  x)**.

Queda por definir el operador 0-ario **unit**. Este se definió en la formalización como un elemento de tipo  $M \top$ . En el caso del tipo *delay*, será un habitante del tipo  $\top \perp$ . Se define entonces **unit** como el elemento más simple de tal tipo:

```
unit :  $\top \perp$ 
unit = now tt
```

Definidos estos tres operadores, lo que sigue son las demostraciones de las propiedades. La prueba de que el tipo *delay* es un funtor monoidal puede darse, al igual que la de mónada,

tanto para la bisemejanza fuerte como para la débil, sinendo ambas pruebas análogas. Se detalla entonces la versión de la bisemejanza débil. Esta prueba se dará dentro de un módulo llamado **Weak** que recibirá como parámetro la relación de igualdad subyacente junto con una prueba de que es una relación de equivalencia.

```
module Weak ( _ ~ _ :  $\forall \{A\} \rightarrow A \rightarrow A \rightarrow \text{Set}$  )
  ( eq $\sim$  :  $\forall \{A\} \rightarrow \text{IsEquivalence } (\_ \sim \_ \{A\})$  ) where
```

La primera propiedad que se demostrará será la asociatividad del operador **merge**. Esta se da dentro de un módulo anónimo en el cual se fija como tipo de retorno el producto cartesiano de tres conjuntos  $A$ ,  $B$  y  $C$ . Este módulo requiere como parámetro una relación de igualdad entre elementos de este conjunto y la prueba de que la misma es reflexiva. Dentro del módulo se abre, primero, el módulo **Equality** para el conjunto de retorno  $A \times B \times C$  con la igualdad subyacente recibida como argumento, del cual se extrae la relación de bisemejanza débil  $\_ \approx \_$ . Luego se abre el tipo **Rel** de manera que puedan usarse sus constructores sin prefijo.

```
module _ { A B C : Set } { _ ~ _ : A  $\times$  B  $\times$  C  $\rightarrow$  A  $\times$  B  $\times$  C  $\rightarrow$  Set }
  ( reflABC : Reflexive _ ~ _ ) where

  open Equality { A  $\times$  B  $\times$  C } _ ~ _ using ( _  $\approx$  _ )
  open Equality.Rel

  associative : ( a : A  $\perp$  ) ( b : B  $\perp$  ) ( c : C  $\perp$  )
     $\rightarrow$  ( fmap (  $\lambda \{ ((a, b), c) \rightarrow (a, (b, c)) \}$  ) ( merge ( merge a b ) c ) )
       $\approx$  ( merge a ( merge b c ) )

  associative ( now a ) ( now b ) ( now c ) = now reflABC
  associative ( now a ) ( now b ) ( later c ) = later (  $\#$  ( associative ( now a ) ( now b ) (  $\flat$  c ) ) )
  associative ( now a ) ( later b ) ( now c ) = later (  $\#$  ( associative ( now a ) (  $\flat$  b ) ( now c ) ) )
  associative ( now a ) ( later b ) ( later c ) = later (  $\#$  ( associative ( now a ) (  $\flat$  b ) (  $\flat$  c ) ) )
  associative ( later a ) ( now b ) ( now c ) = later (  $\#$  ( associative (  $\flat$  a ) ( now b ) ( now c ) ) )
  associative ( later a ) ( now b ) ( later c ) = later (  $\#$  ( associative (  $\flat$  a ) ( now b ) (  $\flat$  c ) ) )
  associative ( later a ) ( later b ) ( now c ) = later (  $\#$  ( associative (  $\flat$  a ) (  $\flat$  b ) ( now c ) ) )
  associative ( later a ) ( later b ) ( later c ) = later (  $\#$  ( associative (  $\flat$  a ) (  $\flat$  b ) (  $\flat$  c ) ) )
```

La prueba no presenta mayores dificultades. En el caso base, donde los tres términos son de la forma  $(\text{now } x)$ , las aplicación  $(\text{merge } (\text{merge } (\text{now } a) (\text{now } b)) (\text{now } c))$  genera el siguiente término:  $(\text{now } ((a, b), c))$ . Al aplicar  $\text{fmap } (\lambda \{ ((a, b), c) \rightarrow (a, (b, c)) \})$  a este resultado, se obtiene el término  $(\text{now } (a, (b, c)))$ , el cual es también el resultado de la aplicación del lado derecho de la ecuación:  $(\text{merge } (\text{now } a) (\text{merge } (\text{now } b) (\text{now } c)))$ , por lo que la prueba queda trivial. En los demás casos, la prueba se resuelve con una llamada co-recursiva, donde los términos de la forma  $(\text{now } x)$  quedan igual, mientras que los que se componen de un constructor **later** se achican.

Para la prueba del neutro a derecha del operador **merge**, se define otro módulo anónimo en el que la relación de igualdad pedida como parámetro y su respectiva prueba de que la relación dada es reflexiva deben estar definidas para elementos del conjunto  $A \times \top$ , para algún conjunto  $A$ . Dentro de este módulo se abren también el módulo **Equality** y el tipo **Rel** para el mismo conjunto.

```
module _ { A : Set } { _ ~ _ : (A  $\times$   $\top$ )  $\rightarrow$  (A  $\times$   $\top$ )  $\rightarrow$  Set }
  ( reflA $\times$  $\top$  : Reflexive _ ~ _ ) where
```

```

open Equality {A × T} _ ~ _ using (_ ≈ _)
open Equality.Rel

rid : (a : A ⊥) → (merge a unit) ≈ (fmap (λ a → (a , tt)) a)
rid (now x) = now reflA × T
rid (later x) = later (λ (rid (b x)))

```

Para el caso en el que  $a$  tiene la forma  $(\text{now } x)$ , la aplicación  $(\text{merge } (\text{now } x) \text{ unit})$  tiene como resultado el término  $(\text{now } (x , \text{tt}))$ . Aplicar  $\text{fmap } (\lambda a \rightarrow (a , \text{tt}))$  a  $(\text{now } x)$  tiene el mismo resultado, por lo que la prueba de este caso es trivial. El otro caso se resuelve, al igual, que antes con una llamada co-recursiva.

La prueba del neutro a izquierda de  $\text{merge}$  es análoga a la anterior, donde el módulo anónimo en el cual se encapsula es análogo también:

```

module _ {A : Set} { _ ~ _ : (T × A) → (T × A) → Set}
  (reflT × A : Reflexive _ ~ _) where

  open Equality {T × A} _ ~ _ using (_ ≈ _)
  open Equality.Rel

  lid : (a : A ⊥) → (merge unit a) ≈ (fmap (λ a → (tt , a)) a)
  lid (now x) = now reflT × A
  lid (later x) = later (λ (lid (b x)))

```

Los elementos faltantes son la relación de igualdad para términos de tipo  $A \perp$ , para cualquier conjunto  $A$ , y la prueba correspondiente de que tal relación es de equivalencia. La definición de estos elementos es igual a la construida para la estructura de mónada.

```

_ ≈ ⊥ _ : ∀ {A} → A ⊥ → A ⊥ → Set
_ ≈ ⊥ _ {A} = Equality._ ≈ _ {A} (_ ~ _ {A})

open Equivalence using (refl; sym; trans)

eq ≈ ⊥ : ∀ {A} → IsEquivalence (_ ≈ ⊥ _ {A})
eq ≈ ⊥ = record
  { refl = refl (IsEquivalence.refl eq ≈) ;
    sym = sym (IsEquivalence.sym eq ≈) tt ;
    trans = trans (IsEquivalence.trans eq ≈) }

```

Finalmente, la instancia de `MonoidalFunctor` para el operador de tipos  $\_ \perp$  queda como sigue:

```

open import Structures.MonoidalFunctor hiding (unit; merge; fmap)

delayMonoidal : MonoidalFunctor _ ⊥
delayMonoidal = makeMonoidalFunctor
  _ ≈ ⊥ _
  eq ≈ ⊥
  unit
  merge
  fmap

```



```
(rid (IsEquivalence.refl eq~))
(lid (IsEquivalence.refl eq~))
(associative (IsEquivalence.refl eq~))
```

La estructura de funtor monoidal para la bisemejanza fuerte es análoga a la anterior. El código de la misma se encuentra en el [Apéndice](#) de este capítulo.

## 5.4. ¿El tipo *delay* es una mónada concurrente?

El objetivo de esta sección es (intentar) demostrar que el tipo *delay* es una mónada concurrente. Como se detallará más adelante, la mayor parte de esta prueba es similar a las dos presentadas anteriormente. La dificultad está principalmente en el último campo: la ley de intercambio.

Una diferencia sustancial con las pruebas anteriores es que, en este caso, no es posible dar la instancia de `ConcurrentMonad` para el tipo *delay* utilizando como igualdad la bisemejanza débil. Esto se debe a que la relación de orden debe ser, entre otras cosas, reflexiva respecto de la igualdad. Es decir que si se tienen dos valores  $x$  e  $y$  tales que  $x \sim y$  (son iguales según la relación de igualdad considerada), entonces se debe cumplir que estos valores también están relacionados por la relación de orden, o sea que  $x \lesssim y$ . Esto no es cierto para la desigualdad definida para el tipo `_⊥` respecto de la bisemejanza débil ya que, como se mencionó en la sección 5.1, la bisemejanza débil es la relación más flexible de las tres que se introdujeron. Por lo tanto, la prueba se realizará únicamente para la bisemejanza fuerte.

Otro cambio respecto de las pruebas ya presentadas es que no se considerará en esta una igualdad parametrizada para los tipos de retorno como se hizo anteriormente. Esta decisión se debe a que en varias demostraciones, principalmente las que tienen que ver con la relación de orden, fue necesario realizar *pattern matching* sobre la igualdad de dos valores de retorno, de manera que se unifiquen dos variables si estas son iguales. Esto sólo es posible *a priori* para la igualdad proposicional, por lo que se desarrolló la prueba en cuestión tomando dicha relación como igualdad en los tipos de retorno. Si bien se pierde algo de generalidad, lo más importante es que se conserva la igualdad parametrizada para el tipo *delay* que es coinductivo, ya que esta clase de tipos es la que tiene más problemas con la igualdad proposicional.

Se importa entonces el módulo de la igualdad proposicional, renombrando `refl`, `sym` y `trans` agregando una `p` delante, de manera que no haya conflicto de nombres con las propiedades demostradas en el módulo `Equality` que llevan los mismos nombres.

```
open import Relation.Binary.PropositionalEquality
renaming (refl to prefl; sym to psym; trans to ptrans)
```

Los operadores que se utilizarán son los mismos que se definieron en las secciones anteriores: `bind` y `return` para la estructura monádica y `merge` y `unit = now tt` para la estructura monoidal.

Al igual que antes, se desarrolla la prueba dentro de un módulo llamado, en este caso, `Strong`, el cual no estará parametrizado ya que, como se explicó, se utilizará la igualdad proposicional para los tipos de retorno.

```
module Strong where
```

Las primeras propiedades que se demuestran son las leyes de las mónadas. Estas se encapsulan dentro de un módulo anónimo que está parametrizado por un conjunto  $A$  : `Set` de retorno. Antes

de realizar las pruebas se abre el módulo `Equality` para el conjunto  $A$  tomando como igualdad en este conjunto la igualdad proposicional ( $\_ \equiv \_$ ) y extrayendo del mismo la relación de bisemejanza fuerte. Luego se abre `Equality.Rel` para poder utilizar sus constructores y, finalmente, se abre el módulo `Equivalence` para utilizar la reflexividad de la bisemejanza fuerte. Las demostraciones son análogas a las realizadas en la sección 5.2, cambiando únicamente en los casos triviales la reflexividad utilizada.

```
module _ {A : Set} where

open Equality {A} _≡_ using (_≅_)
open Equality.Rel
open Equivalence

left-identity : (x : B) (f : B → A ⊥) → bind (now x) f ≅ f x
left-identity x f = refl prefl

right-identity : (t : A ⊥) → bind t now ≅ t
right-identity (now x) = refl prefl
right-identity (later x) = later (# (right-identity (b x)))

bind-assoc : (x : C ⊥) (f : C → B ⊥) (g : B → A ⊥)
  → bind (bind x f) g ≅ bind x (λ y → bind (f y) g)
bind-assoc (now x) f g = refl prefl
bind-assoc (later x) f g = later (# bind-assoc (b x) f g)
```

Luego de las propiedades del operador `bind`, se demuestra su compatibilidad con la relación de orden. Para ello se crea otro módulo anónimo que tiene dos parámetros,  $A$  y  $B$  de tipo `Set`. Antes de realizar la prueba se abre el módulo `Equality` para cada uno de los conjuntos de retorno, utilizando para ambos la igualdad proposicional y renombrando los términos a utilizar de manera que no haya conflicto de nombres. Finalmente, se abre `Equality.Rel`, renombrando `laterl` como `≲laterl`.

```
module _ {A B : Set} where

open Equality {A} _≡_
  renaming (_≅_ to _≅A_ ; _≲_ to _≲A_ ; now to nowA ; later to laterA)
open Equality {B} _≡_
  renaming (_≅_ to _≅B_ ; _≲_ to _≲B_ ; now to nowB ; later to laterB)
open Equality.Rel renaming (laterl to ≲laterl)

bind-comp : (a1 a2 : A ⊥) (f1 f2 : A → B ⊥) → a1 ≲A a2
  → (∀ (a : A) → f1 a ≲B f2 a) → bind a1 f1 ≲B bind a2 f2
bind-comp (now a1) (now a1) f1 f2 (now prefl) f1 ≲f2 = f1 ≲f2 a1
bind-comp (now a1) (later a2) f1 f2 (≲laterl a1 ≲ a2) f1 ≲f2 =
  ≲laterl (bind-comp (now a1) (b a2) f1 f2 a1 ≲ a2 f1 ≲f2)
bind-comp (later a1) (later a2) f1 f2 (later a1 ≲ a2) f1 ≲f2 =
  later (# (bind-comp (b a1) (b a2) f1 f2 (b a1 ≲ a2) f1 ≲f2))
bind-comp (later a1) (later a2) f1 f2 (≲laterl a1 ≲ a2) f1 ≲f2 =
  later (# (bind-comp (b a1) (b a2) f1 f2 (laterr-1 a1 ≲ a2) f1 ≲f2))
```

La prueba toma dos valores  $a_1$  y  $a_2$  de tipo  $A \perp$ , dos funciones  $f_1$  y  $f_2$  de tipo  $A \rightarrow B \perp$ , una prueba de que  $a_1 \lesssim_A a_2$  y una prueba de que para cualquier  $a : A$  se cumple que  $f_1 a \lesssim_B f_2 a$ , es decir que  $f_1 \lesssim f_2$ .

En el caso base, donde ambos valores son de la forma  $(\text{now } x)$ , al hacer *pattern matching* sobre la prueba  $a_1 \lesssim a_2$ , se obtiene que está dada por el término  $(\text{now } \text{prefl})$ , lo cual hace que, obligatoriamente, el valor de retorno del segundo argumento sea igual al del primero. El objetivo es ver que  $\text{bind } (\text{now } a_1) f_1 \lesssim \text{bind } (\text{now } a_2) f_2$ , lo cual por definición es lo mismo que escribir:  $f_1 a_1 \lesssim f_2 a_2$ . Aquí se evidencia la necesidad de poder unificar ambos valores a la misma variable, puesto que la prueba  $f_1 \lesssim f_2$  sólo puede usarse si se trata del mismo valor  $a : A$ . Gracias al patrón con punto dado por el *pattern matching* de  $\text{prefl}$ , ambos valores de retorno son  $a_1$  y el objetivo es probar que  $f_1 a_1 \lesssim f_2 a_1$ , lo cual se demuestra aplicando la prueba  $f_1 \lesssim f_2$  al valor  $a_1 : A$ .

En el segundo caso, el primer argumento es  $(\text{now } a_1)$ , y el segundo,  $(\text{later } a_2)$ . La prueba de que el primero es menor o igual al segundo está dada por el constructor  $\lesssim \text{later}'$  junto con una prueba  $a_1 \lesssim a_2$  de tipo  $(\text{now } a_1) \lesssim \text{B } (\text{b } a_2)$ . Puede parecer a simple vista que el constructor es erróneo, pero en el módulo `Equality` se define la relación de menor o igual como la relación inversa del mayor o igual, por lo que los constructores quedan “invertidos”. El constructor  $\lesssim \text{later}'$  hace referencia en realidad al término  $(\text{later } a_2) \gtrsim \text{B } (\text{now } a_1)$ . La prueba en este caso se reduce a utilizar el mismo constructor con una llamada co-recursiva dentro, en la cual el primer argumento queda igual y el segundo se “achica”. La prueba de que el primero es menor o igual al segundo está dada por la variable  $a_1 \lesssim a_2$  que, como se mencionó antes, tiene el tipo correcto. Las funciones y la prueba de su relación quedan iguales.

En los dos últimos casos los primeros dos argumentos son de la forma  $(\text{later } x)$ . No podría darse el caso de que el primero esté construido con  $\text{later}$  y el segundo se construya con  $\text{now}$  puesto que no habría manera de dar una prueba de que el primero es menor o igual al segundo. Luego la prueba es exhaustiva aunque no contemple este caso. La diferencia entre el penúltimo y el último caso está en la prueba de desigualdad entre los primeros argumentos. En el primero esta prueba está dada por un constructor  $\text{later}$  con una prueba  $a_1 \lesssim a_2$  de tipo  $\infty ((\text{b } a_1) \lesssim \text{B } (\text{b } a_2))$ . En este caso la demostración se realiza con el mismo constructor y dentro de él una llamada co-recursiva (suspendida) donde los dos primeros argumentos se “achican” y la prueba de que el primero es menor o igual al segundo está dada por  $(\text{b } a_1 \lesssim a_2)$ . Una vez más las funciones y su relación quedan iguales.

En el último caso, la prueba de que  $(\text{later } a_1)$  es menor o igual a  $(\text{later } a_2)$  está dada por el constructor  $\lesssim \text{later}'$  y una prueba  $a_1 \lesssim a_2$  de tipo  $((\text{later } a_1) \lesssim \text{B } (\text{b } a_2))$ . La demostración de este caso será similar a la del anterior, donde se utiliza el constructor  $\text{later}$  y los dos primeros argumentos se “achican”. La diferencia está en que, en este caso, no se puede utilizar directamente la prueba  $a_1 \lesssim a_2$  puesto que esta debería tener tipo  $((\text{b } a_1) \lesssim \text{B } (\text{b } a_2))$  en lugar de  $((\text{later } a_1) \lesssim \text{B } (\text{b } a_2))$ , por lo que tiene un constructor  $\text{later}$  en el lado izquierdo que sobra. Esto se soluciona aplicando la función  $\text{later}^{r-1}$  a la prueba para quitar el constructor de más (una vez más se debe recordar que los lados se invierten porque la relación de base es mayor o igual).

Una vez demostradas todas las propiedades del operador `bind`, se pasa a demostrar las correspondientes al operador `merge`. Estas pruebas serán similares a las dadas para `MonoidalFunctor`, salvo que en este caso no se utilizará `fmap` sino su equivalente construido a partir de `bind`, como se explicó en la sección 4.4. Antes de pasar a las propiedades, se demostrará la equivalencia entre ambas expresiones de manera que no queden dudas de que las propiedades del campo `merge` están bien definidas. La demostración no presenta mayores dificultades.

```
module _ {A B : Set} where
  open Equality {B} _≡_ using (_≅_)
  open Equality.Rel
```

```

fmap~bind : (f : A → B) → (a : A ⊥)
           → (fmap f a) ≅ ((λ ma → bind ma (λ a → now (f a))) a)
fmap~bind f (now a) = now prefl
fmap~bind f (later a) = later (# (fmap~bind f (b a)))

```

La primera propiedad de `merge` que se demuestra es su asociatividad, para la cual se crea un módulo anónimo con tres argumentos implícitos que son los tres conjuntos,  $A$ ,  $B$  y  $C$ , que se utilizan en la prueba. Antes de demostrar esta propiedad se abre el módulo `Equality`, esta vez para el producto cartesiano de los tres conjuntos mencionados, tomando como igualdad en el producto la igualdad proposicional y extrayendo del módulo la relación de bisemejanza fuerte. Al igual que antes se abre también `Equality.Rel`.

```

module _ {A B C : Set} where

open Equality {A × B × C} _≡_ using (_≡_)
open Equality.Rel

merge-assoc : (a : A ⊥) (b : B ⊥) (c : C ⊥)
             → (bind (merge (merge a b) c) (λ {((a , b) , c) → now (a , (b , c))}) )
               ≅ (merge a (merge b c))

merge-assoc (now a) (now b) (now c) = now prefl
merge-assoc (now a) (now b) (later c) = later (# (merge-assoc (now a) (now b) (b c)))
merge-assoc (now a) (later b) (now c) = later (# (merge-assoc (now a) (b b) (now c)))
merge-assoc (now a) (later b) (later c) = later (# (merge-assoc (now a) (b b) (b c)))
merge-assoc (later a) (now b) (now c) = later (# (merge-assoc (b a) (now b) (now c)))
merge-assoc (later a) (now b) (later c) = later (# (merge-assoc (b a) (now b) (b c)))
merge-assoc (later a) (later b) (now c) = later (# (merge-assoc (b a) (b b) (now c)))
merge-assoc (later a) (later b) (later c) = later (# (merge-assoc (b a) (b b) (b c)))

```

El tipo de la prueba se corresponde con el tipo del campo `assoc`, el cual se obtiene como se explicó en la sección 4.4. La demostración es, en esencia, igual a la introducida en la sección 5.3 para `MonoidalFunctor`. El único cambio es la reflexividad de la relación de los tipos de retorno que, en este caso, pasa a ser `prefl`.

Las demostraciones de que `(now tt)` es neutro a izquierda y derecha respecto de `merge` también son iguales a las probadas para `MonoidalFunctor`. Los únicos cambios son la igualdad con la que se abre el módulo `Equality` y la reflexividad que se utiliza en el caso base.

```

module _ {A : Set} where

open Equality {A × T} _≡_ using (_≡_)
open Equality.Rel

rid : (a : A ⊥) → (merge a unit) ≅ (bind a (λ a → now (a , tt)))
rid (now x) = now prefl
rid (later x) = later (# (rid (b x)))

module _ {A : Set} where

open Equality {T × A} _≡_ using (_≡_)

```

open Equality.Rel

lid : (a : A  $\perp$ )  $\rightarrow$  (merge unit a)  $\cong$  (bind a ( $\lambda$  a  $\rightarrow$  now (tt , a)))

lid (now x) = now prefl

lid (later x) = later ( $\#$  (lid (b x)))

Al igual que con el operador `bind`, se debe demostrar la compatibilidad de `merge` con la relación de orden. Esto se realiza dentro de un módulo anónimo que toma como argumentos dos conjuntos,  $A$  y  $B$ . Se abre el módulo `Equality` para tres conjuntos distintos:  $A$ ,  $B$  y  $A \times B$ . En los tres casos se utiliza la igualdad proposicional, en los primeros dos se renombran los elementos a utilizar para que no haya conflictos de nombre. Finalmente se abre `Equality.Rel` renombrando el constructor `laterl` como  `$\lesssim$ laterl` por la misma razón.

module \_ {A B : Set} where

open Equality {A}  $\_ \equiv \_$

renaming ( $\_ \equiv \_$  to  $\_ \cong A \_$ ;  $\_ \lesssim \_$  to  $\_ \lesssim A \_$ ; now to nowA; later to laterA)

open Equality {B}  $\_ \equiv \_$

renaming ( $\_ \equiv \_$  to  $\_ \cong B \_$ ;  $\_ \lesssim \_$  to  $\_ \lesssim B \_$ ; now to nowB; later to laterB)

open Equality {A  $\times$  B}  $\_ \equiv \_$  using ( $\_ \lesssim \_$ ;  $\_ \cong \_$ )

open Equality.Rel renaming (later<sup>l</sup> to  $\lesssim$ later<sup>l</sup>)

merge-comp : (a<sub>1</sub> a<sub>2</sub> : A  $\perp$ ) (b<sub>1</sub> b<sub>2</sub> : B  $\perp$ )  $\rightarrow$  a<sub>1</sub>  $\lesssim A$  a<sub>2</sub>  $\rightarrow$  b<sub>1</sub>  $\lesssim B$  b<sub>2</sub>  
 $\rightarrow$  merge a<sub>1</sub> b<sub>1</sub>  $\lesssim$  merge a<sub>2</sub> b<sub>2</sub>

merge-comp (now a<sub>1</sub>) (now a<sub>1</sub>) (now b<sub>1</sub>) (now b<sub>1</sub>) (now prefl) (now prefl) = now prefl

merge-comp (now a<sub>1</sub>) (now a<sub>2</sub>) (later b<sub>1</sub>) (later b<sub>2</sub>) (now a $\sim$ ) (later b $\lesssim$ ) =  
 later ( $\#$  merge-comp (now a<sub>1</sub>) (now a<sub>2</sub>) (b b<sub>1</sub>) (b b<sub>2</sub>) (now a $\sim$ ) (b b $\lesssim$ )))

merge-comp (now a<sub>1</sub>) (now a<sub>2</sub>) (now b<sub>1</sub>) (later b<sub>2</sub>) (now a $\sim$ ) ( $\lesssim$ later<sup>l</sup> b $\lesssim$ ) =  
 $\lesssim$ later<sup>l</sup> (merge-comp (now a<sub>1</sub>) (now a<sub>2</sub>) (now b<sub>1</sub>) (b b<sub>2</sub>) (now a $\sim$ ) b $\lesssim$ ))

merge-comp (now a<sub>1</sub>) (now a<sub>2</sub>) (later b<sub>1</sub>) (later b<sub>2</sub>) (now a $\sim$ ) ( $\lesssim$ later<sup>l</sup> b $\lesssim$ ) =

later ( $\#$  (merge-comp (now a<sub>1</sub>) (now a<sub>2</sub>) (b b<sub>1</sub>) (b b<sub>2</sub>) (now a $\sim$ ) (later<sup>r-1</sup> b $\lesssim$ ))))

merge-comp (later a<sub>1</sub>) (later a<sub>2</sub>) (now b<sub>1</sub>) (now b<sub>2</sub>) (later a $\lesssim$ ) (now b $\sim$ ) =

later ( $\#$  (merge-comp (b a<sub>1</sub>) (b a<sub>2</sub>) (now b<sub>1</sub>) (now b<sub>2</sub>) (b a $\lesssim$ ) (now b $\sim$ ))))

merge-comp (later a<sub>1</sub>) (later a<sub>2</sub>) (later b<sub>1</sub>) (later b<sub>2</sub>) (later a $\lesssim$ ) (later b $\lesssim$ ) =

later ( $\#$  (merge-comp (b a<sub>1</sub>) (b a<sub>2</sub>) (b b<sub>1</sub>) (b b<sub>2</sub>) (b a $\lesssim$ ) (b b $\lesssim$ ))))

merge-comp (later a<sub>1</sub>) (later a<sub>2</sub>) (now b<sub>1</sub>) (later b<sub>2</sub>) (later a $\lesssim$ ) ( $\lesssim$ later<sup>l</sup> b $\lesssim$ ) =

later ( $\#$  (merge-comp (b a<sub>1</sub>) (b a<sub>2</sub>) (now b<sub>1</sub>) (b b<sub>2</sub>) (b a $\lesssim$ ) b $\lesssim$ )))

merge-comp (later a<sub>1</sub>) (later a<sub>2</sub>) (later b<sub>1</sub>) (later b<sub>2</sub>) (later a $\lesssim$ ) ( $\lesssim$ later<sup>l</sup> b $\lesssim$ ) =

later ( $\#$  (merge-comp (b a<sub>1</sub>) (b a<sub>2</sub>) (b b<sub>1</sub>) (b b<sub>2</sub>) (b a $\lesssim$ ) (later<sup>r-1</sup> b $\lesssim$ ))))

merge-comp (now a<sub>1</sub>) (later a<sub>2</sub>) (now b<sub>1</sub>) (now b<sub>2</sub>) ( $\lesssim$ later<sup>l</sup> a $\lesssim$ ) (now b $\sim$ ) =

$\lesssim$ later<sup>l</sup> (merge-comp (now a<sub>1</sub>) (b a<sub>2</sub>) (now b<sub>1</sub>) (now b<sub>2</sub>) a $\lesssim$  (now b $\sim$ )))

merge-comp (later a<sub>1</sub>) (later a<sub>2</sub>) (now b<sub>1</sub>) (now b<sub>2</sub>) ( $\lesssim$ later<sup>l</sup> a $\lesssim$ ) (now b $\sim$ ) =

later ( $\#$  (merge-comp (b a<sub>1</sub>) (b a<sub>2</sub>) (now b<sub>1</sub>) (now b<sub>2</sub>) (later<sup>r-1</sup> a $\lesssim$ ) (now b $\sim$ ))))

merge-comp (now a<sub>1</sub>) (later a<sub>2</sub>) (later b<sub>1</sub>) (later b<sub>2</sub>) ( $\lesssim$ later<sup>l</sup> a $\lesssim$ ) (later b $\lesssim$ ) =

later ( $\#$  (merge-comp (now a<sub>1</sub>) (b a<sub>2</sub>) (b b<sub>1</sub>) (b b<sub>2</sub>) a $\lesssim$  (b b $\lesssim$ ))))

merge-comp (later a<sub>1</sub>) (later a<sub>2</sub>) (later b<sub>1</sub>) (later b<sub>2</sub>) ( $\lesssim$ later<sup>l</sup> a $\lesssim$ ) (later b $\lesssim$ ) =

later ( $\#$  (merge-comp (b a<sub>1</sub>) (b a<sub>2</sub>) (b b<sub>1</sub>) (b b<sub>2</sub>) (later<sup>r-1</sup> a $\lesssim$ ) (b b $\lesssim$ ))))

merge-comp (now a<sub>1</sub>) (later a<sub>2</sub>) (now b<sub>1</sub>) (later b<sub>2</sub>) ( $\lesssim$ later<sup>l</sup> a $\lesssim$ ) ( $\lesssim$ later<sup>l</sup> b $\lesssim$ ) =

$\lesssim$ later<sup>l</sup> (merge-comp (now a<sub>1</sub>) (b a<sub>2</sub>) (now b<sub>1</sub>) (b b<sub>2</sub>) a $\lesssim$  b $\lesssim$ )

merge-comp (now a<sub>1</sub>) (later a<sub>2</sub>) (later b<sub>1</sub>) (later b<sub>2</sub>) ( $\lesssim$ later<sup>l</sup> a $\lesssim$ ) ( $\lesssim$ later<sup>l</sup> b $\lesssim$ ) =

$$\begin{aligned}
& \text{later } (\# (\text{merge-comp } (\text{now } a_1) (\flat a_2) (\flat b_1) (\flat b_2) a \lesssim (\text{later}^{r-1} b \lesssim))) \\
\text{merge-comp } (\text{later } a_1) (\text{later } a_2) (\text{now } b_1) (\text{later } b_2) (\lesssim \text{later}^l a \lesssim) (\lesssim \text{later}^l b \lesssim) = \\
& \text{later } (\# (\text{merge-comp } (\flat a_1) (\flat a_2) (\text{now } b_1) (\flat b_2) (\text{later}^{r-1} a \lesssim) b \lesssim)) \\
\text{merge-comp } (\text{later } a_1) (\text{later } a_2) (\text{later } b_1) (\text{later } b_2) (\lesssim \text{later}^l a \lesssim) (\lesssim \text{later}^l b \lesssim) = \\
& \text{later } (\# (\text{merge-comp } (\flat a_1) (\flat a_2) (\flat b_1) (\flat b_2) (\text{later}^{r-1} a \lesssim) (\text{later}^{r-1} b \lesssim)))
\end{aligned}$$

Esta prueba toma cuatro valores:  $a_1$  y  $a_2$  de tipo  $A \perp$  y  $b_1$  y  $b_2$  de tipo  $B \perp$ . Además toma dos pruebas: una demuestra que  $a_1$  es menor o igual a  $a_2$  y la otra que  $b_1$  es menor o igual a  $b_2$ . Cada uno de los cuatro valores puede tener dos formas,  $(\text{now } x)$  o  $(\text{later } x)$ , y las dos pruebas pueden estar dadas de tres formas distintas:  $(\text{now } x \sim)$ ,  $(\text{later } x \lesssim)$  y  $(\lesssim \text{later}^l x \lesssim)$ , salvo por algunas pocas combinaciones que no son posibles. Esto hace que la prueba tenga muchos casos y explicar uno por uno sea muy largo y tedioso. Se detallan a continuación algunos lineamientos generales:

- En el caso base se puede observar cómo, una vez más, se hace uso del *pattern matching* en `prefl` para unificar variables y resolver, en este caso, la igualdad en el producto cartesiano. Si se tuviera una igualdad parametrizada para el conjunto  $A$ , otra para el conjunto  $B$  y otra para  $A \times B$ , no se podría deducir la igualdad del producto cartesiano a partir de las igualdades individuales.
- El resto de los casos se resuelve con uno de los constructores no básicos, `later` o  `$\lesssim \text{later}^l$` , y una llamada co-recursiva dentro.
- En las llamadas co-recursivas todos los valores que tenían la forma  $(\text{later } x)$  se reducen a  $(\flat x)$ , sin importar qué forma tenga la prueba de desigualdad que lo involucra.
- En los casos en los que, por la forma de la prueba de desigualdad, sobra un constructor `later` en uno de los lados para poder utilizar la prueba en la llamada co-recursiva, se utiliza la función  `$\text{later}^{r-1}$`  para solucionarlo al igual que en la prueba de compatibilidad del `bind`.

Queda por demostrar una última propiedad para el operador `merge`: la conmutatividad. Esta es requerida para que el functor monoidal agregado a la mónada sea simétrico y se demuestra dentro del mismo módulo anónimo que la propiedad anterior ya que requiere los mismos conjuntos. La prueba es bastante simple, el caso base es trivial y los demás casos se resuelven simplemente utilizando el constructor `later` con una llamada co-recursiva suspendida dentro. En las llamadas co-recursivas los términos del tipo  $(\text{now } x)$  quedan igual y los que tienen la forma  $(\text{later } x)$  se reducen a  $(\flat x)$ .

$$\begin{aligned}
\text{merge-comm} & : (a : A \perp) \rightarrow (b : B \perp) \\
& \rightarrow \text{merge } a \, b \cong \text{bind } (\text{merge } b \, a) (\lambda \{ (a, b) \rightarrow \text{now } (b, a) \}) \\
\text{merge-comm } (\text{now } a) (\text{now } b) & = \text{now prefl} \\
\text{merge-comm } (\text{now } a) (\text{later } b) & = \text{later } (\# (\text{merge-comm } (\text{now } a) (\flat b))) \\
\text{merge-comm } (\text{later } a) (\text{now } b) & = \text{later } (\# (\text{merge-comm } (\flat a) (\text{now } b))) \\
\text{merge-comm } (\text{later } a) (\text{later } b) & = \text{later } (\# (\text{merge-comm } (\flat a) (\flat b)))
\end{aligned}$$

Se definen a continuación las relaciones de igualdad y orden. Estas son extraídas del módulo `Equality` utilizando como igualdad subyacente la igualdad proposicional.

$$\begin{aligned}
\_ \cong \_ & : \forall \{A\} \rightarrow A \perp \rightarrow A \perp \rightarrow \text{Set} \\
\_ \cong \_ \{A\} & = \text{Equality.} \_ \cong \_ \{A\} \_ \equiv \_ \\
\_ \lesssim \_ & : \forall \{A\} \rightarrow A \perp \rightarrow A \perp \rightarrow \text{Set} \\
\_ \lesssim \_ \{A\} & = \text{Equality.} \_ \lesssim \_ \{A\} \_ \equiv \_
\end{aligned}$$

Una vez definidas ambas relaciones, se prueba que estas son una relación de equivalencia y un orden parcial, respectivamente. Para esto se incluyen del módulo `Equivalence` las pruebas

`refl` y `trans` que prueban la reflexividad y transitividad para ambas relaciones, `sym` que prueba la simetría de la igualdad y `antisym` que prueba la antisimetría del orden. La prueba de que la bisemejanza fuerte es una relación de equivalencia es similar a las explicadas en las secciones anteriores, con la diferencia de que la reflexividad, simetría y transitividad de la igualdad subyacente están dadas ahora por las pruebas correspondientes de la igualdad proposicional importadas del módulo `Relation.Binary.PropositionalEquality`.

```
open Equivalence using (refl; sym; trans; antisym)

eq≅⊥ : ∀ {A} → IsEquivalence ( _≅⊥_ {A} )
eq≅⊥ = record
  { refl = refl prefl ;
    sym = sym psym tt ;
    trans = trans ptrans }

porder≤⊥ : ∀ {A} → IsPartialOrder ( _≅⊥_ {A} ) ( _≤⊥_ {A} )
porder≤⊥ {A} = record
  { isPreorder = record
    { isEquivalence = eq≅⊥ ;
      reflexive = ≅⊥⇒≤⊥ ;
      trans = trans≤⊥ (trans ptrans) } ;
    antisym = antisym≤⊥ antisym }

where
  ≅⊥⇒≤⊥ : ∀ {A} → _≅⊥_ {A} ⇒ _≤⊥_ {A}
  ≅⊥⇒≤⊥ (Equality.now nowx~nowy) = Equality.now (psym nowx~nowy)
  ≅⊥⇒≤⊥ (Equality.later x~y) = Equality.later (λ (≅⊥⇒≤⊥ (λ x~y)))
  trans≤⊥ : ∀ {A} → Transitive (Equality._≥⊥_ {A} _≡_) → Transitive ( _≤⊥_ {A} )
  trans≤⊥ trans≥ ij jk = flip trans≥ ij jk
  antisym≤⊥ : ∀ {A} → Antisymmetric _≅⊥_ (Equality._≥⊥_ {A} _≡_)
    → Antisymmetric ( _≅⊥_ {A} ) ( _≤⊥_ {A} )
  antisym≤⊥ antisym≥ ij ji = flip antisym≥ ij ji
```

La prueba de que la relación de orden es un orden parcial se realiza respecto de la relación de igualdad. Esta consta de dos partes: la primera prueba que es un preorden y la segunda agrega la antisimetría. La prueba de que es un preorden requiere, primero, una prueba de que la relación de igualdad es una relación de equivalencia, para lo cual se utiliza la prueba `eq≅⊥` realizada previamente.

Luego se pide la reflexividad en términos de la relación de igualdad establecida, es decir que se pide que la relación `≅⊥` implique la relación `≤⊥`. Esta prueba está dada por el término `≅⊥⇒≤⊥`. En el caso base la prueba de igualdad está dada por el constructor `Equality.now` y una prueba `nowx~nowy` de que  $x \equiv y$ . Para dar la prueba de la desigualdad, no se da la prueba de que  $(\text{now } x) \leq (\text{now } y)$ , sino que se prueba que  $(\text{now } y) \geq (\text{now } x)$ , puesto que el menor o igual se define en términos del mayor o igual. Por lo tanto, se requiere una prueba de que  $y \equiv x$ , la cual se obtiene aplicando la simetría de la igualdad proposicional a la prueba `nowx~nowy`. El otro caso se resuelve co-recursivamente.

La última prueba pedida para que la relación sea un preorden es la transitividad. La prueba `trans`, importada del módulo `Equivalence`, prueba la transitividad para el mayor o igual, es decir que si  $x \geq y$  e  $y \geq z$  entonces  $x \geq z$ . Esto en términos del menor o igual es equivalente a decir que si  $y \leq x$  y  $z \leq y$  entonces  $z \leq x$ . Como se puede observar, esto no es exactamente la definicion



de transitividad, puesto que las hipótesis deberían estar al revés, lo cual se soluciona aplicando `flip` a la prueba de transitividad dada como se realiza en `trans $\lesssim$  $\perp$` .

A la hora de probar la antisimetría, por último, sucede algo similar que hace necesario aplicar `flip` a la prueba de antisimetría dada puesto que cambia el orden en que se toman las pruebas de desigualdad respecto del orden en que se toman los argumentos implícitos que son los valores sobre los cuales se realizan las pruebas de desigualdad. En `antisym $\lesssim$  $\perp$`  se realiza este cambio y queda así concluida la prueba de que la relación de orden es un orden parcial.

Con todas las demostraciones realizadas hasta esta parte, se puede generar la instancia de `ConcurrentMonad  $\perp$` , quedando únicamente el hueco para la prueba de la ley de intercambio.

```
open import Structures.ConcurrentMonad hiding (unit; merge)

partiality : ConcurrentMonad  $\perp$ 
partiality = makeConcurrentMonad
  _ $\cong$  $\perp$ _
  eq $\cong$  $\perp$ _
  _ $\lesssim$  $\perp$ _
  porder $\lesssim$  $\perp$ _
  now
  bind
  bind-comp
  left-identity
  right-identity
  bind-assoc
  unit
  merge
  merge-comp
  rid
  lid
  merge-assoc
  merge-comm
  {! !}
```

A la hora de intentar probar la ley de intercambio, empezaron a surgir varios problemas. A continuación se muestran varias pruebas que se fueron realizando y lemas que se fueron definiendo en el intento de poder demostrarlo. Estos intentos fueron realizados dentro de un módulo anónimo que toma como parámetros implícitos cuatro conjuntos. Dentro de él se abre el módulo `Equality` para tres conjuntos distintos,  $C$ ,  $D$  y  $C \times D$ , haciendo los renombres correspondientes. Al igual que antes también se abren `Equality.Rel` renombrando el constructor `laterl` y el módulo `Equivalence`.

```
module _ {A B C D : Set} where

open Equality {C} _ $\equiv$ _
  renaming (_ $\cong$ _ to _ $\cong$ C_; _ $\gtrsim$ _ to _ $\gtrsim$ C_; now to nowC; later to laterC)
open Equality {D} _ $\equiv$ _
  renaming (_ $\cong$ _ to _ $\cong$ D_; _ $\gtrsim$ _ to _ $\gtrsim$ D_; now to nowD; later to laterD)
open Equality {C  $\times$  D} _ $\equiv$ _ using (_ $\gtrsim$ _)
open Equality.Rel renaming (laterl to  $\lesssim$ laterl)
open Equivalence
```



Los primeros tres lemas prueban propiedades básicas cuya necesidad surgió en el proceso de intentar probar la ley de intercambio. `merge-ext` prueba que, si se tienen cuatro valores,  $c_1$  y  $c_2$  de tipo  $C \perp$  y  $d_1$  y  $d_2$  de tipo  $D \perp$ , tales que  $c_1 \cong_C c_2$  y  $d_1 \cong_D d_2$ , entonces se cumple que `merge`  $c_1$   $d_1 \gtrsim$  `merge`  $c_2$   $d_2$ .

```
merge-ext : (c1 c2 : C ⊥) (d1 d2 : D ⊥) → (c1 ≅C c2) → (d1 ≅D d2)
→ merge c1 d1 ≳ merge c2 d2
merge-ext (now c1) (now .c1) (now d1) (now .d1) (now prefl) (now prefl) = now prefl
merge-ext (now c1) (now c2) (later d1) (later d2) pc (later d1 ~ d2) =
  later (# (merge-ext (now c1) (now c2) (b d1) (b d2) pc (b d1 ~ d2)))
merge-ext (later c1) (later c2) (now d1) (now d2) (later c1 ~ c2) pd =
  later (# (merge-ext (b c1) (b c2) (now d1) (now d2) (b c1 ~ c2) pd))
merge-ext (later c1) (later c2) (later d1) (later d2) (later c1 ~ c2) (later d1 ~ d2) =
  later (# (merge-ext (b c1) (b c2) (b d1) (b d2) (b c1 ~ c2) (b d1 ~ d2)))
```

Las dos propiedades que siguen prueban que si dos términos son iguales según la igualdad proposicional, entonces deben ser también iguales según la bisemejanza fuerte. Se realizan dos pruebas por separado para cada tipo de retorno que fue necesario ya que la bisemejanza fuerte con cada tipo de retorno está nombrada de una manera diferente.

```
≡⇒≅C : (c1 c2 : C ⊥) → (c1 ≡ c2) → c1 ≅C c2
≡⇒≅C (now c1) (now .c1) prefl = now prefl
≡⇒≅C (later c1) (later .c1) prefl = later (# (≡⇒≅C (b c1) (b c1) prefl))

≡⇒≅D : (d1 d2 : D ⊥) → (d1 ≡ d2) → d1 ≅D d2
≡⇒≅D (now d1) (now .d1) prefl = now prefl
≡⇒≅D (later d1) (later .d1) prefl = later (# (≡⇒≅D (b d1) (b d1) prefl))
```

Se muestra a continuación el intento de prueba de la ley de intercambio, en el cual se utilizan dos lemas que se describirán luego.

```
interchange : (a : A ⊥) (b : B ⊥) (f : A → C ⊥) (g : B → D ⊥)
→ (bind (merge a b) (λ { (a , b) → merge (f a) (g b) } ))
  ≳ (merge (bind a f) (bind b g))

interchange (now a) (now b) f g = refl prefl
interchange (now a) (later b) f g with f a | inspect f a
... | now x | [ eq ] = later (# lema1 a b x f g eq)
... | later y | [ eq ] = later (# lema2 a b y f g eq)
interchange (later a) b f g = {! !}
```

Si se observa la prueba, se puede ver que en el segundo caso se abre con una sentencia `with` el término  $f a$ , de manera que se puedan analizar los diferentes resultados posibles de la aplicación. Al intentar resolver estos casos, surge un problema ya que Agda no logra unificar, en el primer caso, por ejemplo, el término  $f a$  con `now x`. Si bien ambos términos son iguales por definición de la expresión `with`, en el objetivo a demostrar de ese caso aparece de un lado de la desigualdad la expresión  $f a$  y, del otro, `now x`. En el intento de solucionar este problema, se agrega a la cláusula `with` el término `inspect f a` que lo que hace cuando uno realiza *pattern matching* sobre el mismo es dar una prueba de que los términos en cuestión son iguales. Así se crea el `lema1` en el intento de demostrar ese caso específico. En su tipo se puede observar cómo, de un lado de la desigualdad, aparece el término  $f a$  y, del otro, `now x`, y se pide como argumento una prueba

de que estos dos términos son iguales (según la igualdad proposicional ya que es lo que provee `inspect`). En la prueba del `lema1` se utilizan los tres lemas anteriores.

```

lema1 : (a : A) (b : ∞ (B ⊥)) (c : C) (f : A → C ⊥) (g : B → D ⊥) → (p : (f a) ≡ (now c))
  → (bind (merge (now a) (b b)) (λ { (a , b) → merge (f a) (g b) }))
  ≃ (merge (now c) (bind (b b) g))

lema1 a b c f g p with b b
... | now b1 =
  merge-ext (f a) (now c) (g b1) (g b1)
    ((⇒≅C (f a) (now c) p)) ((⇒≅D (g b1) (g b1) prefl))
... | later b2 = later (# lema1 a b2 c f g p)

lema2 : (a : A) (b : ∞ (B ⊥)) (y : ∞ (C ⊥)) (f : A → C ⊥) (g : B → D ⊥)
  → (p : (f a) ≡ (later y))
  → (bind (merge (now a) (b b)) (λ { (a , b) → merge (f a) (g b) }))
  ≃ (merge (b y) (bind (b b) g))

lema2 a b y f g p with b b
... | now b1 with g b1
... | now b3 =
  laterr-1 (merge-ext (f a) (later y) (now b3) (now b3) (⇒≅C (f a) (later y) p) (refl prefl))
... | later b4 =
  merge-comp (b y) (f a) (later b4) (later b4) (laterr-1
    ((lsPartialOrder.reflexive porder ≃ ⊥) (⇒≅C (later y) (f a) (psym p)))) (refl prefl)
lema2 a b y f g p | later b2 with b y | inspect b y
lema2 a b y f g p | later b2 | now c1 | [ eq ] =
  later (# lema2 a b2 (# now c1) f g (ptrans p {! !}))
lema2 a b y f g p | later b2 | later c2 | [ eq ] = {! !}

```

El `lema2` es también un caso particular de `interchange`: el caso en que `f a` da como resultado `later y`. En el intento de probar este segundo lema, vuelve a surgir el mismo problema al intentar inspeccionar con `with` el término `b y`. De esta manera, la prueba se empezó a complejizar cada vez más, evidenciando un problema de base en el soporte para coinducción con notación musical.

Luego de batallar bastante se obtuvieron los siguientes ejemplos, en los cuales se puede observar el problema real de fondo.

```

≡#b : ∀ {S : Set} → (s : S ⊥) → (b (# s)) ≡ s
≡#b s = prefl

≡#b : ∀ {S : Set} → (s : ∞ (S ⊥)) → (# (b s)) ≡ s
≡#b s = {! prefl !}

⇒≡# : ∀ {S : Set} → (s1 s2 : S ⊥) → s1 ≡ s2 → (# s1) ≡ (# s2)
⇒≡# s1 .s1 prefl = {! prefl !}

```

El primer lema, `≡#b`, prueba que los operadores `b` y `#` son opuestos en el sentido de que si se tiene un `s : S ⊥`, se lo suspende con `#` y luego se lo desencapsula con `b`, se obtiene como resultado el término inicial. La prueba de este lema es trivial. Al intentar probar el lema complementario, `≡#b`, que toma un `s : ∞ (S ⊥)` y postula que `(# (b s)) ≡ s`, surgen los problemas. Agda no acepta la prueba trivial y, al observar el tipo objetivo de la prueba en el modo interactivo, aparece lo siguiente: `Instances.ConcurrentMonad.#-54 s ≡ s`. Se evidencia entonces que hay problemas a la

hora de igualar elementos contru idos con  $\#$ . M s a n, en el tercer lema,  $\Rightarrow\#$ , se intenta probar que si se tienen dos t rminos,  $s_1$  y  $s_2$ , que son iguales seg n la igualdad proposicional, entonces los resultados de aplicar  $\#$  a ambos t rminos tambi n son iguales seg n la misma igualdad. Una vez m s Agda no acepta la prueba trivial y el tipo objetivo que se observa en el modo interactivo es: `Instances.ConcurrentMonad.#-56 s1 s1  $\equiv$  Instances.ConcurrentMonad.#-57 s1 s1`. Con estos ejemplos se evidencia que esta notaci n, en particular el operador  $\#$ , tiene problemas para deducir la igualdad de los t rminos, es como si hubiera diferentes tipos de  $\#$ , con diferentes n meros cada uno.

## 5.5. Reduciendo el problema a los conaturales

A ra z de las dificultades encontradas en el intento de probar la ley de intercambio para la m nada *delay*, se decidi  simplificar el problema e intentar probar que los n meros conaturales forman un monoide concurrente, lo cual implica probar la ley de intercambio para dicho conjunto. Los n meros conaturales pueden verse como una versi n simplificada del tipo *delay*, en donde no hay un tipo de retorno, o bien, este tipo es  $\top$ . As , cada constructor *later* se corresponder a con un constructor *suc*. El objetivo de esta simplificaci n es ver si el problema para reconocer t rminos iguales se debe al valor que (posiblemente) retornan los t rminos de tipo  $A \perp$ .

### 5.5.1. Definici n de los conaturales con notaci n musical

### 5.5.2.  Se puede probar que los conaturales forman un monoide concurrente?

## 5.6. Cambio de paradigma: *sized types*

### 5.6.1. Definici n de los conaturales utilizando *sized types*

### 5.6.2. Prueba de que los conaturales forman un monoide concurrente

### 5.6.3. Prueba alternativa de que *delay* es una m nada concurrente

## Ap ndice

En la secci n 5.2 se prueba que el tipo *delay* es una m nada tomando como relaci n de igualdad la bisemejanza d bil. El siguiente bloque de c digo corresponde a la prueba an loga, tomando en este caso como relaci n de igualdad la bisemejanza fuerte.

```
module Strong (_ ~_ :  $\forall \{A\} \rightarrow A \rightarrow A \rightarrow \text{Set}$ )
  (eq~ :  $\forall \{A\} \rightarrow \text{IsEquivalence } (\_ \sim \_ \{A\})$ ) where

module _ {A : Set} { _ ~_ : A  $\rightarrow$  A  $\rightarrow$  Set } (refl~ : Reflexive _ ~_) where

  open Equality _ ~_ using (_  $\cong$  _)
  open Equality.Rel
  open Equivalence using (refl)
```

```

left-identity : (x : B) (f : B → A ⊥) → bind (now x) f ≅ f x
left-identity x f = refl refl~

right-identity : (t : A ⊥) → bind t now ≅ t
right-identity (now x) = refl refl~
right-identity (later x) = later (# (right-identity (b x)))

associative : (x : C ⊥) (f : C → B ⊥) (g : B → A ⊥)
  → bind (bind x f) g ≅ bind x (λ y → bind (f y) g)
associative (now x) f g = refl refl~
associative (later x) f g = later (# (associative (b x) f g))

open import Structures.Monad

_≅⊥_ : ∀ {A} → A ⊥ → A ⊥ → Set
_≅⊥_ {A} = Equality._≅_ {A} (_~_ {A})

open Equivalence using (refl; sym; trans)

eq≅⊥ : ∀ {A} → IsEquivalence (_≅⊥_ {A})
eq≅⊥ = record
  { refl = refl (IsEquivalence.refl eq~) ;
    sym = sym (IsEquivalence.sym eq~) tt ;
    trans = trans (IsEquivalence.trans eq~) }

delayMonad : Monad _⊥
delayMonad = makeMonad
  _≅⊥_
  eq≅⊥
  now
  bind
  (left-identity (IsEquivalence.refl eq~))
  (right-identity (IsEquivalence.refl eq~))
  (associative (IsEquivalence.refl eq~))

```

La estructura de funtor monoidal para el mismo tipo de dato fue introducida en la sección 5.3, también considerando como relación de igualdad la bisemejanza débil. A continuación se muestra el código correspondiente a la estructura de funtor monoidal para el tipo *delay* con la bisemejanza fuerte como igualdad.

```

module Strong (_~_ : ∀ {A} → A → A → Set)
  (eq~ : ∀ {A} → IsEquivalence (_~_ {A})) where

module _ {A B C : Set} { _~_ : A × B × C → A × B × C → Set}
  (reflABC : Reflexive _~_) where

open Equality {A × B × C} _~_ using (_≅_)
open Equality.Rel

associative : (a : A ⊥) (b : B ⊥) (c : C ⊥)
  → (fmap (λ {(a, b), c} → (a, (b, c)))) (merge (merge a b) c)

```

```

                                                    ≅ (merge a (merge b c))
associative (now a) (now b) (now c) = now reflABC
associative (now a) (now b) (later c) = later (# (associative (now a) (now b) (b c)))
associative (now a) (later b) (now c) = later (# (associative (now a) (b b) (now c)))
associative (now a) (later b) (later c) = later (# (associative (now a) (b b) (b c)))
associative (later a) (now b) (now c) = later (# (associative (b a) (now b) (now c)))
associative (later a) (now b) (later c) = later (# (associative (b a) (now b) (b c)))
associative (later a) (later b) (now c) = later (# (associative (b a) (b b) (now c)))
associative (later a) (later b) (later c) = later (# (associative (b a) (b b) (b c)))

module _ {A : Set} {_~_ : (A × T) → (A × T) → Set}
  (reflA×T : Reflexive _~_) where

  open Equality {A × T} _~_ using (_≅_)
  open Equality.Rel

  rid : (a : A ⊥) → (merge a unit) ≅ (fmap (λ a → (a , tt)) a)
  rid (now x) = now reflA×T
  rid (later x) = later (# (rid (b x)))

module _ {A : Set} {_~_ : (T × A) → (T × A) → Set}
  (reflT×A : Reflexive _~_) where

  open Equality {T × A} _~_ using (_≅_)
  open Equality.Rel

  lid : (a : A ⊥) → (merge unit a) ≅ (fmap (λ a → (tt , a)) a)
  lid (now x) = now reflT×A
  lid (later x) = later (# (lid (b x)))

open import Structures.MonoidalFunctor hiding (unit; merge; fmap)

_≅⊥_ : ∀ {A} → A ⊥ → A ⊥ → Set
_≅⊥_ {A} = Equality._≅_ {A} (_~_ {A})

open Equivalence using (refl; sym; trans)

eq≅⊥ : ∀ {A} → IsEquivalence (_≅⊥_ {A})
eq≅⊥ = record
  { refl = refl (IsEquivalence.refl eq~) ;
    sym = sym (IsEquivalence.sym eq~) tt ;
    trans = trans (IsEquivalence.trans eq~) }

delayMonoidal : MonoidalFunctor _⊥
delayMonoidal = makeMonoidalFunctor
  _≅⊥_
  eq≅⊥
  unit
  merge
  fmap
  (rid (IsEquivalence.refl eq~))
  (lid (IsEquivalence.refl eq~))
  (associative (IsEquivalence.refl eq~))

```



## 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.
- [Gor17] Gordon, Mike: *Corecursion and coinduction: what they are and how they relate to recursion and induction*. 2017. <https://api.semanticscholar.org/CorpusID:9734802>.
- [HHM<sup>+</sup>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.
- [JR12] Jacobs, Bart y Jan Rutten: *An introduction to (co)algebra and (co)induction*. En Sangiorgi, Davide y Jan J. M. M. Rutten (editores): *Advanced Topics in Bisimulation and Coinduction*, volumen 52 de *Cambridge tracts in theoretical computer science*, páginas 38–99. Cambridge University Press, 2012.
- [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.
- [Man76] Manes, Ernest G.: *Algebraic Theories*. Graduate Texts in Mathematics. Springer New York, NY, 1976, ISBN 978-1-4612-9862-5.
- [MT91] Milner, Robin y Mads Tofte: *Co-induction in relational semantics*. Theoretical Computer Science, 87(1):209–220, 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.