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

UNR Universidad Nacional de Rosario

Facultad de Ciencias Exactas, Ingeniería y Agrimensura FCEIA

Resumen

Índice general

Capítulo 1

Introducción

En este primer capítulo se busca dar una introducción al trabajo, explicando la motivación del mismo y presentando también sus objetivos. Además se da una breve descripción de la estructura que tendrá la tesina y qué contenidos se incluyen en cada parte.

1.1. Motivación y estado del arte

La ley de Moore postula que el número de transistores que se pueden poner en un chip de computadora se duplica (aproximadamente) cada un par de años. De manera práctica, esto significa que para obtener mejoras en la velocidad de ejecución de un programa simplemente hay que esperar: después de un par de años, los programas que se escriben hoy ejecutarán más rápido. Sin embargo, esta ley ya no se verifica en la práctica. Por razones físicas, es cada vez más costoso aumentar el número de transistores de un chip y este tipo de progreso ya no puede ser garantizado. Por lo tanto, los programadores se ven obligados a encontrar otras maneras de mejorar el desempeño de sus programas. Una de las maneras más naturales que surgió fue pensar que se pueden resolver varias tareas al mismo tiempo y, de esa manera, evitar puntos ociosos en los que sólo se está esperando un evento del entorno (por ejemplo, entrada/salida bloqueante). En los últimos años, con la masificación de los microprocesadores con múltiples núcleos, se hizo aún más evidente la necesidad de proveer al programador con la capacidad de concurrencia.

Si bien los sistemas operativos modernos proveen primitivas para manejar la concurrencia, no siempre los lenguajes de programación incorporan estas características de manera natural. Los lenguajes de programación imperativos, en general, se basan en la idea de un modelo de ejecución secuencial, donde la computación se desarrolla siguiendo una única serie de pasos. Es así que lenguajes como C o Java pueden manejar concurrencia pero sin proponer cambios radicales en la concepción del lenguaje: simplemente incluyen librerías que capturan esta capacidad de manera externa.

La misma situación se refleja en los lenguajes de programación funcional, sobre todo cuando se escriben programas con efectos. Los programas con efectos suelen representarse utilizando mónadas, de manera que los programas funcionales toman una apariencia fundamentalmente imperativa. El uso de mónadas para estructuras semánticas de lenguajes con efectos fue desarrollado originalmente por E. Moggi [Mog89, Mog91]. Poco más tarde, P. Wadler [Wad92] adaptó el concepto de manera interna en los lenguajes de programación funcional, dando origen a la programación con mónadas. Existe una gran variedad de efectos que pueden ser capturados usando mónadas internas, por ejemplo, estado, excepciones, entornos, continuaciones, etc. Al igual que en la programación imperativa, al considerar la concurrencia de los efectos, la manera usual de propuesta es mediante llamadas a funciones *ad-hoc* en lugar de usar primitivas bien fundadas que deriven de alguna estructura matemática como lo hacen las operaciones de las mónadas.

Las mónadas concurrentes fueron recientemente introducidas en una pre-impresión por M.

Jaskelioff y E. Rivas [RJ19]. Esta definición surge de categorificar la noción de monoide concurrente, definido por C. A. R. Hoare et al. [HHM+11] hace unos años. Este concepto fue definido buscando obtener primitivas bien fundadas para la concurrencia, extendiendo las mónadas con nuevos operadores. Si bien esta estructura es matemáticamente bien fundada, basada en monoides concurrentes, es considerablemente complicado encontrar y probar modelos válidos de esta estructura.

La mónada Delay fue introducida por Capretta [Cap05] con el objetivo de capturar el efecto de no terminación de programas de manera explícita y uniforme. Su estructura fue estudiada en varios artículos, entre los que se puede mencionar la tesis de N. Veltri [Vel17]. Una hipótesis particular es que la mónada *delay* puede ser dotada de una estructura de mónada concurrente. Agda es un asistente de pruebas basado en teoría de tipos con soporte para inducción-recursión que ya ha sido utilizado para estudiar la mónada *delay*.

1.2. Objetivos del trabajo

Esta tesina tiene dos objetivos principales. El primero de ellos es formalizar los conceptos de monoide concurrente y mónada concurrente en el ámbito del lenguaje de pruebas Agda. Se busca realizar esta formalización de manera completa, incluyendo dentro de las estructuras las pruebas de las propiedades necesarias para demostrar que los elementos que las conforman efectivamente cumplen con las leyes de cada estructura.

El segundo es estudiar la mónada *delay* y su implementación en Agda para luego adaptar sus operaciones a la formalización propuesta con el propósito principal de probar o refutar la hipótesis de que a la mónada *delay* se le puede dar una estructura de mónada concurrente.

1.3. Estructura de la tesina

La tesina está dividida en dos partes. La primera, denominada Preliminares, contiene tres capítulos que exponen el marco teórico que fue utilizado para el desarrollo de este trabajo. En el primero de ellos se da una introducción al lenguaje Agda, describiendo sus principales características y mostrando ejemplos de uso. El segundo presenta una introducción teórica a las mónadas concurrentes, partiendo de la teoría de categorías sobre la cual se definen las mónadas y luego yendo a lo más específico hasta dar el concepto de mónada concurrente y sus principales características. Por último, en el tercero se introduce la noción de coinducción y la definición y propiedades de la mónada *delay*.

En la segunda parte, cuyo título coincide con el de la tesina, se pueden encontrar dos capítulos. Cada capítulo se corresponde con uno de los objetivos del trabajo. En el primero se da la formalización de diversas estructuras algebraicas, partiendo desde algunas más básicas como los monoides, las mónadas y los funtores monoidales para luego llegar a las más complejas que son la meta de este trabajo: los monoides concurrentes y las mónadas concurrentes. Para cada estructura se explica cada uno de los campos introducidos y se hace un paralelismo con el marco teórico para que quede clara la relación entre la teoría y la formalización. En el siguiente capítulo se analiza el caso de la mónada *delay*. Primero se define el tipo *delay* en Agda junto con varias funciones útiles para manipularlo. Luego se prueba que este puede dotarse de una estructura de mónada y también de funtor monoidal utilizando las formalizaciones definidas. Finalmente, se intenta probar o refutar que este puede dotarse de una estructura de mónada concurrente.

Parte I

Preliminares

Capítulo 2

Introducción a Agda

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

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

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

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

2.1. 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 2.2. 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 2.6.1). 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.

2.1.1. 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 $\mathbb{N}$ es el tipo de las listas de números naturales.

2.1.2. 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 $m * m$.

2.1.3. 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 pureba de paridad nos asegura que no hay riesgo de intentar dividir por dos un número impar.

2.1.4. 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
```

2.2. 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 $(x : \mathsf{A}) \to \mathsf{B}$ para indicar el tipo de una función que toma un argumento x de tipo A y devuelve un resultado de tipo B, donde x puede aparecer en B. Un caso especial de esto es cuando x es un tipo en sí mismo. Se podría definir, por ejemplo:

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

zero' : ℕ
zero' = identity ℕ zero
```

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

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

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

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 Σ-*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 2.5.1. Se reserva el nombre Σ 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 Σ'.

```
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.

2.2.1. 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. $\{x : \mathsf{A}\} \to \mathsf{B}$ significa lo mismo que $(x : \mathsf{A}) \to \mathsf{B}$, 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)
```

2.3. Familias de tipos de datos

Se definió en el apartado 2.1.1 el tipo de las listas de tipo arbitrario parametrizado por A. Estas listas pueden tener cualquier largo, tanto una lista vacía como una lista con un millón de elementos son de tipo `List` A. En ciertos casos es útil que el tipo restrinja el largo que tiene la lista, y es así como surgen las listas de largo definido, llamadas comúnmente vectores, que se definen como sigue:

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

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

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

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

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

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

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 Σ-*types*. Escribir Σ′ ℕ (Vec A) (o lo que es equivalente, Σ′ ℕ (λ n → 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
```

2.4. Sistema de módulos

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

```
module Agda where
```

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

```
module Numbers where
  data Nat : Set where
    zero : Nat
    suc : Nat → Nat
```

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

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

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

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

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

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

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

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

2.4.1. 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)
```

2.4.2. 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.

2.5. 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.

2.5.1. 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 Σ-*types* como un tipo record, en la cual el tipo del segundo campo depende del valor del primero.

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

open Σ public
```

2.6. Características adicionales

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

2.6.1. Universos

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

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

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

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

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

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

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

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

```
lzero : Level
lsuc : (n : Level) → Level
_⊔_ : (n m : Level) → Level
```

2.6.2. 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
```

2.6.3. Coinducción

Agda tiene varios soportes diferentes para coinducción. Se describirán algunos de ellos más adelante en la sección 4.4, junto con sus características principales y su utilidad.

Capítulo 3

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.

3.1. 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 3.1 (Categoría)**.** Una **categoría** $\mathscr{C}$ consiste de:

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

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

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

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

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

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

A continuación se presentan algunos ejemplos de categorías que pueden ser de utilidad para

comprender mejor el concepto.

Ejemplo 3.2 (Categoría **Set**)**.** La categoría **Set** es aquella tal que:

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

Ejemplo 3.3 (Categoría $\mathbb{1}$)**.** La categoría $\mathbb{1}$ es aquella tal que:

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

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

Definición 3.4 (Objetos iniciales y terminales)**.** Un objeto $\mathbf{0} \in \mathbf{ob}\ \mathscr{C}$ se dice **inicial** si $\forall A \in \mathbf{ob}\ \mathscr{C}$, $\exists! \mathbf{0} \to A$. Un objeto $\mathbf{1} \in \mathbf{ob}\ \mathscr{C}$ se dice **terminal** si $\forall A \in \mathbf{ob}\ \mathscr{C}$, $\exists! A \to \mathbf{1}$.

Ejemplo 3.5. 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 3.6 (Producto)**.** El **producto** de dos objetos A y B en una categoría $\mathscr{C}$ es una terna $(A \times B, \pi_A, \pi_B)$ donde:

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

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

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

Definición 3.7 (Coproducto)**.** El **coproducto** de dos objetos A, B de una categoría $\mathscr{C}$ es una terna $(A + B, \iota_A, \iota_B)$ donde:

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

Una última relación que puede establecerse entre dos objetos de una categoría es el exponencial. En **Set**, el exponencial de dos conjuntos A y B es el conjunto B^A de todas las funciones

que van de A en B, es decir que toman un elemento de A y devuelven un elemento de B. Esta noción puede generalizarse a una categoría arbitraria que tenga productos cartesianos.

Definición 3.8 (Exponencial)**.** Sea $\mathscr{C}$ una categoría con productos binarios y sean $A, B \in \mathbf{ob}\ \mathscr{C}$. Un objeto B^A es un **exponencial** si existe un morfismo $\varepsilon : B^A \times A \to B$ tal que para todo morfismo $g : C \times A \to B$ existe un único morfismo $\tilde{g} : C \to B^A$ tal que $g = \varepsilon \circ (\tilde{g} \times id_A)$.

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

Definición 3.9 (Funtor)**.** Sean $\mathscr{C}$ y $\mathscr{D}$ dos categorías. Un **funtor** $F : \mathscr{C} \to \mathscr{D}$ asigna:

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

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

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

Definición 3.10 (Transformación Natural)**.** Sean $F, G : \mathscr{C} \to \mathscr{D}$ dos funtores (entre las mismas categorías). Una **transformación natural** $\eta : F \to G$ asigna a cada $A \in \mathbf{ob}\ \mathscr{C}$ un morfismo $\eta_A : F(A) \to G(A)$ tal que para todo $f \in Hom(A, B)$ se cumple que:

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

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

Definición 3.11 (Monoide)**.** Un **monoide** es un conjunto M dotado de una operación asociativa $M \times M \to M$, $(m, n) \to mn$ tal que existe un elemento neutro:

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

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

3.2. 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.

3.2.1. Definición clásica de mónadas

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

Definición 3.12 (Mónada)**.** Dada una categoría $\mathscr{C}$, una **mónada** sobre $\mathscr{C}$ es una tupla (T, μ, η), donde:

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

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

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

Ejemplo 3.13 (Mónada *Error*)**.** Sea $T : \mathbf{Set} \to \mathbf{Set}$ el funtor $TX = X + E$, donde E es un conjunto de errores. Intuitivamente un elemento de TX puede ser un elemento de X (un valor) o un error pertenenciente a E. Luego se definen η y μ como siguen:

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

Ejemplo 3.14 (Mónada *State*)**.** Sea $T : \mathbf{Set} \to \mathbf{Set}$ el funtor $TX = (X \times S)^S$, donde S es un conjunto no vacío de estados. Intuitivamente, TX es una computación que toma un estado y retorna el valor resultante junto con el estado modificado. Luego se definen η y μ como sigue:

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

3.2.2. 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 3.15 (Sistema de extensión)**.** Un **sistema de extensión** sobre una categoría $\mathscr{C}$ es una tupla $(T, \eta, \_^*)$, donde

- $T : \mathbf{ob}\ \mathscr{C} \to \mathbf{ob}\ \mathscr{C}$,
- para cada $A \in \mathbf{ob}\ \mathscr{C}$, $\eta_A : A \to TA$,

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

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

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

Ejemplo 3.16 (Mónada *Error*)**.** Tomando el funtor descripto en la versión clásica:

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

Ejemplo 3.17 (Mónada *State*)**.** Tomando el funtor descripto en la versión clásica:

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

3.2.3. 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 3.18 (Funtor fuerte)**.** Un **funtor** $F : \mathscr{C} \to \mathscr{C}$ es **fuerte** si viene equipado con una transformación natural $\sigma_{X,Y} : FX \times Y \to F(X \times Y)$, de manera que se cumplen las siguientes ecuaciones:

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

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

Definición 3.19 (Mónada fuerte)**.** Una **mónada** (T, μ, η) sobre $\mathscr{C}$ es **fuerte** si el funtor subyacente T es fuerte y la fortaleza es compatible con μ y η:

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

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

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

Definición 3.20 (Funtor monoidal)**.** Un **funtor monoidal** es un funtor $F : \mathscr{C} \to \mathscr{C}$ equipado con una estructura monoidal (m, e), donde $m : FX \times FY \to F(X \times Y)$ es una transformación natural y $e : \mathbf{1} \to F\mathbf{1}$ es un morfismo tal que las siguientes ecuaciones se cumplen:

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

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

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

Definición 3.21 (Mónada monoidal)**.** Una **mónada monoidal** es una mónada (T, μ, η) que tiene una estructura monoidal (m, e) en su funtor subyacente T tal que $e = \eta_{\mathbf{1}}$ y las estructuras monoidal y monádica son compatibles:

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

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

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

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

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

En esta tesina se utilizará la categoría **Set**, la cual es la categoría de conjuntos y funciones, y las mónadas que se presenten serán sobre esta categoría. El objeto terminal $\mathbf{1} = \{\star\}$ es un conjunto unitario. Una consecuencia particular de esto es que cualquier funtor F y mónada (T, μ, η) sobre esta categoría son fuertes, y cada uno admite una única fortaleza posible σ ($\bar{\sigma}$).

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

3.3. Mónadas concurrentes

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

3.3.1. 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 3.22 (argumento Eckmann-Hilton)**.** *Sea X un conjunto con dos operaciones binarias $;$ y $*$ tal que $e_;$ es el elemento neutro de $;$, e_* es el elemento neutro de $*$ y la ley de intercambio $(a * b); (c * d) = (a; c) * (b; d)$ se cumple. Entonces, ambas operaciones $;$ y $*$ coinciden, y ambas son conmutativas y asociativas.*

Demostración. Primero se muestra que ambos elementos neutros coinciden:

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

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

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

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

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

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

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

3.3.2. Dos modelos

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

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

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

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

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

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

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

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

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

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

3.3.3. Monoides concurrentes

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

Definición 3.23 (Bimonoide ordenado)**.** Un **bimonoide ordenado** es un conjunto parcialmente ordenado $(A, \sqsubseteq)$ junto con dos estructuras monoidales $(A, ;, \texttt{skip})$ y $(A, *, \texttt{nothing})$ 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 3.24 (Monoide concurrente)**.** Un **monoide concurrente** es un bimonoide ordenado tal que los neutros coinciden, es decir $\texttt{nothing} = \texttt{skip}$, y la siguiente ley de intercambio se cumple:

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

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

3.3.4. Generalización a nivel de funtores y mónadas

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

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

Definición 3.25 (Funtor ordenado)**.** Un **orden** para un **funtor** F es una asignación de un orden parcial $\sqsubseteq_I$ en FI para cada conjunto I, tal que para cada morfismo $f : I \to J$, el morfismo $Ff : FI \to FJ$ es una función monótona respecto de $\sqsubseteq_I$ y $\sqsubseteq_J$.

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

Definición 3.26 (Mónada ordenada)**.** Un **orden** para una **mónada** (T, μ, η) es una asignación de un orden $\sqsubseteq_I$ sobre TI para cada conjunto I, tal que la categoría Kleisli de T se enriquece en la categoría de órdenes con el orden correspondiente a $\mathcal{K}\ell(T)(A, B)$ definido por $f \sqsubseteq_{A,B} g$ si y sólo si $\forall a : \mathbf{1} \to A, f \circ a \sqsubseteq_B g \circ a$.

Esta noción de orden sólo se relaciona con la estructura monádica. En comparación a los bimonoides ordenados, se corresponde con la condición de que ; sea compatible con $\sqsubseteq$. La compatibilidad entre $*$ y $\sqsubseteq$ se postula como sigue:

Definición 3.27 (Funtor monoidal ordenado)**.** Sea T un funtor con una estructura monoidal (m, e) y una asignación de un orden $\sqsubseteq_I$ sobre TI para cada conjunto I. Se dice que el orden es compatible con (m, e) si $v \sqsubseteq_A v'$ y $w \sqsubseteq_B w'$ implican que $m \circ \langle v, w\rangle \sqsubseteq_{A\times B} m \circ \langle v', w'\rangle$ para cada $v, v' : \mathbf{1} \to TA$ y $w, w' : \mathbf{1} \to TB$.

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

Definición 3.28 (Mónada monoidal ordenada)**.** Una mónada $(T, \eta, \_^*)$ es una mónada monoidal ordenada si está dotada de una estructura de funtor monoidal simétrico

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

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

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

Definición 3.29 (Mónada concurrente)**.** Una mónada monoidal ordenada T es una mónada concurrente si $e = \eta_{\mathbf{1}} : \mathbf{1} \to T\mathbf{1}$ y se cumple la siguiente ley de intercambio:

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

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

Ejemplo 3.30. Sea (T, μ, η) una mónada conmutativa. Entonces T tiene una única estructura de mónada monoidal $(m, \eta_{\mathbf{1}})$, y se puede definir la estructura de orden por el orden diagonal. La ley de intercambio se reduce a las condiciones de mónada monoidal.

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

Lema 3.31. *Sea (T, μ, η) una mónada monoidal ordenada (mónada concurrente). Entonces $T\mathbf{1}$ es un bimonoide ordenado (monoide concurrente).*

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

Lema 3.32. *Sea $(A, \sqsubseteq, *, \texttt{nothing}, ;, \texttt{skip})$ un bimonoide ordenado. Este puede convertirse en una mónada monoidal ordenada con funtor soporte $T_A\ X = A \times X$, operaciones y orden. Más aún, si* `nothing` $=$ `skip` *(es decir que es un monoide concurrente), entonces $e = \eta_{\mathbf{1}}$ (es decir que es una mónada concurrente).*

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

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

Ejemplo 3.33. La mónada $Tr_L(X) = \mathcal{P}_f(Trazas_{L \times X})$ es concurrente. La estructura de orden se define como la inclusión de conjuntos al igual que antes.

Capítulo 4

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* [Cap05, §3].

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.

4.1. Introducción a la coinducción

El principio de inducción está bien establecido en el área de las matemáticas y las ciencias de la computación. En esta última, se utiliza principalmente para razonar sobre tipos de datos definidos inductivamente, tales como listas finitas, árboles finitos y números naturales. La coinducción es el pincipio dual de la inducción y puede ser utilizado para razonar sobre tipos de datos definidos coinductivamente, tales como flujos de datos, trazas infinitas o árboles infinitos, pero no está tan difundido ni se comprende tan bien en general. 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 4.1 (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 $f : U^n \to U$.

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

- $B \subseteq A$

- si $a_1, ..., a_n \in A$ y $f(a_1, ..., a_n) = a$, 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 4.2 (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 $f(a_1, ..., a_{ar(f)}) = a$ y $P(a_i)$ se cumple para cada $i = 1, ..., ar(f)$, 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 4.3 (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 $f : U^n \to U$.

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

- $B \subseteq A$
- si $a_1, ..., a_n \in A$ y $f(a_1, ..., a_n) = a$, 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 4.4 (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 $f(a_1, ..., a_{ar(f)}) = a$ y $P(a_i)$ se cumple para cada $i = 1, ..., ar(f)$,*

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:

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

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

$$\texttt{Lista de } A = \texttt{nil} + A \times \texttt{Lista de } A \tag{4.1}$$

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 4.1. 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 4.5 (Álgebra de un funtor)**.** Dado un endofuntor F sobre una categoría $\mathscr{C}$, un **álgebra** de F es un objeto X de $\mathscr{C}$ junto con un morfismo $\alpha : FX \to X$. Dadas dos álgebras $(X, \alpha : FX \to X)$, $(Y, \beta : FY \to Y)$ de F, $m : X \to Y$ es un morfismo de álgebras si se cumple la siguiente ecuación:

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

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

Definición 4.6 (Coálgebra de un funtor)**.** Una **coálgebra** para un endofuntor F sobre una categoría $\mathscr{C}$ es un objeto A junto con un morfismo $u : A \to FA$. Dadas dos coálgebras $(A, \eta : A \to FA)$, $\quad (B, \theta : B \to FB)$, $f : A \to B$ es un morfismo de coálgebras si respeta la

estructura coalgebraica:

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

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

Definición 4.7 (Álgebra inicial)**.** Un **álgebra inicial** para un endofuntor F sobre una categoría $\mathscr{C}$ es un objeto inicial en la categoría de las F-álgebras.

Definición 4.8 (Coálgebra final)**.** Una **coálgebra final** para un endofuntor F sobre una categoría $\mathscr{C}$ es un objeto terminal en la categoría de las F-coálgebras.

Formalmente, 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 endofuntor dado en la categoría **Set**.

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

Ejemplo 4.9 (Flujo de datos infinitos)**.** El conjunto A^ω de flujos de datos infinitos sobre un alfabeto A es (el conjunto soporte de) la coálgebra final del funtor $FX = A \times X$.

Ejemplo 4.10 (Cadenas infinitas)**.** El conjunto A^∞ de las cadenas finitas e infinitas sobre un alfabeto A es (el conjunto soporte de) la coálgebra final del funtor $FX = \mathbb{1} + A \times X$.

Mientras que los tipos inductivos se definen mediante sus constructores, los tipos coinductivos usualmente se presentan junto con sus destructores. Por ejemplo, los flujos de datos o *streams* admiten dos operaciones $hd : A^\omega \to A$ y $tl : A^\omega \to A^\omega$, los cuales representan la función *head* que devuelve el primer elemento del *stream* y la función *tail* que devuelve la cola del *stream*. La existencia de los destructores es una consecuencia del hecho de que A^ω es una coálgebra para el funtor $FX = A \times X$. Todas estas coálgebras vienen equipadas con una función estructural $\langle obs, cont\rangle : X \to A \times X$; para A^ω se tiene que $obs = hd$ y $cont = tl$.

4.2. Los 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 $S : \mathbb{N} \to \mathbb{N}$ 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 $s : \mathcal{A} \to \mathcal{A}$. Estas estructuras se denominan estructuras de Peano. Para cada una de ellas existe una única función $f : \mathbb{N} \to \mathcal{A}$ 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 coálgebra final en la categoría de las coálgebras 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 $\mathrm{Dom}(P) = \{n \mid n > 0\}$. 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 $is0 : \{0\} \to \{\star\}$, por lo que necesariamente $is0(0) = \star$ y $\mathrm{Dom}(is0) = \{0\}$.

El dual de un álgebra de Peano $(\mathcal{A}, z, s)$ es una coálgebra de Peano $(\mathcal{C}, isz, p)$, donde isz y p son destructores: $isz : \mathcal{C} \nrightarrow \{\star\}$ y $p : \mathcal{C} \nrightarrow \mathcal{C}$. Si una coálgebra 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 $x \in \mathrm{Dom}(isz)$ o bien $x \in \mathrm{Dom}(p)$, pero no ambos. Los números conaturales son la coálgebra de Peano $(\bar{\mathbb{N}}, is0, P)$, con la propiedad de que para cualquier coálgebra de Peano $(\mathcal{C}, isz, p)$ hay una única función $g : \mathcal{C} \to \bar{\mathbb{N}}$ tal que para cada $x \in \mathcal{C}$:

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

La coálgebra $(\bar{\mathbb{N}}, is0, P)$ es entonces una coálgebra terminal o final en la categoría de las coálgebras 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 ∞ tal que $P(\infty) = \infty$. El conjunto soporte $\bar{\mathbb{N}}$ sería entonces de alguna manera equivalente a $\mathbb{N} \cup \{\infty\}$.

4.3. La 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 4.11 (*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:

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

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

Definición 4.12 (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:

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

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

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

Definición 4.14 (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:

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

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

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

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

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

4.4. 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.

4.4.1. 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 CoℕN : Set where
  zero : CoℕN
  suc : ∞ CoℕN → CoℕN
```

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

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

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

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

```
inf : CoℕN
inf = suc (♯ inf)
```

Como aproximación a la productividad, en el chequeo de terminación se pide que en la definición de funciones corecursivas las llamadas recursivas aparezcan bajo la aplicación directa de un construcor coinductivo. Esta restricción en general hace que programar con tipos coinductivos sea incómodo, y es por eso que se buscan técnicas alternativas para asegurar que las definiciones corecursivas estén bien definidas.

4.4.2. *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 : 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 : Size< i y k : Size< j, entonces k : Size< i. La relación de orden es, además, bien fundada, lo cual se usa para definir funciones corecursivas productivas. Existe también una operación sucesor de tamaños ↑ y un tamaño "infinito" ∞ tal que para cada tamaño i, i : Size< ∞. Finalmente, un *sized type* es un tipo indexado por Size.

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

```
mutual

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

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

open Conat′ public
```

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

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

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

Parte II

Formalización de Mónadas Concurrentes en Agda: el Caso de la Mónada Delay

Capítulo 5

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.

5.1. Formalización en Agda: los monoides

Como se introdujo en la definición 3.11, 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 zero_m) ≅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 $\_\cong_m\_$, 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 eq_m es una prueba de que la relación $\_\cong_m\_$ 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 4.3, 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, zero_m, 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 zero_m 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 $(\text{zero}_m +_m x) \cong_m x$. El siguiente, idr, es análogo a idl y representa la prueba de que zero_m 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 $(x +_m (y +_m z)) \cong_m ((x +_m y) +_m z)$. Estas pruebas se dan en torno a la noción de igualdad introducida en el primer campo.

5.2. 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 3.29,

estas son mónadas cuyo funtor subyacente tiene una estructura monoidal.

5.2.1. 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 $M : \mathsf{Set} \to \mathsf{Set}$.

```
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 $\eta : A \to MA$. 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 3.15 se define el operador $\_^*$ que toma una función de tipo $A \to MB$ y devuelve una funciónń de tipo $MA \to MB$. El tipo del operador es por tanto $\_^* : (A \to MB) \to MA \to MB$. Si se da vuelta el orden de sus argumentos se obtiene el tipo de la función *bind*: $\_>>=\_ : M\ A \to (A \to M\ B) \to M\ B$.

Los últimos tres campos representan las leyes de mónadas. Estos están dados por tipos que representan proposiciones. El primero de ellos, law_1, 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_1 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, law_2 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 η_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, law_3, 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 $f\ x \gg= g$. Esta última ley se corresponde con la ecuación que indica que $g^* \circ f^* = (g^* \circ f)^*$.

5.2.2. Formalización de funtor monoidal

Como se indica en la definición 3.20, 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 $M : \mathsf{Set} \to \mathsf{Set}$.

```
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 $m : MA \times MB \to M(A \times B)$ está dada por el

campo merge. A diferencia de como se describe en la definición formal (3.20), 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 currificada.

El otro ingrediente que se necesita para dar la estructura monoidal del funtor es el morfismo $e : \mathbf{1} \to M\mathbf{1}$. En la sección 3.2.3 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 morfistmo e con una función de tipo $\top \to M\ \top$, 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 MonoidalFunctor es fmap. Como se describió en la definición 3.9, un funtor F de $\mathscr{C}$ en $\mathscr{D}$ no sólo asigna un objeto de $\mathscr{D}$ a cada objeto de $\mathscr{C}$, sino que también asigna a cada morfismo $f : A \to B \in \mathbf{mor}\ \mathscr{C}$ un morfismo $F(f) : FA \to FB$. 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.

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

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 5.1. Si luego se reemplaza F por M, $\mathbf{1}$ por $\top$ y $m_{A,\mathbf{1}}$ por merge, se obtiene el diagrama 5.2.

$$\begin{array}{ccc} & & FA \times F\mathbf{1} \\ & \swarrow^{\pi_1} & \downarrow{\scriptstyle m_{A,\mathbf{1}}} \\ FA & \xleftarrow[F(\pi_1)]{} & F(A \times \mathbf{1}) \end{array} \qquad (5.1) \qquad \begin{array}{ccc} & & MA \times M\ \top \\ & \swarrow^{\pi_1} & \downarrow{\scriptstyle \text{merge}} \\ MA & \xleftarrow[M(\pi_1)]{} & M(A \times \top) \end{array} \qquad (5.2)$$

El segundo diagrama da lugar a la ecuación: $M(\pi_1) \circ \text{merge} \cong_m \pi_1$. Si en lugar de utilizar π_1 se considera su inversa π_1^{-1}, el diagrama y su ecuación correspondiente pasan a quedar como sigue:

$$\begin{array}{ccc} & & MA \times M\ \top \\ & \nearrow^{\pi_1^{-1}} & \downarrow{\scriptstyle \text{merge}} \\ MA & \xrightarrow[M(\pi_1^{-1})]{} & M(A \times \top) \end{array} \qquad (5.3) \qquad \text{merge} \circ \pi_1^{-1} \ \cong_m \ M(\pi_1^{-1}) \qquad (5.4)$$

Ahora, como en realidad merge no toma un producto cartesiano sino ambos elementos por separado, no es necesario utilizar π_1^{-1} antes de merge. Si se considera por último que π_1^{-1} se representa en Agda como ($\lambda\ a \rightarrow\ (a\ ,\ \mathsf{tt})$) y que la asignación de morfismos del funtor M está dada por el campo fmap, se obtiene finalmente la ecuación merge $\cong_m$ fmap ($\lambda\ a \rightarrow\ (a\ ,\ \mathsf{tt})$), la cual al agregarle los argumentos correspondientes forma la ecuación del campo idr:

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

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 α representa la asociatividad del producto cartesiano.

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

α 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 α 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:

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

En la definición formal hay una condición extra para los funtores monoidales que dice que si la estructura monoidal es además compatible con γ, 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 γ. Agregando este último campo se obtiene entonces la formalización de los funtores monoidales simétricos.

5.3. 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 3.23 se presenta a un bimonoide ordenado como un conjunto parcialmente ordenado $(A, \sqsubseteq)$ junto con dos estructuras monoidales $(A, ;, \texttt{skip})$ y $(A, *, \texttt{nothing})$ tales que ; y $*$ son compatibles con $\sqsubseteq$ y $*$ es conmutativa. Luego, en la definición 3.24 se define a un monoide concurrente como un bimonoide 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 $x \lesssim_m y \Leftrightarrow y \sqsubseteq x$. 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 $\_\cong_m\_$. La prueba porder_m de que la relación $\_\lesssim_m\_$ 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 zero_m, representan la primera estructura monoidal del bimonoide 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: sidl y sidr son las pruebas de que zero_m es neutro de la operación $\_+_m\_$, a izquierda y derecha respectivamente, y 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: max_m. Esta operación junto con 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 max_m como una versión simplificada de la operación 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 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: midl y midr representan pruebas de que zero_m es neutro respecto de max_m, a izquierda y derecha respectivamente, y massoc representa la prueba de que dicha operación es asociativa. Se antepone la m delante para indicar que estas propiedades pertenecen a 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 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 ichange se obtiene de forma casi directa de la ecuación de la ley de intercambio introducida en la definición 3.24: $(x * z); (y * w) \sqsubseteq (x; y) * (z; w)$. Si se reemplazan los operadores $*$ y ; por max_m y $+_m$, respectivamente, y se invierte el orden de la desigualdad como se explicó previamente, se llega a la siguiente ecuación: $(\text{max}_m\ (x +_m y)\ (z +_m w)) \lesssim_m ((\text{max}_m\ x\ z) +_m (\text{max}_m\ y\ w))$.

5.4. 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 record parametrizado. Como en el caso de la formalización de mónadas, el parámetro del record ConcurrentMonad será una asignación $M : \text{Set} \to \text{Set}$, la cual representa la asignación de objetos a objetos de un funtor $M : \mathbf{Set} \to \mathbf{Set}$.

```
record ConcurrentMonad (M : Set → Set) : Set₁ where
  constructor
    makeConcurrentMonad
  field
    _≅ₘ_      : ∀ {A} → M A → M A → Set
    eqₘ       : ∀ {A} → IsEquivalence (_≅ₘ_ {A})
    _≲ₘ_      : ∀ {A} → M A → M A → Set
    porderₘ   : ∀ {A} → IsPartialOrder (_≅ₘ_ {A}) (_≲ₘ_ {A})
    return    : ∀ {A : Set} → A → M A
    _>>=_     : ∀ {A B : Set} → M B → (B → M A) → M A
    bcomp≲ₘ   : ∀ {A B : Set} → (a₁ a₂ : M A) → (f₁ f₂ : A → M B) → a₁ ≲ₘ a₂
                → (∀ (a : A) → (f₁ a) ≲ₘ (f₂ a)) → (a₁ >>= f₁) ≲ₘ (a₂ >>= f₂)
```

```
    monad₁     : ∀ {A B : Set} → (x : B) (f : B → M A)
                               → (((return x) >>= f) ≅ₘ (f x))
    monad₂     : ∀ {A} → (t : M A) → (t >>= return) ≅ₘ t
    monad₃     : ∀ {A B C : Set} → (t : M C) (f : C → M B) (g : B → M A)
                               → ((t >>= f) >>= g) ≅ₘ (t >>= (λ x → f x >>= g))

  unit : M ⊤
  unit = return tt

  field
    merge      : ∀ {A B : Set} → M A → M B → M (A × B)
    mcomp≲ₘ    : ∀ {A B : Set} → (a₁ a₂ : M A) → (b₁ b₂ : M B) → a₁ ≲ₘ a₂ → b₁ ≲ₘ b₂
                 → (merge a₁ b₁) ≲ₘ (merge a₂ b₂)
    idr        : ∀ {A : Set} → (a : M A)
                 → (merge a unit) ≅ₘ (a >>= (λ a → return (a , tt)))
    idl        : ∀ {B : Set} → (b : M B)
                 → (merge unit b) ≅ₘ (b >>= (λ b → return (tt , b)))
    assoc      : ∀ {A B C : Set} → (a : M A) (b : M B) (c : M C)
                 → ((merge (merge a b) c) >>= (λ {((a , b) , c) → return (a , (b , c))}))
                                                              ≅ₘ (merge a (merge b c))
    comm       : ∀ {A B : Set} → (a : M A) (b : M B)
                 → (merge a b) ≅ₘ ((merge b a) >>= (λ {(a , b) → return (b , a)}))
    ichange    : ∀ {A B C D : Set} → (a : M A) (b : M B) (f : A → M C) (g : B → M D)
                 → (merge (a >>= f) (b >>= g))
                                  ≲ₘ ((merge a b) >>= (λ { (a , b) → (merge (f a) (g b)) }))

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 3.29 se expone que para que una mónada sea concurrente debe ser primero una mónada monoidal ordenada. La definición 3.28, 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 3.15). 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 $\gg=$ sirve para secuenciar dos computaciones.

Antes de pasar a las leyes de las mónadas, se introduce la compatibilidad del operador $\gg=$ con la relación de orden. Esta propiedad está dada por el campo $\mathsf{bcomp}_{\lesssim_m}$. En la definición 3.26, 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 $A \to TB$ con un orden $\sqsubseteq_{A,B}$ tal que $f \sqsubseteq_{A,B} g$ si y sólo si $\forall a : \mathbf{1} \to A, f \circ a \sqsubseteq_B g \circ a$. 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 $a : \mathbf{1} \to A$, se toman directamente valores $a : A$. Se considera entonces que una función $f_1 : A \to M\ B$ es menor o igual a una función $f_2 : A \to M\ B$ 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 $M\ A$ y dos funciones f_1 y f_2 de tipo $A \to M\ B$, si se tiene que $a_1 \lesssim_m a_2$ y $\forall\ (a : A) \to (f_1\ a) \lesssim_m (f_2\ a)$, entonces $(a_1 \gg= f_1) \lesssim_m (a_2 \gg= f_2)$.

Los campos monad_1, monad_2 y monad_3 representan las tres leyes de mónadas y son iguales a los campos law_1, law_2 y law_3 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 3.28 para que la mónada sea una mónada monoidal ordenada. El elemento neutro es un morfismo $e : \mathbf{1} \to M\mathbf{1}$. 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 habitante unit del tipo $M\ \top$ en lugar de como una función de tipo $\top \to M\ \top$ ya que hay un único argumento posible para la función. Según la definición 3.29, para que la mónada sea una mónada concurrente, el morfismo e debe ser igual a $\eta_{\mathbf{1}} : \mathbf{1} \to M\mathbf{1}$. Considerando $\mathbf{1} = \top$, luego e debería ser igual a return $\{\top\} : \top \to M\ \top$. Como $\top$ tiene un único habitante, se obtiene que unit debe ser igual a return tt : $M\ \top$. Como ya queda definido el valor de unit, no se define como un campo sino como una constante.

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 $f : A \to B$ una función, luego $(\lambda\ ma \to ma \gg= (\lambda\ a \to \mathsf{return}\ f\ a))$ es una función de tipo $M\ A \to M\ B$. De esta manera la asignación de morfismos a morfismos del funtor queda definida.

El campo que sigue, $\mathsf{mcomp}_{\lesssim_m}$, representa la compatibilidad del operador merge con la relación de orden $\lesssim_m$. Dados a_1 y a_2 de tipo $M\ A$ y b_1 y b_2 de tipo $M\ B$, si se tiene que $a_1 \lesssim_m a_2$ y $b_1 \lesssim_m b_2$, luego se debe cumplir que $(\mathsf{merge}\ a_1\ b_1) \lesssim_m (\mathsf{merge}\ a_2\ b_2)$.

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 $\forall\ \{A : \mathsf{Set}\} \to (a : M\ A) \to (\mathsf{merge}\ \mathsf{unit}\ a) \cong_m (\mathsf{fmap}\ (\lambda\ a \to (a\ ,\ \mathsf{tt}))\ a)$. Reemplazando fmap $(\lambda\ a \to (a\ ,\ \mathsf{tt}))$ por la función propuesta anteriormente, donde se sustituye f por $(\lambda\ a \to (a\ ,\ \mathsf{tt}))$, quedaría el siguiente tipo:

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

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

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

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

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

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:

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

Reemplazando fmap $(\lambda\ \{((a\ ,\ b)\ ,\ c) \to (a\ ,\ (b\ ,\ c))\})$ por la función equivalente a fmap, donde la f se sustituye por la función $(\lambda\ \{((a\ ,\ b)\ ,\ c) \to (a\ ,\ (b\ ,\ c))\})$, el tipo queda como sigue:

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

La aplicación $(\lambda\ \{((a\ ,\ b)\ ,\ c) \to (a\ ,\ (b\ ,\ c))\})\ a))$ 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:

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

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

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

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 demostrarse 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 $(\mathsf{merge}\ a\ b)$, y del otro se aplica $(\mathsf{merge}\ b\ a)$ 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 $a' : \mathbf{1} \to MA$, $b' : \mathbf{1} \to MB$, $f : A \to MC$ y $g : B \to MD$ (pueden tomarse a' y b' como funciones que salen del conjunto $\mathbf{1}$ debido a las características de la categroría **Set**). En la definición 3.29, 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:

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

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:

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

La aplicación (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: $(\lambda\ \{\ (a\ ,\ b) \to (\mathsf{merge}\ (f\ a)\ (g\ b))\ \}\ )$. 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:

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

Capítulo 6

El caso de la Mónada Delay

Como se mencionó en el capítulo 4, 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 4.4.1. 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.

6.1. 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 : Set, el tipo $A\ \bot$ representa el tipo $\mathbf{D}A$ definido en la sección 4.3.

```
data _⊥ (A : Set) : Set where
  now : (x : A) → A ⊥
  later : (x : ∞ (A ⊥)) → A ⊥
```

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

Un habitante del tipo $A\ \bot$ 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 _⊥ 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 = geq, o la bisemejanza débil si k = weak. La igualdad entre tipos de igualdad, es decir entre valores del tipo Kind, es decidible. El operador $\overset{?}{=}$-Kind toma dos tipos de igualdad y decide si son iguales o no, dando a su vez una prueba de ello.

```
infix 4 _≟-Kind_

_≟-Kind_ : Decidable (_≡_ {A = Kind})
_≟-Kind_ strong       strong       = yes P.refl
_≟-Kind_ strong       (other k)    = no λ()
_≟-Kind_ (other k)    strong       = no λ()
_≟-Kind_ (other geq)  (other geq)  = yes P.refl
_≟-Kind_ (other geq)  (other weak) = no λ()
_≟-Kind_ (other weak) (other geq)  = no λ()
_≟-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 ≟-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 : Set, que será el tipo de retorno, y una relación _∼_ : $A \to A \to$ Set que establece una noción de igualdad entre valores de tipo A.

```
module Equality {A : Set} (_∼_ : 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 (♭ x) (♭ y)))    → Rel k (later x) (later y)
  later′ : ∀ {x y}  (x≈y : Rel (other weak) x (♭ y)) → Rel (other weak) x (later y)
  later″ : ∀ {k x y} (x∼y : Rel (other k) (♭ 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 $A\ \bot$, 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 $A\ \bot$ 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 $\infty\ A\ \bot$ (valores de tipo $A\ \bot$ posiblemente infinitos). Este constructor pide una prueba (posiblemente infinita) de que ($\flat\ x$) y ($\flat\ y$) están relacionados por la relación k y afirma que entonces (later x) y (later y) están relacionados por la misma relación k.
- El constructor later′ se puede utilizar únicamente para bisemejanza débil. Dados $x : A\ \bot$ e $y : \infty\ A\ \bot$ que son parámetros implícitos y una prueba de que x y ($\flat\ y$) son débilmente bisemejantes, este constructor afirma que entonces x y (later y) son débilmente bisemejantes también. Intuitivamente, later′ permite agregar un constructor later en el lado derecho de una bisemejanza débil.
- El constructor later″ es análogo a later′ 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 $\cong$, $\gtrsim$, $\lesssim$ 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 _≳_
```

```
_≈_ : A ⊥ → A ⊥ → Set _
_≈_ = Rel (other weak)
```

A continuación, se definen otros operadores que tienen que ver con la noción de convergencia introducida en la definición 4.13.

```
infix 4 _⇓[_]_ _⇓_

_⇓[_]_ : A ⊥ → Kind → A → Set _
x ⇓[ k ] y = Rel k x (now y)

_⇓_ : A ⊥ → A → Set _
x ⇓ y = x ⇓[ other weak ] y

infix 4 _⇓

_⇓ : A ⊥ → Set _
x ⇓ = ∃ λ v → x ⇓ v

infix 4 _⇑[_] _⇑

_⇑[_] : A ⊥ → Kind → Set _
x ⇑[ k ] = Rel k x never

_⇑ : A ⊥ → Set _
x ⇑ = x ⇑[ other weak ]
```

$x \Downarrow[\ k\ ]\ y$ indica que x se relaciona mediante la relación k con (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 (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.

```
module _ {A : Set} {_∼_ : A → A → Set} where

  open Equality _∼_ using (Rel; _≅_; _≳_; _≲_; _≈_; _⇓[_]_; _⇑[_])
  open Equality.Rel

  ≅⇒ : ∀ {k} {x y : A ⊥} → x ≅ y → Rel k x y
  ≅⇒ (now x∼y)  = now x∼y
  ≅⇒ (later x≅y) = later (♯ ≅⇒ (♭ x≅y))

  ≳⇒ : ∀ {k} {x y : A ⊥} → x ≳ y → Rel (other k) x y
  ≳⇒ (now x∼y)    = now x∼y
```

```
≳⇒ (later x≳y)    = later (♯ ≳⇒ (♭ x≳y))
≳⇒ (later^l x≳y)  = later^l (≳⇒ x≳y )

⇒≈ : ∀ {k} {x y : A ⊥} → Rel k x y → x ≈ y
⇒≈ {strong}       = ≅⇒
⇒≈ {other geq}    = ≳⇒
⇒≈ {other weak}   = id
```

El lema $\cong\Rightarrow$ demuestra que todas las relaciones incluyen a la bisemejanza fuerte, es decir que si dos términos $x\ y : A\ \bot$ son fuertemente bisemejantes, entonces también son débilmente bisemejantes y, además, $x \gtrsim y$. De manera similar, el segundo lema, $\gtrsim\Rightarrow$, postula que la desigualdad está incluída 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 incluídas 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 later^r, later^l y 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.

```
later^r-1 : ∀ {k} {x : A ⊥} {y} →
          Rel (other k) x (later y) → Rel (other k) x (♭ y)
later^r-1 (later x∼y)   = later^l (♭ x∼y)
later^r-1 (later^r x≈y) = x≈y
later^r-1 (later^l x∼ly) = later^l (later^r-1 x∼ly)

later^l-1 : ∀ {x} {y : A ⊥} → later x ≈ y → ♭ x ≈ y
later^l-1 (later x≈y)   = later^r (♭ x≈y)
later^l-1 (later^r lx≈y) = later^r (later^l-1 lx≈y)
later^l-1 (later^l x≈y) = x≈y

later^-1 : ∀ {k} {x y : ∞ (A ⊥)} →
          Rel k (later x) (later y) → Rel k (♭ x) (♭ y)
later^-1 (later x∼y)   = ♭ x∼y
later^-1 (later^r lx≈y) = later^l-1 lx≈y
later^-1 (later^l x∼ly) = later^r-1 x∼ly
```

La función later^{r-1} permite sacar un constructor later del lado derecho de una ecuación que tenga como operador a la desigualdad o la bisemejanza débil. Si se tiene Rel (other k) x (later y), para algún tipo k : OtherKind y algún par de términos $x : A\ \bot$ e $y : \infty\ (A\ \bot)$, entonces se tiene que x se relaciona por el mismo tipo de relación k con ($\flat\ y$). La función later^{l-1}, de manera análoga, permite quitar un constructor later del lado izquierdo de la ecuación, pero sólo para el caso de la bisemejanza débil. Esto implica que si se tiene later $x \approx y$, entonces también se cumple que $\flat\ x \approx y$. Por último, la función later^{-1} permite quitar el constructor later de ambos lados de la ecuación. Esto puede realizarse para cualquier tipo de relación k : 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 _∼_ 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 : Kind. Se pide como condición una prueba de que la relación subyacente _∼_ 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 _∼_ y devuelve el tipo correspondiente a la prueba de que dicha relación es reflexiva, es decir $\forall\ \{x\} \to x \sim x$. 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 (♭ x∼y))
sym sym-∼ eq (later^r x≈y)      = later^l (sym sym-∼ eq x≈y )
sym sym-∼ eq (later^l {weak} x≈y) = later^r (sym sym-∼ eq x≈y )
```

La prueba de simetría requiere, además de la prueba de que la relación subyacente _∼_ es simétrica, una prueba de que el tipo de relación k : 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 (later^l {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-∼ () (later^l {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 *trans-∼* 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 (later^l x∼y) y∼z       = later^l (now-trans x∼y y∼z)
      now-trans x∼ly        (later^l y∼z) = now-trans (later^r-1 x∼ly) y∼z

      mutual

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

```
                  Rel k x y → Rel k y (later z) → Rel k x (later z)
later-trans (later x∼y)  ly∼lz        = later (♯ trans (♭ x∼y) (later⁻¹ ly∼lz))
later-trans (laterˡ x∼y) y∼lz         = later (♯ trans x∼y (laterʳ⁻¹ y∼lz))
later-trans (laterʳ x≈y) ly≈lz        = later-trans x≈y (laterˡ⁻¹ ly≈lz)
later-trans x≈y          (laterʳ y≈z) = laterʳ ( trans x≈y          y≈z )

trans : ∀ {k} {x y z : A ⊥} → Rel k x y → Rel k y z → Rel k x z
trans {z = now v}   x∼y y∼v  = now-trans x∼y y∼v
trans {z = later z} x∼y y∼lz = later-trans x∼y y∼lz
```

```
open Trans public using (trans)
```

La transitividad postula que, dados tres términos x, y y z, si se tienen las relaciones Rel k x y y Rel k y z, entonces se debe tener también la relación Rel k x z. Para demostrar esto, se necesitan dos pruebas auxiliares: now-trans que prueba esto asumiendo $z =$ (now v) para algún $v : A$ y later-trans que hace lo mismo para el caso en que $z =$ (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 $x\ y : A\ \bot$, si se tiene que $x \gtrsim y$ y $x \lesssim y$, luego $x \cong y$, es decir que x e y son fuertemente bisemejantes.

```
antisym : {x y : A ⊥} → x ≳ y → x ≲ y → x ≅ y
antisym (now x∼y)   (now _)      = now x∼y
antisym (later x≳y) (later x≲y)  = later (♯ antisym (♭ x≳y) (♭ x≲y))
antisym (later x≳y) (laterˡ x≲ly) = later (♯ antisym (♭ x≳y) (laterʳ⁻¹ x≲ly))
antisym (laterˡ x≳ly) (later x≲y)  = later (♯ antisym (laterʳ⁻¹ x≳ly) (♭ x≲y))
antisym (laterˡ x≳ly) (laterˡ x≲ly) = later (♯ antisym (laterʳ⁻¹ x≳ly) (laterʳ⁻¹ x≲ly))
```

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.

6.2. 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 5.2.1, cuyo parámetro será el funtor _⊥. 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:

```
bind : A ⊥ → (A → B ⊥) → B ⊥
bind (now x) f = f x
bind (later x) f = later (♯ (bind (♭ x) f))
```

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

término t de tipo $A\ \bot$ y una función f de tipo $A \to B\ \bot$. 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 $\_\sim\_ : \forall\ \{A\} \to A \to A \to$ 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 (_∼_ : ∀ {A} → A → A → Set)
            (eq∼ : ∀ {A} → IsEquivalence (_∼_ {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 → A → Set} (refl∼ : Reflexive _∼_) where

    open Equality _∼_ using (_≈_)
    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 → A ⊥) → bind (now x) f ≈ f x
    left-identity x f = refl refl∼
```

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 ⊥) → bind t now ≈ t
    right-identity (now x) = refl refl∼
    right-identity (later x) = later (♯ (right-identity (♭ x)))
```

En este caso se prueba que el constructor now es neutro a derecha de bind, es decir que para un término $t : A\ \bot$, 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 (♭ 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 $eq\sim$.

```
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 A.1.

6.3. 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 $M\,A \to M\,B \to M(A \times B)$. En este caso particular, como el funtor M está dado por el operador de tipos $\_\,\bot$, la operación merge tendrá el siguiente tipo: $A\ \bot \to B\ \bot \to (A \times B)\ \bot$.

```
merge : A ⊥ → B ⊥ → (A × B) ⊥
merge (now a) (now b)   = now (a , b)
merge (now a) (later b) = later (♯ (merge (now a) (♭ b)))
merge (later a) (now b) = later (♯ (merge (♭ a) (now b)))
merge (later a) (later b) = later (♯ (merge (♭ a) (♭ 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 5.2.2, la asignación de morfismos del funtor que en este caso está dado por $\_\,\bot$. Este operador, por lo tanto, asignará, para cada función $A \to B$, una función $A\ \bot \to B\ \bot$.

```
fmap : ∀ {A B : Set} → (A → B) → (A ⊥ → B ⊥)
fmap f (now x)   = now (f x)
fmap f (later x) = later (♯ (fmap f (♭ x)))
```

Dada la función $f : A \to B$, si el elemento de tipo $A\ \bot$ recibido es de la forma (now x), entonces el resultado de aplicar fmap f a tal elemento es un término construído 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 (♭ 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 $\top\ \bot$. Se define entonces unit como el elemento más simple de tal tipo:

```
unit : ⊤ ⊥
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 (_∼_ : ∀ {A} → A → A → Set)
            (eq∼ : ∀ {A} → IsEquivalence (_∼_ {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 _≈_. Luego se abre el tipo Rel de manera que puedan usarse sus constructores sin prefijo.

```
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)   (♭ c)))
  associative (now a)   (later b) (now c)   = later (♯ (associative (now a)   (♭ b)   (now c)))
  associative (now a)   (later b) (later c) = later (♯ (associative (now a)   (♭ b)   (♭ c)))
  associative (later a) (now b)   (now c)   = later (♯ (associative (♭ a)   (now b)   (now c)))
  associative (later a) (now b)   (later c) = later (♯ (associative (♭ a)   (now b)   (♭ c)))
  associative (later a) (later b) (now c)   = later (♯ (associative (♭ a)   (♭ b)   (now c)))
  associative (later a) (later b) (later c) = later (♯ (associative (♭ a)   (♭ b)   (♭ c)))
```

La prueba no presenta mayores dificultades. En el caso base, donde los tres términos son de la forma (now x), las aplicación (merge (merge (now a) (now b)) (now c)) genera el siguiente término: (now ((a , b) , c)). Al aplicar fmap ($\lambda\{((a , b) , c) \to (a , (b , c))\}$) a este resultado, se obtiene el término (now (a , (b , c))), el cual es también el resultado de la aplicación del lado derecho de la ecuación: (merge (now a) (merge (now b) (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 (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 × ⊤) → (A × ⊤) → Set}
         (reflA×⊤ : Reflexive _∼_) where
```

```
  open Equality {A × ⊤} _∼_ using (_≈_)
  open Equality.Rel

  rid : (a : A ⊥) → (merge a unit) ≈ (fmap (λ a → (a , tt)) a)
  rid (now x)   = now reflA×⊤
  rid (later x) = later (♯ (rid (♭ x)))
```

Para el caso en el que a tiene la forma (now x), la aplicación (merge (now x) unit) tiene como resultado el término (now (x , tt)). Aplicar fmap (lambda $a \to$ (a , tt)) a (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 merge es análoga a la anterior, donde el módulo anónimo en el cual se encapsula es análogo también:

```
module _ {A : Set} {_∼_ : (⊤ × A) → (⊤ × A) → Set}
         (refl⊤×A : Reflexive _∼_) where

  open Equality {⊤ × A} _∼_ using (_≈_)
  open Equality.Rel

  lid : (a : A ⊥) → (merge unit a) ≈ (fmap (λ a → (tt , a)) a)
  lid (now x)   = now refl⊤×A
  lid (later x) = later (♯ (lid (♭ 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 construída 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 _⊥ 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 A.1.

6.4. ¿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 6.1, 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 utlizarán 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. Es preciso notar que la definición de unit dada para la instancia de funtor monoidal se puede reutilizar en esta sección debido a que coincide con la expresión return tt, puesto que unit no es un campo a definir sino que es una constante del record ConcurrentMonad.

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 (_≡_) 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 6.2, 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 (♭ 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 (♭ 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 later^l como $\lesssim$later^l.

```
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 (later^l to ≲later^l)

  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 (≲later^l a₁≲a₂) f₁≲f₂ =
                                ≲later^l (bind-comp (now a₁) (♭ a₂) f₁ f₂ a₁≲a₂ f₁≲f₂)
  bind-comp (later a₁) (later a₂) f₁ f₂ (later a₁≲a₂) f₁≲f₂ =
                                later (♯ (bind-comp (♭ a₁) (♭ a₂) f₁ f₂ (♭ a₁≲a₂) f₁≲f₂))
  bind-comp (later a₁) (later a₂) f₁ f₂ (≲later^l a₁≲a₂) f₁≲f₂ =
                                later (♯ (bind-comp (♭ a₁) (♭ a₂) f₁ f₂ (later^r-1 a₁≲a₂) f₁≲f₂))
```

La prueba toma dos valores a_1 y a_2 de tipo $A\ \bot$, dos funciones f_1 y f_2 de tipo $A \to B\ \bot$, una prueba de que $a_1 \lesssim\mathsf{A}\ a_2$ y una prueba de que para cualquier $a : A$ se cumple que $f_1\ a \lesssim\mathsf{B}\ f_2\ a$, es decir que $f_1 \lesssim f_2$.

En el caso base, donde ambos valores son de la forma (now x), al hacer *pattern matching* sobre la prueba $a_1 \lesssim a_2$, se obtiene que está dada por el término (now prefl), lo cual hace que, obligatoriamente, el valor de retorno del segundo argumento sea igual al del primero. El objetivo es ver que bind (now a_1) $f_1 \lesssim\mathsf{B}$ bind (now a_2) f_2, lo cual por definición es lo mismo que escribir: $f_1\ a_1 \lesssim\mathsf{B}\ f_2\ a_2$. 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 prefl, ambos valores de retorno son a_1 y el objetivo es probar que $f_1\ a_1 \lesssim\mathsf{B}\ f_2\ a_1$, 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 (now a_1), y el segundo, (later a_2). La prueba de que el primero es menor o igual al segundo está dada por el constructor $\lesssim\mathsf{later}^l$ junto con una prueba $a_1 \lesssim a_2$ de tipo (now a_1) $\lesssim\mathsf{B}$ ($\flat\ a_2$). 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 $\lesssim\mathsf{later}^l$ hace referencia en realidad al término (later a_2) $\gtrsim\mathsf{B}$ (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 (later x). No podría darse el caso de que el primero esté construído con later y el segundo se construya con 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 later con una prueba $a_1 \lesssim a_2$ de tipo ∞ (($\flat\ a_1$) $\lesssim\mathsf{B}$ ($\flat\ a_2$)). 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 ($\flat\ a_1 \lesssim a_2$). Una vez más las funciones y su relación quedan iguales.

En el último caso, la prueba de que (later a_1) es menor o igual a (later a_2) está dada por el constructor $\lesssim\mathsf{later}^l$ y una prueba $a_1 \lesssim a_2$ de tipo ((later a_1) $\lesssim\mathsf{B}$ ($\flat\ a_2$)). La demostración de este caso será similar a la del anterior, donde se utiliza el constructor 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 (($\flat\ a_1$) $\lesssim\mathsf{B}$ ($\flat\ a_2$)) en lugar de ((later a_1) $\lesssim\mathsf{B}$ ($\flat\ a_2$)), por lo que tiene un constructor later en el lado izquierdo que sobra. Esto se soluciona aplicando la función 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 construído a partir de bind, como se explicó en la sección 5.4. 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 (♭ 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) (♭ c)))
  merge-assoc (now a)  (later b) (now c)  = later (♯ (merge-assoc (now a) (♭ b) (now c)))
  merge-assoc (now a)  (later b) (later c) = later (♯ (merge-assoc (now a) (♭ b) (♭ c)))
  merge-assoc (later a) (now b)  (now c)  = later (♯ (merge-assoc (♭ a) (now b) (now c)))
  merge-assoc (later a) (now b)  (later c) = later (♯ (merge-assoc (♭ a) (now b) (♭ c)))
  merge-assoc (later a) (later b) (now c)  = later (♯ (merge-assoc (♭ a) (♭ b) (now c)))
  merge-assoc (later a) (later b) (later c) = later (♯ (merge-assoc (♭ a) (♭ 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 5.4. La demostración es, en esencia, igual a la introducida en la sección 6.3 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 × ⊤} _≡_ 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 (♭ x)))

module _ {A : Set} where
```

```
  open Equality {⊤ × A} _≡_ using (_≅_)
  open Equality.Rel

  lid : (a : A ⊥) → (merge unit a) ≅ (bind a (λ a → now (tt , a)))
  lid (now x)    = now prefl
  lid (later x) = later (♯ (lid (♭ 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 later^l como $\lesssim$later^l por la misma razón.

```
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 {A × B} _≡_ using (_≲_ ; _≅_)
  open Equality.Rel renaming (later^l to ≲later^l)

  merge-comp : (a₁ a₂ : A ⊥) (b₁ b₂ : B ⊥) → a₁ ≲A a₂ → b₁ ≲B b₂
                        → merge a₁ b₁ ≲ merge a₂ b₂
  merge-comp (now a₁) (now .a₁) (now b₁) (now .b₁) (now prefl) (now prefl) = now prefl
  merge-comp (now a₁) (now a₂) (later b₁) (later b₂) (now a∼) (later b≲) =
             later (♯ merge-comp (now a₁) (now a₂) (♭ b₁) (♭ b₂) (now a∼) (♭ b≲))
  merge-comp (now a₁) (now a₂) (now b₁) (later b₂) (now a∼) (≲later^l b≲) =
             ≲later^l (merge-comp (now a₁) (now a₂) (now b₁) (♭ b₂) (now a∼) b≲)
  merge-comp (now a₁) (now a₂) (later b₁) (later b₂) (now a∼) (≲later^l b≲) =
             later (♯ (merge-comp (now a₁) (now a₂) (♭ b₁) (♭ b₂) (now a∼) (later^r-1 b≲)))
  merge-comp (later a₁) (later a₂) (now b₁) (now b₂) (later a≲) (now b∼) =
             later (♯ (merge-comp (♭ a₁) (♭ a₂) (now b₁) (now b₂) (♭ a≲) (now b∼)))
  merge-comp (later a₁) (later a₂) (later b₁) (later b₂) (later a≲) (later b≲) =
             later (♯ (merge-comp (♭ a₁) (♭ a₂) (♭ b₁) (♭ b₂) (♭ a≲) (♭ b≲)))
  merge-comp (later a₁) (later a₂) (now b₁) (later b₂) (later a≲) (≲later^l b≲) =
             later (♯ (merge-comp (♭ a₁) (♭ a₂) (now b₁) (♭ b₂) (♭ a≲) b≲))
  merge-comp (later a₁) (later a₂) (later b₁) (later b₂) (later a≲) (≲later^l b≲) =
             later (♯ (merge-comp (♭ a₁) (♭ a₂) (♭ b₁) (♭ b₂) (♭ a≲) (later^r-1 b≲)))
  merge-comp (now a₁) (later a₂) (now b₁) (now b₂) (≲later^l a≲) (now b∼) =
             ≲later^l (merge-comp (now a₁) (♭ a₂) (now b₁) (now b₂) a≲ (now b∼))
  merge-comp (later a₁) (later a₂) (now b₁) (now b₂) (≲later^l a≲) (now b∼) =
             later (♯ (merge-comp (♭ a₁) (♭ a₂) (now b₁) (now b₂) (later^r-1 a≲) (now b∼)))
  merge-comp (now a₁) (later a₂) (later b₁) (later b₂) (≲later^l a≲) (later b≲) =
             later (♯ (merge-comp (now a₁) (♭ a₂) (♭ b₁) (♭ b₂) a≲ (♭ b≲)))
  merge-comp (later a₁) (later a₂) (later b₁) (later b₂) (≲later^l a≲) (later b≲) =
             later (♯ (merge-comp (♭ a₁) (♭ a₂) (♭ b₁) (♭ b₂) (later^r-1 a≲) (♭ b≲)))
  merge-comp (now a₁) (later a₂) (now b₁) (later b₂) (≲later^l a≲) (≲later^l b≲) =
             ≲later^l (merge-comp (now a₁) (♭ a₂) (now b₁) (♭ b₂) a≲ b≲)
```

```
merge-comp (now a₁) (later a₂) (later b₁) (later b₂) (≲later^l a≲) (≲later^l b≲) =
           later (♯ (merge-comp (now a₁) (♭ a₂) (♭ b₁) (♭ b₂) a≲ (later^r-1 b≲)))
merge-comp (later a₁) (later a₂) (now b₁) (later b₂) (≲later^l a≲) (≲later^l b≲) =
           later (♯ (merge-comp (♭ a₁) (♭ a₂) (now b₁) (♭ b₂) (later^r-1 a≲) b≲))
merge-comp (later a₁) (later a₂) (later b₁) (later b₂) (≲later^l a≲) (≲later^l b≲) =
           later (♯ (merge-comp (♭ a₁) (♭ a₂) (♭ b₁) (♭ b₂) (later^r-1 a≲) (later^r-1 b≲)))
```

Esta prueba toma cuatro valores: a_1 y a_2 de tipo $A\ \bot$ y b_1 y b_2 de tipo $B\ \bot$. 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, (now x) o (later x), y las dos pruebas pueden estar dadas de tres formas distintas: (now $x\sim$), (later $x\lesssim$) y ($\lesssim$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 $\lesssim$later^l, y una llamada co-recursiva dentro.
- En las llamadas co-recursivas todos los valores que tenían la forma (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 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 funtor 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 (now x) quedan igual y los que tienen la forma (later x) se reducen a ($\flat$ x).

```
merge-comm : (a : A ⊥) → (b : B ⊥)
           → merge a b ≅ bind (merge b a) (λ { (a , b) → now (b , a) })
merge-comm (now a) (now b)   = now prefl
merge-comm (now a) (later b) = later (♯ (merge-comm (now a) (♭ b)))
merge-comm (later a) (now b) = later (♯ (merge-comm (♭ a) (now b)))
merge-comm (later a) (later b) = later (♯ (merge-comm (♭ a) (♭ b)))
```

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

```
_≅⊥_ : ∀ {A} → A ⊥ → A ⊥ → Set
_≅⊥_ {A} = Equality._≅_ {A} _≡_

_≲⊥_ : ∀ {A} → A ⊥ → A ⊥ → Set
_≲⊥_ {A} = Equality._≲_ {A} _≡_
```

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 (♯ (≅⊥⇒≲⊥ (♭ 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 $\cong\bot$ implique la relación $\lesssim\bot$. Esta prueba está dada por el término $\cong\bot\Rightarrow\lesssim\bot$. En el caso base la prueba de igualdad está dada por el constructor Equality.now y una prueba $nowx{\sim}nowy$ de que $x \equiv y$. Para dar la prueba de la desigualdad, no se da la prueba de que (now x) $\lesssim$ (now y), sino que se prueba que (now y) $\gtrsim$ (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 $nowx{\sim}nowy$. 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 $x \gtrsim y$ e $y \gtrsim z$ entonces $x \gtrsim z$. Esto en términos del menor o igual es equivalente a decir que si $y \lesssim x$ y $z \lesssim y$ entonces $z \lesssim x$. 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$\lesssim\bot$.

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$\lesssim\bot$ se realiza este cambio y queda así concluída 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 _$\bot$, quedando únicamente el hueco para la prueba de la ley de intercambio.

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

delayConcurrent : ConcurrentMonad _⊥
delayConcurrent = makeConcurrentMonad
                    _≅⊥_
                    eq≅⊥
                    _≲⊥_
                    porder≲⊥
                    now
                    bind
                    bind-comp
                    left-identity
                    right-identity
                    bind-assoc
                    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 later^{l} y el módulo Equivalence.

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

  open Equality {C} _≡_
       renaming (_≅_ to _≅C_; _≳_ to _≳C_; now to nowC; later to laterC)
  open Equality {D} _≡_
       renaming (_≅_ to _≅D_; _≳_ to _≳D_; now to nowD; later to laterD)
  open Equality {C × D} _≡_ using (_≳_)
  open Equality.Rel renaming (later^l to ≲later^l)
  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 $C\ \bot$ y d_1 y d_2 de tipo $D\ \bot$, tales que $c_1 \cong\mathsf{C}\ c_2$ y $d_1 \cong\mathsf{D}\ d_2$, 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) (♭ d₁) (♭ d₂) pc (♭ d₁∼d₂)))
merge-ext (later c₁) (later c₂) (now d₁) (now d₂) (later c₁∼c₂) pd =
                  later (♯ (merge-ext (♭ c₁) (♭ c₂) (now d₁) (now d₂) (♭ c₁∼c₂) pd))
merge-ext (later c₁) (later c₂) (later d₁) (later d₂) (later c₁∼c₂) (later d₁∼d₂) =
                  later (♯ (merge-ext (♭ c₁) (♭ c₂) (♭ d₁) (♭ d₂) (♭ c₁∼c₂) (♭ 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 (♭ c₁) (♭ 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 (♭ d₁) (♭ 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 lema_1 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 lema_1 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)) (λ { (a , b) → merge (f a) (g b)}))
                                                  ≳ (merge (now c) (bind (♭ b) g))
lema₁ a b c f g p with ♭ 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)) (λ { (a , b) → merge (f a) (g b) }))
                                                  ≳ (merge (♭ y) (bind (♭ b) g))
lema₂ a b y f g p with ♭ b
...                  | now b₁ with g b₁
...                              | now b₃ =
  later^r-1 (merge-ext (f a) (later y) (now b₃) (now b₃) (≡⇒≅C (f a) (later y) p) (refl prefl))
...                              | later b₄ =
  merge-comp (♭ y) (f a) (later b₄) (later b₄) (later^r-1
  ((IsPartialOrder.reflexive porder≲⊥) (≡⇒≅C (later y) (f a) (psym p)))) (refl prefl)
lema₂ a b y f g p      | later b₂ with ♭ y | inspect ♭ 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 lema_2 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 $\flat\ y$. 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.

```
≡♯♭ : ∀ {S : Set} → (s : S ⊥) → (♭ (♯ s)) ≡ s
≡♯♭ s = prefl

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

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

El primer lema, ≡♯♭, prueba que los operadores ♭ y ♯ son opuestos en el sentido de que si se tiene un $s : S\ \bot$, se lo suspende con ♯ y luego se lo desencapsula con ♭, se obtiene como resultado el término inicial. La prueba de este lema es trivial. Al intentar probar el lema complementario, ≡♭♯, que toma un $s : \infty\ (S\ \bot)$ y postula que $(\sharp\ (\flat\ s)) \equiv s$, 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 construídos con ♯. Más aún, en el tercer lema, ≡⇒♯≡, se intenta probar que si se tienen dos términos, s_1 y s_2, que son iguales según la igualdad proposicional, entonces los resultados de aplicar ♯ a ambos términos también son iguales según la misma igualdad. Una vez más Agda no acepta la prueba trivial y el tipo objetivo que se observa en el modo interactivo es: Instances.ConcurrentMonad.♯-56 s_1 s_1 ≡ Instances.ConcurrentMonad.♯-57 s_1 s_1 . Con estos ejemplos se evidencia que esta notación, en particular el operador ♯, tiene problemas para deducir la igualdad de los términos, es como si hubiera diferentes tipos de ♯, con diferentes números cada uno.

6.5. Reduciendo el problema a los conaturales

Los números conaturales pueden verse como una versión simplificada del tipo *delay* en la que no hay un tipo de retorno, o bien, este tipo es ⊤, el cual tiene un único habitante. Así, cada constructor later se correspondería con un constructor suc.

A raíz de las dificultades encontradas en el intento de probar la ley de intercambio para la mónada *delay*, se decidió simplificar el problema e intentar probar la ley de intercambio para los números conaturales, en este caso la ley a demostrar sería la ley de intercambio de los monoides concurrentes. El objetivo de esta simplificación es ver si el problema para reconocer términos iguales se debe al valor que (posiblemente) retornan los términos de tipo A ⊥ o si el problema está en el soporte para coinducción utilizado.

6.5.1. Definición de los conaturales con notación musical

En la sección 4.4.1 se introdujo la siguiente definición para los números conaturales:

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

Se definen para el tipo dado dos relaciones: una de igualdad, a la cual se la llamará bisemejanza, y la otra de orden. Como se puede observar, estas relaciones se definen de manera análoga a las relaciones de bisemejanza fuerte y desigualdad definidas para el tipo *delay*.

```
data _≈_ : Coℕ → Coℕ → Set where
  zero : zero ≈ zero
  suc : ∀ {m n} → ∞ (♭ m ≈ ♭ n) → suc m ≈ suc n

data _≳_ : Coℕ → Coℕ → Set where
  zero : zero ≳ zero
  suc  : ∀ {m n} → ∞ (♭ m ≳ ♭ n) → suc m ≳ suc n
  suc′ : ∀ {m n} → (♭ m) ≳ n → suc m ≳ n
```

A continuación se prueba que la relación de igualdad definida es una relación de equivalencia, es decir que es reflexiva simétrica y transitiva. Las demostraciones de tales propiedades son bastante simples, los casos base son triviales y los demás se resuelven mediante co-recursión.

```
refl≈ : {n : Coℕ} → n ≈ n
refl≈ {zero}  = zero
```

```
refl≈ {suc x} = suc (♯ refl≈)

sym≈ : {m n : Coℕ} → m ≈ n → n ≈ m
sym≈ zero     = zero
sym≈ (suc p) = suc (♯ (sym≈ (♭ p)))

trans≈ : {m n o : Coℕ} → m ≈ n → n ≈ o → m ≈ o
trans≈ zero     zero     = zero
trans≈ (suc p) (suc q) = suc (♯ (trans≈ (♭ p) (♭ q)))

eq≈ : IsEquivalence _≈_
eq≈ = record { refl = refl≈
             ; sym = sym≈
             ; trans = trans≈ }
```

Siendo la igualdad una relación de equivalencia, se prueba seguidamente que la relación de orden es un orden parcial respecto de la bisemejanza. Para ello se definen algunos operadores que facilitan la manipulación de los constructores suc. La función suc^{r-1} es la inversa del constructor suc^{l} y sirve para quitar un constructor suc del lado derecho de la desigualdad. La función suc^{-1} es la inversa del constructor suc de la desigualdad y quita un constructor de ambos lados de la misma. Por último, ≳suc demuestra que, para cualquier valor n de tipo ∞ Coℕ, suc $n \gtrsim \flat\ n$.

```
refl≳ : {n m : Coℕ} → n ≈ m → n ≳ m
refl≳ zero        = zero
refl≳ (suc m≈n) = suc (♯ (refl≳ (♭ m≈n)))

suc^r-1 : ∀ {m} {n : ∞ Coℕ} → m ≳ suc n → m ≳ ♭ n
suc^r-1 (suc p)    = suc^l (♭ p)
suc^r-1 (suc^l H)  = suc^l (suc^r-1 H)

suc^-1 : ∀ {m} {n : ∞ Coℕ} → suc m ≳ suc n → ♭ m ≳ ♭ n
suc^-1 (suc x)   = ♭ x
suc^-1 (suc^l H) = suc^r-1 H

≳suc : {n : ∞ Coℕ} → suc n ≳ ♭ n
≳suc = suc^l (refl≳ refl≈)

trans≳zero : {m n : Coℕ} → m ≳ n → n ≳ zero → m ≳ zero
trans≳zero zero     zero     = zero
trans≳zero (suc p)  (suc^l q) = suc^l (trans≳zero (♭ p) q)
trans≳zero (suc^l p) q        = suc^l (trans≳zero p q)

mutual
  trans≳suc : ∀ {m n : Coℕ} {o} → m ≳ n → n ≳ suc o → m ≳ suc o
  trans≳suc (suc p)   q = suc (♯ trans≳ (♭ p) (suc^-1 q))
  trans≳suc (suc^l p) q = suc (♯ (trans≳ p (suc^r-1 q)))

  trans≳ : {m n p : Coℕ} → m ≳ n → n ≳ p → m ≳ p
  trans≳ {p = zero}  p q = trans≳zero p q
  trans≳ {p = suc x} p q = trans≳suc p q
```

```
antisym≳ : ∀ {m n : Coℕ} → m ≳ n → n ≳ m → m ≈ n
antisym≳ zero          zero          = zero
antisym≳ (suc m≳n)     (suc n≳m)     = suc (♯ (antisym≳ (♭ m≳n) (♭ n≳m)))
antisym≳ (suc m≳n)     (suc^l n≳sucm) = suc (♯ antisym≳ (♭ m≳n) (trans≳ n≳sucm ≳suc))
antisym≳ (suc^l m≳sucn) (suc n≳m)     = suc (♯ (antisym≳ (trans≳ m≳sucn ≳suc) (♭ n≳m)))
antisym≳ (suc^l m≳sucn) (suc^l n≳sucm) = suc (♯ (antisym≳ (trans≳ m≳sucn ≳suc)
                                                          (trans≳ n≳sucm ≳suc)))

partial≳ : IsPartialOrder _≈_ _≳_
partial≳ = record { isPreorder = record { isEquivalence = eq≈
                                        ; reflexive = refl≳
                                        ; trans = trans≳ }
                  ; antisym = antisym≳ }
```

La prueba de transitividad es la única que tiene una dificultad un poco mayor ya que se definió en tres partes. trans≳zero prueba la transitividad para el caso en que el elemento más pequeño es zero. Esta tiene pocos casos y se define co-recursivamente. Luego están trans≳suc y trans≳ que son mutuamente co-recursivas. La primera prueba el caso particular en el que el elemento más chico es sucesor de algún número y la segunda es la prueba general, la cual utiliza las dos anteriores. Son mutuamente co-recursivas puesto que para probar trans≳suc se necesita también utilizar trans≳.

Por último, se definen las dos operaciones que forman las estructuras monoidales de Coℕ junto con el neutro zero.

- max se define como el máximo de dos co-números y es la versión simplificada del operador merge definido para el tipo *delay*. Esto se puede notar ya que en merge, en el caso de tener dos valores de la forma (later x), el resultado se construye con un sólo later y ambos términos reducidos dentro de la llamada co-recursiva. Se puede pensar que el resultado de aplicar merge es el par de los valores de retorno de sus argumentos con una cantidad de constructores later equivalente al máximo de las cantidades que tenía cada argumento.
- sum calcula la suma de dos conúmeros. En este caso, a diferencia de max, todos los constructores suc persisten. En el caso de que ambos conúmeros sean de la forma (suc x), el resultado consta de dos constructores suc consecutivos y dentro de ellos la llamada co-recursiva con ambos números reducidos. Aunque es un poco más difícil ver el paralelismo, esta función se corresponde con el operador bind definido para el tipo *delay*. Si se observa su definición, se puede ver que la función f queda intacta hasta el caso en el que el primer argumento es de tipo (now x), en el cual se aplica al valor x. Si se piensa el comportamiento global de la función, la cantidad de constructores later que tiene el resultado de aplicar bind, es la suma de la cantidad que posee el primer argumento y la cantidad que posee $f\ x$. Todos los later de $f\ x$ quedan dentro de los del primer argumento.

```
max : Coℕ → Coℕ → Coℕ
max zero     n        = n
max (suc m) zero     = suc m
max (suc m) (suc n) = suc (♯ (max (♭ m) (♭ n)))

sum : Coℕ → Coℕ → Coℕ
sum zero     n        = n
sum (suc m) zero     = suc m
sum (suc m) (suc n) = suc (♯ (suc (♯ (sum (♭ m) (♭ n)))))
```

6.5.2. ¿Se puede probar la ley de intercambio?

Contando con las definiciones básicas introducidas en la sección anterior, se analiza la prueba de la ley de intercambio para los números conaturales. En ella se utilizan varios lemas cuyas pruebas se encuentran en el Apéndice A.2. Se listan a continuación sólo los tipos de las propiedades que se utilizan.

- $\equiv\!\Rightarrow\!\gtrsim : \{n_1\ n_2 : \mathsf{Co}\mathbb{N}\} \to n_1 \equiv n_2 \to n_1 \gtrsim n_2$
  prueba que la desigualdad es reflexiva respecto de la igualdad proposicional.
- $\mathsf{sumzero}_2 : \{m : \mathsf{Co}\mathbb{N}\} \to \mathsf{sum}\ m\ \mathsf{zero} \gtrsim m$
  prueba que sumar zero a un conúmero m es mayor o igual a m.
- $\gtrsim\mathsf{sum} : \{m_1\ m_2\ n_1\ n_2 : \mathsf{Co}\mathbb{N}\} \to m_1 \gtrsim m_2 \to n_1 \gtrsim n_2 \to \mathsf{sum}\ m_1\ n_1 \gtrsim \mathsf{sum}\ m_2\ n_2$
  prueba que la suma es compatible con la relación de orden.
- $\gtrsim\mathsf{max} : \{m_1\ m_2\ n_1\ n_2 : \mathsf{Co}\mathbb{N}\} \to m_1 \gtrsim m_2 \to n_1 \gtrsim n_2 \to \mathsf{max}\ m_1\ n_1 \gtrsim \mathsf{max}\ m_2\ n_2$
  prueba que el operador max es compatible con la relación de orden.

En el siguiente código se puede observar el esqueleto de la prueba de interchange en el cual muchos casos están resueltos de manera trivial. Algunos de los demás casos se pueden probar mediante el uso de lemas extra pero, para este análisis, se centrará el foco en los últimos dos casos.

```
interchange : (a b c d : Coℕ) → (sum (max a b) (max c d)) ≳ (max (sum a c) (sum b d))
interchange zero    zero    zero    zero    = zero
interchange zero    zero    zero    (suc d) = refl≳ refl≈
interchange zero    zero    (suc c) zero    = refl≳ refl≈
interchange zero    zero    (suc c) (suc d) = suc (♯ refl≳ refl≈)
interchange zero    (suc b) zero    zero    = suc (♯ refl≳ refl≈)
interchange zero    (suc b) zero    (suc d) = suc (♯ (suc (♯ refl≳ refl≈)))
interchange zero    (suc b) (suc c) zero    = {! !}
interchange zero    (suc b) (suc c) (suc d) = {! !}
interchange (suc a) zero    zero    zero    = refl≳ refl≈
interchange (suc a) zero    zero    (suc d) = {! !}
interchange (suc a) zero    (suc c) zero    = suc (♯ (suc (♯ refl≳ refl≈)))
interchange (suc a) zero    (suc c) (suc d) = suc (♯ {! !})
interchange (suc a) (suc b) zero    zero    = suc (♯ refl≳ refl≈)
interchange (suc a) (suc b) zero    (suc d) = {! !}
interchange (suc a) (suc b) zero    (suc d) with ♭ a | inspect ♭ a
... | zero    | [ eq ] = suc (♯ trans≳ (suc (♯ (trans≳ (≳sum (≳max (≡⇒≳ eq) (refl≳ refl≈))
                         (refl≳ refl≈)) (refl≳ refl≈)))) (≳max (≡⇒≳ (sym eq)) (refl≳ refl≈)))
... | suc a₁ | [ eq ] = suc (♯ (trans≳ (suc (♯ (trans≳ (≳sum (≳max (≡⇒≳ eq) (refl≳ refl≈))
                         (refl≳ refl≈)) (trans≳ ((≳sum {max (suc a₁) (♭ b)} {max (♭ a₁) (♭ b)}
                         (≳max {suc a₁} {♭ a₁} (≳suc) (refl≳ refl≈)) (refl≳ refl≈)))
                         (trans≳ (interchange (♭ a₁) (♭ b) zero (♭ d)) (≳max {(sum (♭ a₁) zero)}
                         {(♭ a₁)} sumzero₂ (refl≳ refl≈)))))))) (≳max (≡⇒≳ (sym eq)) (refl≳ refl≈))))
interchange (suc a) (suc b) (suc c) (suc d) = suc (♯ (suc (♯ (interchange (♭ a) (♭ b) (♭ c) (♭ d)))))
```

En el penúltimo caso se puede observar que el problema para reconocer términos iguales cuando se utiliza la estructura with persiste aún eliminando los tipos de retorno. Una vez más uno se ve obligado a utilizar la cláusula inspect para "obligar" a Agda a notar que dos términos son iguales por definición. Esto hace, además, que la prueba sea mucho más larga de lo necesario,

puesto que hay que incluir muchas veces un conector trans$\gtrsim$ extra que agrega una prueba más sólo para que se unifiquen términos iguales. Aún haciendo todas las pruebas necesarias, surge otro problema: a la hora de realizar la llamada co-recursiva, Agda lanza un error que dice que falló el chequeo de terminación, aún cuando la llamada está suspendida dentro de un operador $\sharp$ y todos los términos se reducen (salvo zero que es el caso base), por lo que uno pensaría que la prueba sí es productiva. Esto mismo sucede en el último caso, el cual se resuelve simplemente utilizando dos constructores suc y dentro la llamada co-recursiva en la cual todos los términos se reducen.

Esta falla en el chequeo de terminación se repitió en varios lemas que fueron parte de diferentes intentos realizados para demostrar interchange. Llamadas que parecen productivas y válidas no pasan el chequeo. Como ejemplo básico de una propiedad que, intuitivamente, debería poder demostrarse sin problemas, se presenta el caso de $\gtrsim$zero que postula que cualquier conúmero es mayor o igual a zero. Una vez más, la prueba de este lema no pasa el chequeo de terminación.

```
≳zero : (n : CoℕI) → n ≳ zero
≳zero zero     = zero
≳zero (suc n) = suc^I (≳zero (♭ n))
```

Dada la dificultad de reconocimiento de términos iguales y el problema con el chequeo de terminación, se decidió después de muchos intentos, idas y vueltas, descartar la notación musical como soporte para coinducción.

6.6. Cambio de paradigma: *sized types*

Luego de descartar la notación musical como soporte para la coinducción, los *sized types* fueron la opción elegida. Como se introdujo en la sección 4.4.2, estos poseen índices que ayudan en el chequeo de terminación, por lo que el objetivo de esta sección es ver si los problemas con tal chequeo encontrados en la notación musical están presentes también en esta notación o no.

En caso de que no se presenten dificultades con la notación, se intentará demostrar que los números conaturales forman un monoide concurrente con la suma y el máximo, demostrando la ley de intercambio para este conjunto que es una versión simplificada del tipo *delay*. La implementación de números conaturales que se presenta a continuación está basada en el módulo Conat definido por Danielsson [Dan18].

6.6.1. Definición de los conaturales utilizando *sized types*

En la sección 4.4.2 se definieron los números conaturales utilizando *sized types* de la siguiente manera:

```
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
```

Esta definición consta de dos partes mutuamente recursivas que trabajan juntas para asegurar la terminación de los programas, tal como se explicó cuando se introdujo esta definición. En general muchas de las definiciones sobre este tipo de dato se darán de la misma manera.

Antes de pasar a definir las relaciones entre números conaturales, se definen dos operaciones que serán de mucha utilidad más adelante. La primera de ellas es pred, la cual calcula el predecesor de un conúmero, donde el predecesor de zero es sí mismo y el predecesor de cualquier otro conúmero se calcula quitando un constructor suc.

```
pred : ∀ {i} {j : Size< i} → Conat i → Conat j
pred zero = zero
pred (suc n) = force n
```

La segunda función que se define es una conversión de números naturales a números conaturales. Esta función se define, al igual que el tipo Conat, en dos partes mutuamente recursivas.

```
mutual

  ⌜_⌝ : ∀ {i} → ℕ → Conat i
  ⌜ zero ⌝ = zero
  ⌜ suc n ⌝ = suc ⌜ n ⌝′

  ⌜_⌝′ : ∀ {i} → ℕ → Conat′ i
  force ⌜ n ⌝′ = ⌜ n ⌝
```

Se define ahora entonces la primera de las relaciones entre conúmeros: la bisemejanza. Esta relación está dada también en dos partes y se define únicamente para números de tipo Conat ∞ que son los conúmeros completamente definidos. La segunda definición, [_]_∼′_, está dada por un tipo record coinductivo que tiene un único campo: force. Dados un tamaño i y dos conúmeros m y n de tipo Conat ∞, este campo da una prueba de que m y n son iguales según la primera definición ([_]_∼_) para un tamaño j menor a i.

La primera definición, [_]_∼_, está dada por un tipo data y consta de dos constructores. zero postula que zero es igual a sí mismo. Por otra parte, suc dice que, dados $m\ n$: Conat ∞, si se tiene que force m y force n son iguales según la segunda definición para un cierto tamaño i, es decir que son iguales según la definición actual para un tamaño j menor a i, entonces suc m y suc n son iguales en la primera relación para el índice i.

```
mutual

  infix 4 [_]_∼_ [_]_∼′_

  data [_]_∼_ (i : Size) : Conat ∞ → Conat ∞ → Set where
    zero : [ i ] zero ∼ zero
    suc : ∀ {m n} → [ i ] force m ∼′ force n → [ i ] suc m ∼ suc n

  record [_]_∼′_ (i : Size) (m n : Conat ∞) : Set where
    coinductive
```

```
    field
      force : {j : Size< i} → [ j ] m ∼ n

open [_]_∼′_ public
```

La bisemejanza es una relación de equivalencia, es decir que es reflexiva, simétrica y transitiva, tal como se muestra en los lemas que siguen. Estas demostraciones se resuelven todas de la misma manera: el caso base se prueba con el costructor zero y el caso coinductivo se prueba con el constructor suc y una llamada co-recursiva dentro.

```
reflexive-∼ : ∀ {i} n → [ i ] n ∼ n
reflexive-∼ zero    = zero
reflexive-∼ (suc n) = suc λ { .force → reflexive-∼ (force n) }

symmetric-∼ : ∀ {i m n} → [ i ] m ∼ n → [ i ] n ∼ m
symmetric-∼ zero = zero
symmetric-∼ (suc p) = suc λ { .force → symmetric-∼ (force p) }

transitive-∼ : ∀ {i m n o} → [ i ] m ∼ n → [ i ] n ∼ o → [ i ] m ∼ o
transitive-∼ zero zero = zero
transitive-∼ (suc p) (suc q) =
  suc λ { .force → transitive-∼ (force p) (force q) }
```

Es el turno ahora de definir la relación de orden, la cual está dada también en dos partes y se define sólo para los conúmeros completamente definidos (Conat ∞). Al igual que para la bisemejanza, la segunda definición da, para un cierto tamaño i y conúmeros m n, una prueba de que m es menor o igual a n según la primera definición pero para un tamaño j menor a i.

La primera definición, la cual es la principal, tiene dos constructores. El constructor zero postula que para cualquier conúmero n, zero es menor o igual a n para el tamaño dado i. El constructor suc, dada una prueba de que force m es menor o igual a force n para un tamaño j menor a i, asegura que suc m es menor o igual a suc n para el tamaño i.

```
mutual

  infix 4 [_]_≤_ [_]_≤′_

  data [_]_≤_ (i : Size) : Conat ∞ → Conat ∞ → Set where
    zero : ∀ {n} → [ i ] zero ≤ n
    suc : ∀ {m n} → [ i ] force m ≤′ force n → [ i ] suc m ≤ suc n

  record [_]_≤′_ (i : Size) (m n : Conat ∞) : Set where
    coinductive
    field
      force : {j : Size< i} → [ j ] m ≤ n

open [_]_≤′_ public
```

Esta relación es un orden parcial, es decir que es reflexiva, antisimétrica y transitiva. Se prueban a continuación estos tres lemas. La reflexividad está dada respecto de la igualdad proposicional, mientras que la antisimetría se prueba respecto de la bisemejanza. Las tres pruebas se resuelven sin dificultades.

```
reflexive-≤ : ∀ {i} n → [ i ] n ≤ n
reflexive-≤ zero = zero
reflexive-≤ (suc n) = suc λ { .force → reflexive-≤ (force n) }

transitive-≤ : ∀ {i m n o} → [ i ] m ≤ n → [ i ] n ≤ o → [ i ] m ≤ o
transitive-≤ zero _ = zero
transitive-≤ (suc p) (suc q) =
  suc λ { .force → transitive-≤ (force p) (force q) }

antisymmetric-≤ : ∀ {i m n} → [ i ] m ≤ n → [ i ] n ≤ m → [ i ] m ∼ n
antisymmetric-≤ zero zero = zero
antisymmetric-≤ (suc p) (suc q) =
  suc λ { .force → antisymmetric-≤ (force p) (force q) }
```

Para probar que el orden es un orden parcial respecto de la bisemejanza falta un ingrediente más: la reflexividad respecto de la bisemejanza. Es decir que la relación de bisemejanza implica la relación de orden. Se prueba este lema a continuación:

```
∼→≤ : ∀ {i m n} → [ i ] m ∼ n → [ i ] m ≤ n
∼→≤ zero = zero
∼→≤ (suc p) = suc λ { .force → ∼→≤ (force p) }
```

Luego de dar las definiciones básicas se demuestran tres lemas elementales en relación al orden que serán de utilidad para las pruebas que se llevarán a cabo más adelante. El primero de ellos postula que, para cualquier n : Conat′ ∞ y tamaño i, force n es menor o igual a suc n para el tamaño i. Para esta prueba es necesario utilizar la función helper que toma un conúmero m y una prueba de que $m \equiv$ force n y hace la demostración valiéndose de esta prueba. Esto se debe a que en esta notación también hay un pequeño conflicto para unificar términos (en este caso con el campo force), pero se puede solucionar sin mayores dificultades y no vuelve a repetirse este problema en ninguna de las demostraciones que siguen.

```
≤suc : ∀ {i n} → [ i ] force n ≤ suc n
≤suc = helper _ refl
  where
  helper : ∀ {i} m {n} → m ≡ force n → [ i ] m ≤ suc n
  helper zero _ = zero
  helper (suc m) 1+m≡ = suc λ { .force {j} →
                                subst ([ j ] _ ≤_) 1+m≡ ≤suc }
```

El segundo lema también tiene que ver con la relación entre force y suc. En este caso, para cualesquiera m : Conat ∞, n : Conat′ ∞ e i : Size, se prueba que si se tiene una prueba de que [i] $m \leq'$ force n, entonces m es menor o igual a suc n para el mismo tamaño i. El caso base de esta prueba se da con el constructor zero. En el otro caso se utiliza el constructor suc y dentro la propiedad transitiva entre el lema anterior (que prueba que [j] force $m \leq$ suc m) y (force p) (que prueba que [j] suc $m \leq$ force n).

```
≤-step : ∀ {m n i} → [ i ] m ≤′ force n → [ i ] m ≤ suc n
≤-step {zero} _ = zero
≤-step {suc m} {n} p = suc λ { .force → transitive-≤ ≤suc (force p) }
```

Por último, el tercer lema que se prueba indica que el predecesor de un conúmero es menor o igual a dicho conúmero. Esta prueba se resuelve utilizando también el lema ≤suc.

```
pred≤ : ∀ {i m} → [ i ] pred m ≤ m
pred≤ {i} {zero} = zero
pred≤ {i} {suc m} = ≤suc
```

Antes de intentar demostrar la ley de intercambio y dar la instancia de ConcurrentMonoid para el tipo Conat ∞, es importante saber si esta representación tiene también problemas con el chequeo de terminación o no. Si bien no se puede asegurar por completo, es deseable corroborar al menos el ejemplo dado para la notación musical. En este caso, la prueba de que todo conúmero es mayor o igual a zero está dada por la definición de la relación de orden. Uno podría pensar entonces que la forma de dar tal definición influye en el desarrollo de las demostraciones a realizar. Para refutar esto, se muestra una definición alternativa de la relación de orden, la cual es análoga a la utilizada en la notación musical.

```
mutual

  infix 4 [_]_⊑_ [_]_⊑′_

  data [_]_⊑_ (i : Size) : Conat ∞ → Conat ∞ → Set where
    zero : [ i ] zero ⊑ zero
    sucr : ∀ {m n} → [ i ] m ⊑′ force n → [ i ] m ⊑ suc n
    suc : ∀ {m n} → [ i ] force m ⊑′ force n → [ i ] suc m ⊑ suc n

  record [_]_⊑′_ (i : Size) (m n : Conat ∞) : Set where
    coinductive
    field
      force : {j : Size< i} → [ j ] m ⊑ n

open [_]_⊑′_ public
```

Usando esta relación alternativa, se prueba a continuación que zero es menor o igual a todo conúmero, prueba que se puede hacer sin problemas gracias a los índices que ayudan en el chequeo de terminación.

```
zero⊑ : ∀ {i} n → [ i ] zero ⊑ n
zero⊑ zero = zero
zero⊑ (suc H) = sucr λ { .force → zero⊑ (H .force) }
```

Más aún, se puede probar que ambas relaciones de orden son equivalentes.

```
⊑⇒≤ : ∀ {i} {n m} → [ i ] n ⊑ m → [ i ] n ≤ m
⊑⇒≤ zero = zero
⊑⇒≤ {n = zero} {m = suc m} (sucr H) = zero
⊑⇒≤ {n = suc n} {m = suc m} (sucr H) = suc λ { .force → transitive-≤ ≤suc (⊑⇒≤ (H .force)) }
⊑⇒≤ (suc H) = suc λ { .force {j} → ⊑⇒≤ (force H) }

≤⇒⊑ : ∀ {i} {n m} → [ i ] n ≤ m → [ i ] n ⊑ m
≤⇒⊑ {n} {.zero} zero = zero⊑ _
≤⇒⊑ {n} {.(suc _)} (suc H) = suc λ { .force → ≤⇒⊑ (H .force) }
```

Se utiliza entonces la primera por cuestiones de practicidad.

6.6.2. Prueba de que los conaturales forman un monoide concurrente

Una vez definidos los números conaturales y sus relaciones, y con bastante seguridad de que esta notación no generará tantos conflictos como la primera, se procede a demostrar que los conúmeros forman un monoide concurrente. Para comenzar, se define la operación correspondiente a la suma como sigue:

```
infixl 6 _+_

_+_ : ∀ {i} → Conat i → Conat i → Conat i
zero + n = n
suc m + n = suc λ { .force → force m + n }
```

La suma junto con el valor especial zero forman una estructura monoidal. Es decir que zero es neutro a izquierda y derecha de la suma y que, además, la suma es asociativa. Estas tres propiedades se demuestran sin mayores dificultades.

```
+-left-identity : ∀ {i} n → [ i ] zero + n ∼ n
+-left-identity = reflexive-∼

+-right-identity : ∀ {i} n → [ i ] n + zero ∼ n
+-right-identity zero = zero
+-right-identity (suc n) = suc λ { .force → +-right-identity (force n) }

+-assoc : ∀ m {n o i} → [ i ] m + (n + o) ∼ (m + n) + o
+-assoc zero = reflexive-∼ _
+-assoc (suc m) = suc λ { .force → +-assoc (force m) }
```

Se prueba a continuación la compatibilidad de la suma con la relación de orden, es decir que la suma es monótona respecto de ≤. Para ello se prueba antes un lema que indica que un número siempre es menor o igual a sí mismo sumado con otro. En la prueba de _max-mono_ sólo se requiere dicho lema con la suma a uno de los lados, se prueba también su simétrico ya que será utilizado más adelante. Estos dos lemas no podían probarse para la notación musical puesto que fallaba el chequeo de terminación, con *sized types* no se tiene este problema.

```
infixl 6 _+-mono_

m≤m+n : ∀ {m n i} → [ i ] m ≤ m + n
m≤m+n {zero} {n} = zero
m≤m+n {suc m} {n} = suc (λ { .force → m≤m+n })

m≤n+m : ∀ {i m n} → [ i ] m ≤ n + m
m≤n+m {n = zero} = reflexive-≤ _
m≤n+m {n = suc n} = ≤-step λ { .force → m≤n+m }

_+-mono_ : ∀ {i m₁ m₂ n₁ n₂} →
           [ i ] m₁ ≤ m₂ → [ i ] n₁ ≤ n₂ → [ i ] m₁ + n₁ ≤ m₂ + n₂
_+-mono_ {m₁ = m₁} {m₂} {n₁} {n₂} zero q = transitive-≤ q m≤n+m
suc p +-mono q = suc λ { .force → force p +-mono q }
```

La compatibilidad de la suma con el orden se prueba por coinducción en el primer argumento: $m_1 \leq m_2$. En el caso base, cuando esta prueba está dada por el constructor zero, se tiene que

$m_1 =$ zero, por lo que lo se quiere probar es que $n_1 \leq m_2 + n_2$. Se utiliza primero el argumento q para probar que $n_1 \leq n_2$ y luego el lema m≤n+m para probar que $n_2 \leq m_2 + n_2$, uniendo ambas pruebas con la transitividad de $\leq$.

El caso en el que el primer argumento es suc p, se tiene que $m_1 =$ suc m_1', $m_2 =$ suc m_2' y $p :$ [i] force $m_1' \leq'$ force m_2'. El objetivo es probar que [i] suc λ { .force $\to$ force m_1' + n_1 } $\leq$ suc λ { .force $\to$ force m_2' + n_2 }. Quitando el constructor suc a ambos lados (lo cual se realiza al aplicar en la prueba el constructor de la desigualdad que tiene el mismo nombre) queda por probar que [j] force m_1' + $n_1 \leq$ force m_2' + n_2, para algún j menor a i. Esta última prueba es el resultado de aplicar _+-mono_ a force p y q.

Terminadas las propiedades respectivas a la suma, se pasa a definir el operador max. Para evitar tener tres casos en la definición como se tenía en la notación musical, se agrupan los últimos dos en uno sólo. El máximo entre (suc m) y n se define como el sucesor del máximo entre (force m) y (pred n). Al aplicar el predecesor al segundo argumento lo que se hace es unificar el caso en que $n =$ zero y el caso en que $n =$ suc n'. En el primer caso la función pred no hace nada y se obtiene el sucesor del máximo entre (force m) y zero, al igual que se definió para la notación musical. En el segundo caso aplicar el predecesor quita un constructor suc al segundo argumento, lo cual se condice con el comportamiento que se tenía en la notación musical donde, cuando ambos conúmeros eran sucesores de otros, se dejaba un único constructor suc y se reducían ambos argumentos en la llamada co-recursiva.

```
max : ∀ {i} → Conat i → Conat i → Conat i
max zero n = n
max (suc m) n = suc λ { .force → max (force m) (pred n) }
```

El operador max junto con el valor especial zero deben formar también una estructura monoidal. Se prueban a continuación ambas identidades de forma muy simple.

```
max-left-identity : ∀ {i} n → [ i ] max zero n ∼ n
max-left-identity = reflexive-∼

max-right-identity : ∀ {i} n → [ i ] max n zero ∼ n
max-right-identity zero = zero
max-right-identity (suc n) = suc λ { .force → max-right-identity (force n) }
```

Para demostrar la asociatividad del operador max, se necesitan dos lemas extra:

- _max-cong_ : ∀ {i m_1 m_2 n_1 n_2} → [i] $m_1 \sim m_2$ → [i] $n_1 \sim n_2$ → [i] max m_1 $n_1 \sim$ max m_2 n_2 que prueba que el operador max es compatible con la bisemejanza
- y pred-max : ∀ {i} m n → [i] max (pred m) (pred n) $\sim$ pred (max m n) que prueba que el máximo de los predecesores es igual al predecesor del máximo.

Las demostraciones de estos lemas se encuentran en el apéndice A.3 para no extender demasiado esta sección.

```
max-assoc : ∀ {i} n m o → [ i ] max (max n m) o ∼ max n (max m o)
max-assoc zero m o = reflexive-∼ (max m o)
max-assoc (suc n) m o = suc λ { .force → transitive-∼ (max-assoc (force n) (pred m) (pred o))
                                  (reflexive-∼ (force n) max-cong pred-max m o) }
```

Se prueba max-assoc por coinducción en el primer argumento. En el caso en que $n =$ zero, el objetivo pasa a ser probar que [i] max m $o \sim$ max m o, puesto que max zero $m = m$

y max zero (max m o) = max m o. Luego se resuelve ese caso mediante la reflexividad de la bisemejanza.
En el segundo caso el objetivo es ver que: [i] max (max (suc n) m) o ∼ max (suc n) (max m o). O lo que es equivalente:

```
[ i ] max (suc λ { .force → max (force n) (pred m) }) o ∼
    suc λ { .force → max (force n) (pred (max m o)) }
```

Aplicando el max de afuera en el lado izquierdo se obtiene:

```
[ i ] suc λ { .force → max (force λ { .force → max (force n) (pred m) }) (pred o) } ∼
    suc λ { .force → max (force n) (pred (max m o)) }
```

Al aplicar en la prueba suc λ { .force → ... } se quita esta construcción a cada lado de la igualdad. El nuevo objetivo es probar que:

```
[ j ] max (max (force n) (pred m)) (pred o) ∼ max (force n) (pred (max m o))
```

Se resuelve esta prueba utilizando transitive-∼ entre (max-assoc (force n) (pred m) (pred o)) que prueba: [j] max (max (force n) (pred m)) (pred o) ∼ max (force n) (max (pred m) (pred o)) y _max-cong_ aplicado a reflexive-∼ (force n) que deja el primer argumento como está y pred-max que cambia (max (pred m) (pred o)) por (pred (max m o)).

Se presenta a continuación la función _max-mono_, la cual, análogamente a _+-mono_, representa la compatibilidad del operador max con la relación de orden. También de manera similar a lo ocurrido con _+-mono_, para demostrar este lema se requiere de un lema extra, ^{r}≤max, similar a m≤n+m, que postula que un conúmero siempre es menor o igual al máximo entre él mismo y otro. Se prueba también su lema simétrico, ^{l}≤max, el cual es análogo a m≤m+m ya que también será necesario en la prueba de la ley de intercambio. Ambos lemas se demuestran sin dificultade de manera co-recursiva.

```
ˡ≤max : ∀ {i} m n → [ i ] m ≤ max m n
ˡ≤max zero _ = zero
ˡ≤max (suc m) n = suc λ { .force → ˡ≤max (force m) (pred n) }

ʳ≤max : ∀ {i} m n → [ i ] n ≤ max m n
ʳ≤max zero _        = reflexive-≤ _
ʳ≤max (suc _) zero = zero
ʳ≤max (suc m) (suc n) = suc λ { .force → ʳ≤max (force m) (force n) }

infixl 6 _max-mono_

_max-mono_ : ∀ {i m₁ m₂ n₁ n₂} →
             [ i ] m₁ ≤ m₂ → [ i ] n₁ ≤ n₂ → [ i ] max m₁ n₁ ≤ max m₂ n₂
_max-mono_ {m₁ = m₁} {m₂} {n₁} {n₂} zero q = transitive-≤ q (ʳ≤max m₂ n₂)
suc p max-mono zero = suc λ { .force → (force p) max-mono zero }
suc p max-mono suc q = suc λ { .force → (force p) max-mono (force q) }
```

La demostración de _max-mono_ es análoga a la de _+-mono_, con la única diferencia de que se separa en casos el segundo argumento cuando el primero tiene la forma (suc n). En ambos casos la prueba se resuelve con un constructor suc y dentro una llamada co-recursiva, en

el primer caso el segundo argumento de dicha llamada es la prueba zero y en el otro es la prueba del segundo argumento de la llamada actual quitándole un constructor suc.

Queda una propiedad más por probar para el operador max: la conmutatividad. Dicha propiedad es requerida para que los conaturales formen un monoide concurrente. En el caso en que ambos conúmeros son zero, la prueba es trivial. Cuando uno de los dos es zero y el otro tiene la forma (suc x), el resultado de max zero (suc x) es, por definición, (suc x) y se usa el lema max-right-identity para demostrar que max (suc x) zero tiene el mismo resultado. Cuando ambos argumentos están formados por un constructor suc, se resuelve co-recursivamente.

```
max-comm : ∀ {i} n m → [ i ] max n m ∼ max m n
max-comm zero zero = zero
max-comm zero (suc m) = suc λ { .force → symmetric-∼ (max-right-identity (force m)) }
max-comm (suc n) zero = suc λ { .force → max-right-identity (force n) }
max-comm (suc n) (suc m) = suc λ { .force → max-comm (force n) (force m) }
```

Queda por demostrar únicamente la ley de intercambio. Esta está representada por la función interchange, cuya definición se explicará caso por caso. Para poder realizar esta prueba se requieren algunos lemas extra además de los que fueron demostrados o mencionados anteriormente. Se lista a continuación el tipo de cada uno de estos lemas para que la prueba pueda comprenderse, las demostraciones de los mismos se encuentran en el apéndice A.3 para no extender demasiado esta sección.

- +-comm : ∀ m {n i} → [i] m + n ∼ n + m prueba la conmutatividad de la suma.
- _+-cong_ : ∀ {i m_1 m_2 n_1 n_2} → [i] m_1 ∼ m_2 → [i] n_1 ∼ n_2 → [i] m_1 + n_1 ∼ m_2 + n_2, análogamente a _max-cong_, prueba que la suma es compatible con la bisemejanza.
- max≤+ : ∀ {i m n} → [i] max m n ≤ m + n prueba que el máximo de dos conúmeros es menor o igual a su suma.

El tipo de la función interchange postula que, dados cuatro conúmeros a, b, c y d, el máximo de las sumas (a + b) y (c + d) es menor o igual a la suma de los máximos (max a c) y (max b d). Intuitivamente tiene sentido ya que en el primer caso se toma la suma que sea más grande pero, en el segundo, se toman los máximos sumandos para cada lado de la suma, permitiendo combinar el primer sumando más grande con el mayor segundo sumando aunque estos estén en distintas sumas en el lado izquierdo de la desigualdad. Este razonamiento fue el que llevó a elegir la orientación de la desigualdad en la formalización tanto de los monoides concurrentes como las mónadas concurrentes.

```
interchange : ∀ {i} a b c d → [ i ] max (a + b) (c + d) ≤ (max a c) + (max b d)
```

- El primer caso a analizar es aquel en el que tanto a como c son zero. b y d pueden ser cualquier conúmero.

```
interchange {i} zero b zero d = reflexive-≤ (max b d)
```

Por definición se tiene que zero + b = b y zero + d = d. Luego el lado izquierdo se reduce a max b d. Por otro lado, max zero zero = zero y, por lo tanto, max zero zero + max b d = max b d. Como el resultado a ambos lados es el mismo, el primer caso se prueba de forma trivial utilizando la reflexividad de la bisemejanza.

- El segundo caso es el primero de varios en los que se considera que a = zero y el tercer argumento es (suc c). En este en particular se considerará que b también es zero mientras que el último es un d cualquiera.

```
interchange {i} zero zero (suc c) d = reflexive-≤ (suc c + d)
```

En el lado izquierdo la suma de a y b da como resultado zero y entonces se reduce a max zero ((suc c) + d) = (suc c) + d. En el lado derecho se tiene que max zero (suc c) = (suc c) y max zero d = d, por lo que también reduce a (suc c) + d. Por lo tanto, este caso también es trivial.

- El tercer caso que se considera sigue con a = zero y el tercer argumento como sucesor de un conúmero: suc c. Ahora el segundo ya no es zero sino que es también un sucesor, suc b, y se toma d = zero.

```
interchange {i} zero (suc b) (suc c) zero =
          suc λ { .force → (transitive-≤ (∼→≤ (reflexive-∼ (force b)
                                                max-cong +-right-identity (force c)))
                                 (transitive-≤ (max≤+ {_} {force b} {force c})
                                 (transitive-≤ (∼→≤ (+-comm (force b)))
                                               ((reflexive-≤ (force c)) +-mono
                                               transitive-≤ (≤suc {_} {b})
                                                 (suc λ { .force → ∼→≤ (symmetric-∼
                                                                (max-right-identity (force b)))
                                                                                  })) ))) }
```

El lado izquierdo de la desigualdad a demostrar en este caso puede reducirse siguiendo el siguiente análisis:

```
  max (zero + suc b) (suc c + zero)
= max (suc b) (suc λ { .force → force c + zero })
= suc λ { .force → max (force b) (pred (suc λ { .force → force c + zero })) }
= suc λ { .force → max (force b) (force c + zero) }
```

En el lado derecho, por otro lado, se puede realizar la siguiente reducción:

```
                                        (max zero (suc c)) + (max (suc b) zero)
                                (suc c) + (suc λ { .force → max (force b) (pred zero) }) =
                        suc λ { .force → (force c) + (suc λ { .force → max (force b) zero }) } =
```

Quitando de ambos la construcción suc λ { .force → ... } de más afuera al ponerla en la prueba, el objetivo a demostrar es:

```
[ j ] max (force b) (force c + zero) ≤ (force c) + (suc λ { .force → max (force b) zero })
```

donde j es menor a i.

El primer paso de la prueba se realiza a partir del término (reflexive-∼ (force b) max-cong +-right-identity (force c)) que prueba que

```
[ j ] max (force b) (force c + zero) ∼ max (force b) (force c)
```

Luego el lema ∼→≤ lo convierte en desigualdad.

El siguiente paso utiliza el lema max≤+ aplicado a (force b) y (force c), demostrando que:

```
[ j ] max (force b) (force c) ≤ (force b) + (force c)
```

Utilizando la conmutatividad de la suma, que puede convertirse en desigualdad al aplicar el lema ∼→≤, se prueba que:

$$[\ j\ ]\ (\text{force}\ b)\ +\ (\text{force}\ c)\ \leq\ (\text{force}\ c)\ +\ (\text{force}\ b)$$

En el último paso se utiliza el lema +-mono para resolver la prueba sumando a sumando. De un lado se pasa como argumento la prueba (reflexive-≤ (force c)) para que el lado izquierdo de la suma quede igual. El segundo lado es el más interesante. El objetivo es ver que:

$$[\ j\ ]\ (\text{force}\ b)\ \leq\ (\text{suc}\ \lambda\ \{\ .\text{force}\ \rightarrow\ \text{max}\ (\text{force}\ b)\ \text{zero}\ \})$$

Primero se utiliza el lema ≤suc para probar que [j] force $b \leq$ suc b, el nuevo objetivo a demostrar es entonces:

$$[\ j\ ]\ (\text{suc}\ b)\ \leq\ (\text{suc}\ \lambda\ \{\ .\text{force}\ \rightarrow\ \text{max}\ (\text{force}\ b)\ \text{zero}\ \})$$

Al quitar el constructor suc de ambos lados, queda por probar simplemente que, para un k menor a j, [k] (force b) ≤ max (force b) zero. Esto se obtiene utilizando el lema max-right-identity (dado vuelta con symmetric-∼ y convertido en desigualdad por ∼→≤).

- En el cuarto caso se conserva que a = zero y el segundo y el tercer argumento son sucesores de un conúmero: (suc b) y (suc c). El último, que antes era zero ahora es también un sucesor: (suc d).

```
interchange {i} zero (suc b) (suc c) (suc d) =
          suc λ { .force → transitive-≤ (interchange zero (force b) (force c) (suc d))
                           ((reflexive-≤ (force c)) +-mono
                           transitive-≤ (∼→≤ (max-comm (force b) (suc d)))
                             (suc λ { .force → transitive-≤
                                               (∼→≤ (max-comm (force d) (pred (force b))))
                                               (pred≤ max-mono reflexive-≤ (force d)) })) }
```

El lado izquierdo de la desigualdad se puede reducir en este caso de la siguiente manera:

```
  max (zero + suc b) (suc c + suc d)
= max (suc b) (suc λ { .force → force c + suc d })
= suc λ { .force → max (force b) (pred (suc λ { .force → force c + suc d })) }
= suc λ { .force → max (force b) (force c + suc d) }
```

El lado derecho puede reducirse, por su parte, siguiendo los siguientes pasos:

```
                                  (max zero (suc c)) + (max (suc b) (suc d))
                          (suc c) + (suc λ { .force → max (force b) (pred (suc d)) }) =
suc λ { .force → (force c) + (suc λ { .force → max (force b) (force d) }) } =
```

Quitando de ambos lados la estructura suc λ { .force → ... }, el objetivo a demostrar queda, para un j menor a i, como sigue:

$$[\ j\ ]\ \text{max}\ (\text{force}\ b)\ (\text{force}\ c\ +\ \text{suc}\ d)\ \leq\ (\text{force}\ c)\ +\ (\text{suc}\ \lambda\ \{\ .\text{force}\ \rightarrow\ \text{max}\ (\text{force}\ b)\ (\text{force}\ d)\ \})$$

Aplicando la llamada co-recursiva (interchange zero (force b) (force c) (suc d)), se obtiene que:

$$[\ j\ ]\ \text{max}\ (\text{force}\ b)\ (\text{force}\ c\ +\ \text{suc}\ d)\ \leq\ (\text{force}\ c)\ +\ (\text{max}\ (\text{force}\ b)\ (\text{suc}\ d))$$

El nuevo objetivo a demostrar es entonces:

$$[\ j\ ]\ (\mathsf{force}\ c)\ +\ (\mathsf{max}\ (\mathsf{force}\ b)\ (\mathsf{suc}\ d)) \leq (\mathsf{force}\ c)\ +\ (\mathsf{suc}\ \lambda\ \{\ .\mathsf{force} \to \mathsf{max}\ (\mathsf{force}\ b)\ (\mathsf{force}\ d)\ \})$$

Como se puede observar, en ambos lados hay sumas y, además, los primeros sumandos de las mismas son iguales. Se aplica por lo tanto reflexive-$\leq$ (force c) +-mono, de manera que lo que queda por probar es la desigualdad entre los segundos sumandos de cada uno de los lados. El objetivo es ahora:

$$[\ j\ ]\ \mathsf{max}\ (\mathsf{force}\ b)\ (\mathsf{suc}\ d) \leq \mathsf{suc}\ \lambda\ \{\ .\mathsf{force} \to \mathsf{max}\ (\mathsf{force}\ b)\ (\mathsf{force}\ d)\ \}$$

El primer paso para demostrarlo es aplicar ($\sim\!\!\to\!\!\leq$ (max-comm (force b) (suc d))), lo cual cambia el orden de los argumentos del operador max de manera que pueda evaluarse tal operación, obteniéndose max (suc d) (force b). Esta expresión se reduce, por definición, a (suc λ { .force $\to$ max (force d) (pred (force b)) }) Luego de aplicar esto, el objetivo a demostrar es:

$$[\ j\ ]\ \mathsf{suc}\ \lambda\ \{\ .\mathsf{force} \to \mathsf{max}\ (\mathsf{force}\ d)\ (\mathsf{pred}\ (\mathsf{force}\ b))\ \} \leq \mathsf{suc}\ \lambda\ \{\ .\mathsf{force} \to \mathsf{max}\ (\mathsf{force}\ b)\ (\mathsf{force}\ d)\ \}$$

Se quita a ambos lados el constructor suc y el nuevo objetivo es, para un k menor a j:

$$[\ k\ ]\ \mathsf{max}\ (\mathsf{force}\ d)\ (\mathsf{pred}\ (\mathsf{force}\ b)) \leq \mathsf{max}\ (\mathsf{force}\ b)\ (\mathsf{force}\ d)$$

Ambas expresiones son similares, sólo cambia el orden de los argumentos y que (force b) a la izquierda tiene aplicada la función predecesor. Para probar esta desigualdad se realizan dos pasos. El primero es aplicar ($\sim\!\!\to\!\!\leq$ (max-comm (force d) (pred (force b)))) de manera que el orden de los argumentos vuelva a como era inicialmente, obteniéndose max (pred (force b)) (force d). Luego se aplica el lema max-mono con pred$\leq$ a la izquierda que prueba que (pred (force b)) es menor o igual a (force b) y a la derecha reflexive-$\leq$ (force d) que prueba que (force d) es menor o igual a sí mismo.

- En el quinto caso se comienza a considerar que el primer argumento es un sucesor, (suc a), puesto que todos los casos en los cuales el primer argumento es zero ya fueron analizados. En este y los dos casos siguientes se toma c = zero. En este en particular el segundo argumento es un conúmero cualquiera b y el último es zero.

```
interchange {i} (suc a) b zero zero =
        suc λ { .force → ∼→≤ (transitive-∼ (max-right-identity (force a + b))
                                           ((symmetric-∼ (max-right-identity (force a)))
                                           +-cong
                                           (symmetric-∼ (max-right-identity b)))) }
```

Al analizar el lado izquierdo de la desigualdad en este caso, se obtiene lo siguiente:

$$\begin{aligned}
&\mathsf{max}\ (\mathsf{suc}\ a\ +\ b)\ (\mathsf{zero}\ +\ \mathsf{zero}) \\
=\ &\mathsf{max}\ (\mathsf{suc}\ \lambda\ \{\ .\mathsf{force} \to \mathsf{force}\ a\ +\ b\ \})\ \mathsf{zero} \\
=\ &\mathsf{suc}\ \lambda\ \{\ .\mathsf{force} \to \mathsf{max}\ (\mathsf{force}\ \lambda\ \{\ .\mathsf{force} \to \mathsf{force}\ a\ +\ b\ \})\ (\mathsf{pred}\ \mathsf{zero})\ \} \\
=\ &\mathsf{suc}\ \lambda\ \{\ .\mathsf{force} \to \mathsf{max}\ (\mathsf{force}\ a\ +\ b)\ \mathsf{zero}\ \}
\end{aligned}$$

Por otro lado, el lado derecho puede reducirse de la siguiente manera:

$$\begin{aligned}
&(\mathsf{max}\ (\mathsf{suc}\ a)\ \mathsf{zero})\ +\ (\mathsf{max}\ b\ \mathsf{zero}) \\
&(\mathsf{suc}\ \lambda\ \{\ .\mathsf{force} \to \mathsf{max}\ (\mathsf{force}\ a)\ (\mathsf{pred}\ \mathsf{zero})\ \})\ +\ (\mathsf{max}\ b\ \mathsf{zero}) = \\
&\mathsf{suc}\ \lambda\ \{\ .\mathsf{force} \to (\mathsf{force}\ \lambda\ \{\ .\mathsf{force} \to \mathsf{max}\ (\mathsf{force}\ a)\ \mathsf{zero}\ \})\ +\ (\mathsf{max}\ b\ \mathsf{zero})\ \} = \\
&\mathsf{suc}\ \lambda\ \{\ .\mathsf{force} \to (\mathsf{max}\ (\mathsf{force}\ a)\ \mathsf{zero})\ +\ (\mathsf{max}\ b\ \mathsf{zero})\ \} =
\end{aligned}$$

Como en ambos lados hay un constructor suc, este puede quitarse en cada lado. Luego de hacerlo el objetivo a demostrar es, para un j menor a i:

$$[\ j\ ]\ \mathsf{max}\ (\mathsf{force}\ a\ +\ b)\ \mathsf{zero} \leq (\mathsf{max}\ (\mathsf{force}\ a)\ \mathsf{zero}\ )\ +\ (\mathsf{max}\ b\ \mathsf{zero})$$

Ambos lados de esta desigualdad son, en realidad, iguales. Se demuestra entonces la igualdad $[\ j\ ]\ \mathsf{max}\ (\mathsf{force}\ a\ +\ b)\ \mathsf{zero} \sim (\mathsf{max}\ (\mathsf{force}\ a)\ \mathsf{zero})\ +\ (\mathsf{max}\ b\ \mathsf{zero})$ y luego se utiliza el lema $\sim\!\!\rightarrow\!\!\leq$ para obtener la desigualdad. El primer paso para demostrar la igualdad es reducir la expresión max en el lado izquierdo de la misma. Para esto se utiliza el lema max-right-identity aplicado a (force a + b) que postula que el resultado de calcular el máximo de un conúmero y zero es igual al primer conúmero. Luego de aplicar este lema se obtiene la siguiente proposición a demostrar:

$$[\ j\ ]\ (\mathsf{force}\ a)\ +\ b \sim (\mathsf{max}\ (\mathsf{force}\ a)\ \mathsf{zero}\ )\ +\ (\mathsf{max}\ b\ \mathsf{zero})$$

Es preciso notar que a ambos lados hay una suma y que los sumandos son iguales salvo que a la derecha están dentro de una aplicación de max con zero. Se debe utilizar entonces nuevamente el lema max-right-identity a ambos lados de la suma. Las dos aplicaciones de este lema, una a (force a) y la otra a b, se unen con el lema +-cong que establece que las sumas de sumandos iguales dan resultados iguales.

- En el sexto caso se sigue tomando el primer argumento como sucesor, (suc a), y el tercero como zero. El último argumento ya no es zero sino que pasa a ser sucesor de un conúmero: (suc d). Por último, se toma b = zero.

```
interchange {i} (suc a) zero zero (suc d) =
          suc λ { .force → transitive-≤ (∼→≤ ((+-right-identity (force a))
                                              max-cong
                                              (reflexive-∼ (force d))))
                                (transitive-≤ (max≤+ {_} {force a})
                                              (ˡ≤max (force a) zero +-mono ≤suc)) }
```

El análisis del lado izquierdo de la desigualdad en este caso queda como sigue:

```
  max (suc a + zero) (zero + suc d)
= max (suc λ { .force → force a + zero }) (suc d)
= suc λ { .force → max (force λ { .force → force a + zero }) (pred (suc d)) }
= suc λ { .force → max (force a + zero) (force d) }
```

El lado derecho, por su parte, queda de la siguiente manera:

```
                                                (max (suc a) zero) + (max zero (suc d))
                                  (suc λ { .force → max (force a) (pred zero) }) + (suc d) =
suc λ { .force → (force λ { .force → max (force a) (pred zero) }) + (suc d) } =
                                  suc λ { .force → (max (force a) zero) + (suc d) } =
```

Al igual que en los casos anteriores se quita el constructor suc a cada lado y el objetivo a demostrar pasa ser, para un j menor a i:

$$[\ j\ ]\ \mathsf{max}\ (\mathsf{force}\ a\ +\ \mathsf{zero})\ (\mathsf{force}\ d) \leq (\mathsf{max}\ (\mathsf{force}\ a)\ \mathsf{zero})\ +\ (\mathsf{suc}\ d)$$

Primero se resuelve la suma (force a + zero) utilizando el lema +-right-identity que establece la identidad derecha del zero respecto de la suma. Como el término a resolver está dentro de una

aplicación de max, se utiliza el lema max-cong con (+-right-identity (force a)) de un lado para que se resuelva la suma en cuestión y (reflexive-$\sim$ (force d)) del otro para que este permanezca igual. Como max-cong prueba una igualdad, se utiliza luego el lema $\sim\!\!\rightarrow\!\!\leq$ para obtener la desigualdad. Después de esta aplicación el nuevo objetivo a demostrar es el siguiente:

$$[\ j\ ]\ \mathsf{max}\ (\mathsf{force}\ a)\ (\mathsf{force}\ d) \leq (\mathsf{max}\ (\mathsf{force}\ a)\ \mathsf{zero}) + (\mathsf{suc}\ d)$$

De un lado de la desigualdad hay una operación max y, del otro, una suma. Como es sabido que el máximo de dos conúmeros es menor o igual a su suma, se utiliza el lema max$\leq$+ para convertir el lado izquierdo en una suma, de manera que pueda trabajarse luego sumando a sumando. Luego de aplicar este lema queda por demostrar que:

$$[\ j\ ]\ (\mathsf{force}\ a) + (\mathsf{force}\ d) \leq (\mathsf{max}\ (\mathsf{force}\ a)\ \mathsf{zero}) + (\mathsf{suc}\ d)$$

Esta última demostración se realiza, como se adelantó previamente, sumando a sumando, valiéndose para ello del lema +-mono que postula la compatibilidad de la suma con la relación de orden. Para probar en el lado izquierdo que [j] (force a) $\leq$ (max (force a) zero), se utiliza el lema $^{l}\!\leq$max aplicado a (force a) y zero que prueba que un conúmero siempre es menor o igual al máximo de sí mismo con otro. En el lado derecho, por otra parte, se debe probar lo siguiente: [j] (force d) $\leq$ (suc d). Esto se resuelve mediante la utilización del lema $\leq$suc que prueba que todo conúmero es menor o igual al su sucesor.

- En el penúltimo caso se conserva que el primer y el último argumento son sucesores, (suc a) y (suc d), y que el tercer argumento es zero. Lo que cambia es que el segundo argumento ya no es zero sino que es (suc b).

```
interchange {i} (suc a) (suc b) zero (suc d) =
        suc λ { .force → transitive-≤ (interchange (force a) (suc b) zero (force d))
                                      ((reflexive-≤ (max (force a) zero))
                                      +-mono
                                      suc λ { .force → (reflexive-≤ (force b))
                                                       max-mono pred≤ }) }
```

Se analiza el lado izquierdo de la desigualdad de la siguiente manera:

```
   max (suc a + suc b) (zero + suc d)
= max (suc λ { .force → force a + suc b }) (suc d)
= suc λ { .force → max (force λ { .force → force a + suc b }) (pred (suc d)) }
= suc λ { .force → max (force a + suc b) (force d) }
```

Luego se analiza el lado derecho como se muestra a continuación:

```
                                                (max (suc a) zero) + (max (suc b) (suc d))
                                  (suc λ { .force → max (force a) (pred zero) }) + (max (suc b) (suc d)) =
                                        (suc λ { .force → max (force a) zero }) + (max (suc b) (suc d)) =
(suc λ { .force → max (force a) zero }) + (suc λ { .force → max (force b) (pred (suc d)) }) =
      (suc λ { .force → max (force a) zero }) + (suc λ { .force → max (force b) (force d) }) =
      suc λ { .force → (max (force a) zero) + (suc λ { .force → max (force b) (force d) }) } =
```

Si se quita el constructor suc de ambos términos se obtiene que, para un j menor a i, el objetivo a demostrar es el siguiente:

$$[\ j\ ]\ \mathsf{max}\ (\mathsf{force}\ a + \mathsf{suc}\ b)\ (\mathsf{force}\ d) \leq \\ (\mathsf{max}\ (\mathsf{force}\ a)\ \mathsf{zero}) + (\mathsf{suc}\ \lambda\ \{\ .\mathsf{force} \to \mathsf{max}\ (\mathsf{force}\ b)\ (\mathsf{force}\ d)\ \})$$

Se aplica la llamada co-recursiva (interchange (force a) (suc b) zero (force d)) como primer paso para realizar esta prueba. Esta llamada prueba lo siguiente:

$$[\ j\ ]\ \mathsf{max}\ (\mathsf{force}\ a + \mathsf{suc}\ b)\ (\mathsf{force}\ d) \leq (\mathsf{max}\ (\mathsf{force}\ a)\ \mathsf{zero}) + (\mathsf{max}\ (\mathsf{suc}\ b)\ (\mathsf{force}\ d))$$

El lado derecho de la desigualdad se reescribe, por definición de la operación max, de la siguiente manera: (max (force a) zero) + (suc λ { .force → max (force b) (pred (force d)) }). Por lo tanto, después de aplicar la llamada co-recursiva, el nuevo objetivo a demostrar es:

$$[\ j\ ]\ (\mathsf{max}\ (\mathsf{force}\ a)\ \mathsf{zero}) + (\mathsf{suc}\ \lambda\ \{\ .\mathsf{force} \to \mathsf{max}\ (\mathsf{force}\ b)\ (\mathsf{pred}\ (\mathsf{force}\ d))\ \}) \leq \\ (\mathsf{max}\ (\mathsf{force}\ a)\ \mathsf{zero}) + (\mathsf{suc}\ \lambda\ \{\ .\mathsf{force} \to \mathsf{max}\ (\mathsf{force}\ b)\ (\mathsf{force}\ d)\ \})$$

Es fácil observar que a ambos lados se cuenta ahora con una suma, donde los primeros sumandos son iguales y los segundos sumandos son bastante similares. Se resuelve entonces esta nueva desigualdad sumando a sumando utilizando el lema +-mono, pasándole como primer argumento (reflexive-≤ (max (force a) zero)) que prueba que (max (force a) zero) es menor o igual a sí mismo. Como segundo argumento se deberá dar un término que demuestre que:

$$[\ j\ ]\ (\mathsf{suc}\ \lambda\ \{\ .\mathsf{force} \to \mathsf{max}\ (\mathsf{force}\ b)\ (\mathsf{pred}\ (\mathsf{force}\ d))\ \}) \leq \\ (\mathsf{suc}\ \lambda\ \{\ .\mathsf{force} \to \mathsf{max}\ (\mathsf{force}\ b)\ (\mathsf{force}\ d)\ \})$$

Lo primero que hay que hacer para demostrarlo es quitar a ambos lados el constructor suc. Luego de hacerlo, lo que queda por demostrar es, para un k menor a j:

$$[\ k\ ]\ \mathsf{max}\ (\mathsf{force}\ b)\ (\mathsf{pred}\ (\mathsf{force}\ d)) \leq \mathsf{max}\ (\mathsf{force}\ b)\ (\mathsf{force}\ d)$$

En este caso se tiene a ambos lados una operación max, donde los primeros argumentos son iguales entre sí y los segundos sólo varían en una aplicación de la función predecesor. Luego se prueba esta desigualdad mediante el lema max-mono. Como primer argumento a este lema se pasa el término (reflexive-≤ (force b)) que prueba que (force b) es menor o igual a sí mismo. Como segundo argumento se utiliza el lema pred≤ que prueba que el predecesor de un conúmero es menor o igual a dicho conúmero para probar que [k] (pred (force d)) ≤ (force d).

- Habiendo analizado todos los casos en los cuales el primer argumento es (suc a) y el tercero es zero, el octavo y último caso considera que el primer y el tercer argumento son sucesores, (suc a) y (suc c), y los otros dos son conúmeros b y d cualesquiera.

$$\mathsf{interchange}\ \{i\}\ (\mathsf{suc}\ a)\ b\ (\mathsf{suc}\ c)\ d = \mathsf{suc}\ \lambda\ \{\ .\mathsf{force} \to \mathsf{interchange}\ (\mathsf{force}\ a)\ b\ (\mathsf{force}\ c)\ d\ \}$$

Al analizar el lado izquierdo de la desigualdad, se obtiene lo siguiente:

$$\begin{aligned} &\mathsf{max}\ (\mathsf{suc}\ a + b)\ (\mathsf{suc}\ c + d) \\ &= \mathsf{max}\ (\mathsf{suc}\ \lambda\ \{\ .\mathsf{force} \to \mathsf{force}\ a + b\ \})\ (\mathsf{suc}\ \lambda\ \{\ .\mathsf{force} \to \mathsf{force}\ c + d\ \}) \\ &= \mathsf{suc}\ \lambda\ \{\ .\mathsf{force} \to \mathsf{max}\ (\mathsf{force}\ \lambda\ \{\ .\mathsf{force} \to \mathsf{force}\ a + b\ \})\ (\mathsf{pred}\ (\mathsf{suc}\ \lambda\ \{\ .\mathsf{force} \to \mathsf{force}\ c + d\ \}))\ \} \\ &= \mathsf{suc}\ \lambda\ \{\ .\mathsf{force} \to \mathsf{max}\ (\mathsf{force}\ a + b)\ (\mathsf{force}\ c + d)\ \} \end{aligned}$$

Cuando se analiza el lado derecho, por otro lado, se obtiene:

$$(\mathsf{max}\ (\mathsf{suc}\ a)\ (\mathsf{suc}\ c)) + (\mathsf{max}\ b\ d)$$

```
(suc λ { .force → max (force a) (pred (suc c)) }) + (max b d) =
(suc λ { .force → max (force a) (force c) }) + (max b d) =
suc λ { .force → (force λ { .force → max (force a) (force c) }) (max b d) } =
suc λ { .force → (max (force a) (force c)) (max b d) } =
```

Quitando de ambos términos el constructor suc, se obtiene que el objetivo a demostrar es, para un j menor a i:

```
[ j ] max (force a + b) (force c + d) ≤ (max (force a) (force c)) (max b d)
```

Esta desigualdad es, exactamente, el resultado que se obtiene al realizar la siguiente llamada co-recursiva: interchange (force a) b (force c) d.

De esta manera queda demostrada la ley de intercambio para los números conaturales.

Para poder crear la instancia de ConcurrentMonoid para Conat ∞, faltan sólo dos ingredientes. Aunque ya se demostraron las propiedades necesarias para ver que la relación de bisemejanza es una relación de equivalencia y que el orden es un orden parcial respecto de la misma, no se dieron las pruebas completas que agrupan estas propiedades. A continuación se muestran entonces las definiciones de los lemas eq-∼ y partial-≤ que prueban que la relación [∞]_∼_ es una relación de equivalencia y que la misma relación junto con [∞]_≤_ forman un orden parcial, respectivamente.

```
eq-∼ : IsEquivalence ([ ∞ ]_∼_)
eq-∼ = record { refl = reflexive-∼ _
              ; sym = symmetric-∼
              ; trans = transitive-∼ }

partial-≤ : IsPartialOrder ([ ∞ ]_∼_) ([ ∞ ]_≤_)
partial-≤ = record { isPreorder =
                     record { isEquivalence = eq-∼
                            ; reflexive = ∼→≤
                            ; trans = transitive-≤ }
                   ; antisym = antisymmetric-≤ }
```

Finalmente, se puede construir la instancia de ConcurrentMonoid para el tipo Conat ∞, demostrando que los números conaturales forman un monoide concurrente.

```
open import Structures.ConcurrentMonoid

conatConcurrent : ConcurrentMonoid (Conat ∞)
conatConcurrent = makeConcurrentMonoid
                    ([ ∞ ]_∼_)
                    eq-∼
                    ([ ∞ ]_≤_)
                    partial-≤
                    zero
                    _+_
                    (λ x y z w → _+-mono_ {∞} {x} {z} {y} {w})
                    +-left-identity
                    +-right-identity
```

```
(λ x y z → +-assoc x {y} {z} {∞})
max
(λ x y z w → _max-mono_ {∞} {x} {z} {y} {w})
max-left-identity
max-right-identity
max-assoc
max-comm
interchange
```

6.6.3. Una mónada concurrente alternativa a *delay*

Como se mencionó anteriormente, se puede pensar a los conaturales como una versión simplificada del tipo *delay* en donde no hay valores de retorno (o bien el tipo de retorno es un tipo con un único habitante y el valor de retorno es siempre el mismo). Una forma de generar una alternativa similar a la mónada *delay* es, por lo tanto, agregar a los conaturales un valor de retorno mediante la utilización del producto cartesiano. Un par $(n\ ,\ a)$ es similar a un habitante del tipo *delay* que tiene n constructores later y retorna el valor a. La principal diferencia entre el tipo $A\ \bot$ y el tipo Conat $\infty\ \times\ A$ es que, mientras en el primero existe un único término infinito que jamás retorna ningún valor (never), en el segundo todos los términos tienen un valor de retorno. En consecuencia, existen muchos habitantes de Conat $\infty\ \times\ A$ que nunca terminan: para cada valor $a : A$, existe (inf , a). Esta diferencia hace que no sean equivalentes y no puedan utilizarse para los mismos fines, pero de igual manera son bastante similares y en algunos casos puede ser de utilidad.

El lema 3.32 postula que si $(A, \sqsubseteq, *, \texttt{nothing}, ;, \texttt{skip})$ con $\texttt{nothing} = \texttt{skip}$ es un monoide concurrente, luego el funtor $T_A\ X = A \times X$ constituye una mónada concurrente. El objetivo de esta sección es demostrar este lema y luego concluir que la mónada *writer* para conaturales, Conat $\infty\ \times\ A$, es una mónada concurrente.

Lo primero que es necesario definir para poder realizar la demostración del lema es el funtor T_A. La función F que toma dos conjuntos y devuelve su producto cartesiano representa dicho funtor.

```
F : Set → Set → Set
F S X = S × X
```

Las definiciones de cada uno de los campos de ConcurrentMonad se realizarán dentro de un módulo anónimo que tiene como parámetros implícitos un conjunto S y una prueba *cmonoid* de que tal conjunto es un monoide concurrente. El objetivo de la utilización de este módulo es fijar el conjunto S y abrir la prueba de que es un monoide concurrente para poder utilizar sus campos en las definiciones a realizar. Se define el módulo como se muestra a continuación y se abre ConcurrentMonoid $\{S\}$ *cmonoid* renombrando cada campo para que no se generen conflictos de nombre. De igual manera se abren las pruebas de equivalencia, orden parcial y preorden para poder acceder fácilmente a sus propiedades internas.

```
module _ {S : Set} {cmonoid : ConcurrentMonoid S} where

  open ConcurrentMonoid {S} cmonoid
    renaming (_≅ₘ_ to _≅_ ; _≲ₘ_ to _≲_ ; zeroₘ to zero ; _+ₘ_ to _+_)
    renaming (eqₘ to eq ; sidl to +idl ; sidr to +idr ; sassoc to +assoc)
```

```
    renaming (max_m to max ; porder_m to porder ; midr to maxidr ; midl to maxidl)
    renaming (scomp≲_m to scomp≲ ; mcomp≲_m to mcomp≲ ; mcomm to maxcomm)
    renaming (massoc to maxassoc ; ichange to interchange)

open IsEquivalence eq renaming (refl to refl_s ; trans to trans_s ; sym to sym_s)

open IsPartialOrder porder renaming (isPreorder to preorder_s ; antisym to antisym_s)

open IsPreorder preorder_s renaming (reflexive to ≤refl_s ; trans to ≤trans_s)
```

Se define primero la relación de igualdad entre habitantes de F S A para un A arbitrario. Para los habitantes de A se utilizará la igualdad proposicional, cuyas propiedades refl, sym y trans fueron renombradas con una letra p delante para hacer referencia a que corresponden a la igualdad proposicional. Para que dos habitantes de F S A sean iguales según la relación _F≈_, las primeras componentes de cada par deben ser iguales según la igualdad extraída de *cmonoid* y las segundas deben iguales según la igualdad proposicional. Utilizando las propiedades de reflexividad, simetría y transitividad tanto de la igualdad del monoide concurrente como de la igualdad proposicional, se demuestra que _F≈_ es una relación de equivalencia.

```
_F≈_ : ∀ {A : Set} → F S A → F S A → Set
_F≈_ (n , a) (m , a') = (n ≅ m) × a ≡ a'

eqF≈ : ∀ {A} → IsEquivalence (_F≈_ {A})
eqF≈ = record { refl = refl_s , prefl ;
                sym = λ {(n≅m , a≡b) → (sym_s n≅m) , psym a≡b} ;
                trans = λ {(n≅m , a≡b) (m≅o , b≡c)
                           → trans_s n≅m m≅o , ptrans a≡b b≡c}
        }
```

Luego se define la relación de orden entre habitantes de F S A, también para un conjunto A arbitrario. En este caso para que $(n\ ,\ a)$ sea menor o igual que $(m\ ,\ a')$, se debe cumplir que n es menor o igual a m según la relación de orden del monoide concurrente y que, al igual que antes, $a \equiv a'$. Una vez más se utilizan las propiedades de la relación de orden del monoide concurrente y las propiedades de la igualdad proposicional para demostrar que _F≤_ es un orden parcial respecto de _F≈_.

```
_F≤_ : ∀ {A : Set} → F S A → F S A → Set
_F≤_ (n , a) (m , a') = n ≲ m × a ≡ a'

porderF≤ : ∀ {A} → IsPartialOrder (_F≈_ {A}) (_F≤_ {A})
porderF≤ = record
           { isPreorder =
             record { isEquivalence = eqF≈
                    ; reflexive = λ {(n≤m , a≡b) → ≤refl_s n≤m , a≡b}
                    ; trans = λ {(n≤m , a≡b) (m≤o , b≡c)
                                 → (≤trans_s n≤m m≤o) , ptrans a≡b b≡c} }
           ; antisym = λ {(n≤m , a≡b) (m≤n , b≡a) → antisym_s n≤m m≤n , a≡b}
           }
```

Habiendo definido ambas relaciones, se pasa a definir la estructura monádica. La función return toma un elemento x de A y lo encapsula en un habitante de F S A. Esto se realiza

generando el par que contiene a x y, en la primera componente, el elemento neutro zero del monoide concurrente. El operador bind, por su lado, toma un elemento $(n\ ,\ a)$ de F S A y una función f de tipo $A \to$ F S B, evalúa la función f en el valor de retorno a y luego, a partir del resultado obtenido $(m\ ,\ b)$, genera un par de tipo F S B conformado por la suma de n y m (operación del monoide concurrente) y el valor de retorno b obtenido de calcular $f\ a$.

```
return : ∀ {A : Set} → A → F S A
return x = (zero , x)

bind : ∀ {A B : Set} → F S A → (A → F S B) → F S B
bind (n , a) f with f a
...                 | m , b = (n + m) , b
```

Para ver que bind es compatible con la relación de orden definida se debe probar que, dados x_1 y x_2 de tipo F S A y dos funciones f_1 $f_2 : A \to$ F S B tales que se tienen pruebas de que x_1 F$\leq$ x_2 y $\forall\ (a : A) \to f_1\ a$ F$\leq$ $f_2\ a$, se cumple que bind x_1 f_1 F$\leq$ bind x_2 f_2. Para demostrarlo se realiza *pattern matching* sobre x_1, x_2 y la prueba de que x_1 F$\leq$ x_2. Al hacer esto se unifican las segundas componentes de x_1 y x_2 que son iguales. Luego las funciones f_1 y f_2 se aplican sobre el mismo valor a_1 y se puede utilizar el argumento $f\leq$. La prueba $f\leq$ aplicada a a_1 prueba que $f_1\ a_1$ F$\leq$ $f_2\ a_1$. Su primera componente prueba entonces que $(\text{proj}_1\ (f_1\ a_1)) \lesssim (\text{proj}_1\ (f_2\ a_1))$ y su segunda componente demuestra que $(\text{proj}_2\ (f_1\ a_1)) \equiv (\text{proj}_2\ (f_2\ a_1))$.

```
bind-comp : ∀ {A B : Set} → (x₁ x₂ : F S A) (f₁ f₂ : A → F S B) →
              x₁ F≤ x₂ → (∀ (a : A) → f₁ a F≤ f₂ a) → bind x₁ f₁ F≤ bind x₂ f₂
bind-comp (n₁ , a₁) (n₂ , .a₁) f₁ f₂ (fst≤ , refl) f≤ =
        scomp≲ n₁ (proj₁ (f₁ a₁)) n₂ (proj₁ (f₂ a₁)) fst≤ (proj₁ (f≤ a₁)) , proj₂ (f≤ a₁)
```

En la primera componente de la prueba de bind-comp se debe dar un término que demuestre que $n_1 + (\text{proj}_1\ (f_1\ a_1)) \lesssim n_2 + (\text{proj}_1\ (f_2\ a_1))$. Para esto se utiliza la compatibilidad de la suma con la relación de orden del monoide concurrente: scomp$\lesssim$. Como prueba de que $n_1 \lesssim n_2$ se utiliza $fst\leq$, que es la primera componenete de la prueba x_1 F$\leq$ x_2. Por otro lado, para mostrar que $(\text{proj}_1\ (f_1\ a_1)) \lesssim (\text{proj}_1\ (f_2\ a_1))$ se utiliza la primera componente de la prueba $f\leq$ aplicada a a_1. Luego, en el lado derecho, hay que probar que $(\text{proj}_2\ (f_1\ a_1)) \equiv (\text{proj}_2\ (f_2\ a_1))$, para lo cual se utiliza la segunda componente de la prueba $f\leq$ aplicada también a a_1.

Para probar las leyes de las mónadas se utilizan, a la izquierda, las propiedades análogas correspondientes a la suma del monoide concurrente. Las segundas componentes se prueban trivialmente utilizando la reflexividad de la igualdad proposicional.

```
bind-left : ∀ {A B : Set} → (x : B) (f : B → F S A) → bind (return x) f F≈ f x
bind-left x f with f x
...                 | m , a = +idl m , refl

bind-right : ∀ {A : Set} → (t : F S A) → bind t return F≈ t
bind-right (n , a) = +idr n , refl

bind-assoc : ∀ {A B C : Set} (t : F S C) (f : C → F S B) (g : B → F S A) →
               bind (bind t f) g F≈ bind t (λ x → bind (f x) g)
bind-assoc (n , c) f g with f c
...                       | (m , b) with g b
...                                 | (o , a) = sym_s (+assoc n m o) , refl
```

Se introduce a continuación la estructura monoidal. Se define el valor unit únicamente para poder utilizarlo en los tipos de las propiedades que siguen, puesto que este ya está fijo en la formalizacion de mónada concurrente y no se puede modificar. La operación merge, por su lado, dados dos valores de tipo F S A y F S B, debe devolver un valor de tipo F S $(A \times B)$. Este se define como el par formado por el máximo de las primeras componentes y el par ordenado de las segundas componentes.

```
unit : F S ⊤
unit = return tt

merge : {A B : Set} → F S A → F S B → F S (A × B)
merge (n , a) (m , b) = max n m , a , b
```

A la hora de probar la compatibilidad del operador merge con la relación de orden, se tienen valores x_1 x_2 : F S A e y_1 y_2 : F S B tales que x_1 F$\leq$ x_2 e y_1 F$\leq$ y_2. Al hacer *pattern matching* sobre todos los valores y también sobre las pruebas, se unifican los valores de retorno de los dos primeros argumentos y de los dos segundos que son iguales debido a las pruebas de desigualdad.

```
merge-comp : ∀ {A B : Set} → (x₁ x₂ : F S A) (y₁ y₂ : F S B) → x₁ F≤ x₂ → y₁ F≤ y₂
           → merge x₁ y₁ F≤ merge x₂ y₂
merge-comp (n₁ , a₁) (n₂ , .a₁) (m₁ , b₁) (m₂ , .b₁) (n≤ , prefl) (m≤ , prefl)
           = (mcomp≲ n₁ m₁ n₂ m₂ n≤ m≤) , prefl
```

El objetivo es demostrar, por un lado, que max n_1 m_1 $\lesssim$ max n_2 m_2. Para esto se utiliza la compatibilidad del operador max con la relación de orden del monoide concurrente. En el lado derecho, el objetivo es demostrar que $(a_1 , b_1) \equiv (a_1 , b_1)$ puesto que se unificaron los valores. La prueba es, por lo tanto, trivial.

Las demostraciones de las propiedades del operador merge también utilizan las propiedades análogas del operador max, pero es necesario también utilizar la identidad a derecha de la suma debido a que estas propiedades están definidas utilizando el operador bind seguido de un return. Cuando se tiene una expresión del tipo bind $(n , \_)$ $(\lambda \_ \to$ return $\_)$, por definición de return, es lo mismo que tener bind $(n , \_)$ $(\lambda \_ \to$ zero $, \_)$. Luego si se aplica la definición de bind se obtiene: $(n +$ zero $, \_)$. Por lo tanto, para poder realizar pruebas sobre la primer componente de este resultado es necesario utilizar el lema +idr de manera que se pueda reducir la suma con zero.

```
merge-left : {A : Set} (a : F S A) → merge a unit F≈ bind a (λ a₁ → return (a₁ , tt))
merge-left a = trans_s (maxidr _) (sym_s (+idr _)) , prefl

merge-right : {B : Set} (b : F S B) → merge unit b F≈ bind b (λ b₁ → return (tt , b₁))
merge-right (n , a) = trans_s (maxidl n) (sym_s (+idr n)) , prefl

merge-assoc : {A B C : Set} (a : F S A) (b : F S B) (c : F S C) →
              bind (merge (merge a b) c) (λ { ((a , b) , c) → return (a , b , c) })
                                                    F≈ merge a (merge b c)
merge-assoc (n , a) (m , b) (o , c) =
            trans_s (+idr (max (max n m) o)) (maxassoc n m o) , prefl

merge-comm : {A B : Set} → (a : F S A) → (b : F S B)
           → merge a b F≈ bind (merge b a) (λ { (a , b) → zero_m cmonoid , b , a})
merge-comm (n , a) (m , b) = trans_s (maxcomm n m) (sym_s (+idr (max m n))) , prefl
```

Queda por probar únicamente la ley de intercambio. Sean $(n\ ,\ a)$ de tipo F S A y $(m\ ,\ b)$ de tipo F S B. Luego sean $f : A \to$ F S C y $g : B \to$ F S D tales que $f\ a = (o\ ,\ c)$ y $g\ b = (p\ ,\ d)$. Luego, por las definiciones de los operadores bind y merge, el objetivo a demostrar en el lado izquierdo es: max $(n + o)$ $(m + p) \lesssim$ (max n m) + (max o p). Esto se prueba utilizando la ley de intercambio del monoide concurrente. En el lado derecho, se debe demostrar que $(c\ ,\ d)$ es igual a sí mismo, lo cual se prueba trivialmente.

```
ichg : {A B C D : Set} (a : F S A) (b : F S B) (f : A → F S C) (g : B → F S D) →
  merge (bind a f) (bind b g) F≤ bind (merge a b) (λ { (a , b) → merge (f a) (g b) })
ichg (n , a) (m , b) f g with      f a     | g b
...                        |       (o , c) | (p , d) = (interchange n o m p) , prefl
```

Finalmente, la prueba de que dado un conjunto S que es un monoide concurrente se puede construir una mónada concurrente para el funtor (F S) queda como se muestra a continuación. Es preciso notar que a cada una de las definiciones dadas se les pasa como argumentos implícitos el conjunto S y la prueba *cmonoid* que son los parámetros del módulo anónimo donde se encuentran definidas. De esta manera, se utilizan los campos de ConcurrentMonoid para construir los de ConcurrentMonad.

```
cmonoid⇒cmonad : {S : Set} → ConcurrentMonoid S → ConcurrentMonad (F S)
cmonoid⇒cmonad {S} cmonoid
                = makeConcurrentMonad
                    (_F≈_       {S} {cmonoid})
                    (eqF≈       {S} {cmonoid})
                    (_F≤_       {S} {cmonoid})
                    (porderF≤   {S} {cmonoid})
                    (return     {S} {cmonoid})
                    (bind       {S} {cmonoid})
                    (bind-comp  {S} {cmonoid})
                    (bind-left  {S} {cmonoid})
                    (bind-right {S} {cmonoid})
                    (bind-assoc {S} {cmonoid})
                    (merge      {S} {cmonoid})
                    (merge-comp {S} {cmonoid})
                    (merge-left {S} {cmonoid})
                    (merge-right {S} {cmonoid})
                    (merge-assoc {S} {cmonoid})
                    (merge-comm {S} {cmonoid})
                    (ichg       {S} {cmonoid})
```

Luego, se puede definir una instancia de ConcurrentMonad para F (Conat ∞) utilizando la prueba recién definida y la instancia conatConcurrent definida en la sección anterior que prueba que Conat ∞ es un monoide concurrente.

```
writerConatConcurrent : ConcurrentMonad (F (Conat ∞))
writerConatConcurrent = cmonoid⇒cmonad conatConcurrent
```

Por lo tanto, queda demostrado que la mónada *writer* para el tipo Conat ∞ es una mónada concurrente.

Capítulo 7

Conclusiones y trabajo futuro

Para realizar esta tesina se estudió en profundidad el concepto de monoides concurrentes y mónadas concurrentes, así como también los tipos coinductivos el uso de la coinducción en general para luego comprender la definición del tipo *delay* y cómo este se utiliza para representar formalmente la no terminación de programas.

En este trabajo se utilizó el lenguaje y asistente de pruebas Agda para realizar formalizaciones de diversas estructuras algebraicas que ayudaron a construir la formalización de las mónadas concurrentes. Entre ellas se encuentran los monoides, las mónadas, los funtores monoidales y los monoides concurrentes. Posteriormente se analizó el caso de la mónada *delay* con el objetivo de demostrar que esta es una mónada concurrente. La demostración de la ley de intercambio mostró diversas complicaciones que llevaron a tomar la decisión de simplificar el problema a los números conaturales. Al intentar demostrar la misma ley para la versión reducida del problema, se presentaron dificultades en torno al soporte para coinducción elegido. Finalmente, se decidió cambiar este soporte por otro, logrando demostrar que los conaturales forman un monoide concurrente y definiendo una mónada concurrente que tiene ciertas similitudes con la mónada *delay*.

Las principales conclusiones de este trabajo son:

1. La formalización de estructuras algebraicas puede realizarse en Agda mediante la utilización de tipos record. Estos tipos admiten la definición de campos, los cuales se utilizaron para definir tanto los elementos que conforman la estructura como las propiedades que se deben cumplir para que dichos elementos efectivamente constituyan la estructura deseada. Al realizar las formalizaciones de esta manera, una instancia de uno de estos tipos no sólo define un ejemplo de la estructura formalizada, sino que también demuestra que el ejemplo definido cumple con las características necesarias para serlo.
2. Agda es un lenguaje y asistente de pruebas muy potente pero puede llegar a traer muchas complicaciones a la hora de representar la coinducción. En general, la coinducción tiene conflictos con todos los lenguajes que no permitan la no terminación de programas puesto que, como se mencionó anteriormente, las pruebas por coinducción suelen ser infinitas. Esto hace que sea difícil convencer a los asistentes de pruebas de que las demostraciones son productivas y están bien definidas.
3. El soporte para coinducción con tipos de tamaño definido (*sized types*) ayuda al chequeo de terminación de programas de Agda, permitiendo realizar un número elevado de demostraciones que con el soporte de notación musical no eran posibles. Sin embargo, puede llegar a presentar problemas para reconocer y unificar valores que son iguales.
4. Los números conaturales forman un monoide concurrente con la suma y el máximo como operaciones y el cero como elemento neutro. Esto quedó demostrado al generar la instancia de ConcurrentMonoid para el tipo Conat ∞.
5. Si se tiene un conjunto S que es un monoide concurrente, luego el funtor $T_S\ X = S \times X$ puede dotarse de una estructura de mónada concurrente. Esta implicancia se demostró en

la prueba cmonoid⇒cmonad.

6. La mónada *writer* para los números conaturales constituye una mónada concurrente. La prueba writerConatConcurrent lo demuestra utilizando las pruebas mencionadas en los dos items anteriores.

Por cuestiones de tiempo y extensión de la tesina, quedaron algunas tareas pendientes para realizar más adelante. A continuación se detallan las principales:

1. Debido a la forma en la que se realizó la formalización de las mónadas concurrentes, al generar instancias de dicha estructura uno se ve obligado a utilizar la igualdad proposicional para los tipos de retorno. A futuro podía pensarse en modificar la formalización de manera que incluya como parámetro una noción de igualdad para el tipo de retorno. Así podrían darse instancias de mónadas concurrentes donde los valores de retorno se comparen mediante otros tipos de igualdad.
2. El soporte para coinducción utilizando *sized types* parece más prometedor que el primero. Sería interesante intentar definir la mónada *delay* utilizando una representación con dicho soporte y analizar si con esa representación se puede demostrar la ley de intercambio y, por lo tanto, probar que el tipo *delay* constituye una mónada concurrente.

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.

[Dan18] Danielsson, Nils Anders: *Total Definitional Interpreters for Looping Programs.* https://www.cse.chalmers.se/~nad/publications/danielsson-definitional-interpreters-looping.html, 2018.

[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+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.

[Mog89] Moggi, Eugenio: *Computational lambda-calculus and monads.* En *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*, páginas 14–23, 1989.

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

[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.

[Vel17] Veltri, Niccolò: *A Type-Theoretical Study of Nontermination.* Tesis de Doctorado, Department of Software Science, School of Information Technologies, Tallinn University of Technology, May 2017.

[Wad92] Wadler, Philip: *The Essence of Functional Programming.* En *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, página 1–14, New York, NY, USA, 1992. Association for Computing Machinery.

Apéndice A

Pruebas complementarias

A.1. Pruebas análogas para bisemejanza fuerte

En la sección 6.2 se prueba que el tipo *delay* es una mónada tomando como relación de igualdad la bisemejanza débil. El siguiente bloque de código corresponde a la prueba análoga, tomando en este caso como relación de igualdad la bisemejanza fuerte.

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

  module _ {A : Set} {_∼_ : A → A → Set} (refl∼ : Reflexive _∼_) where

    open Equality _∼_ using (_≅_)
    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 (♭ 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 (♭ 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 6.3, 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) (♭ c)))
    associative (now a) (later b) (now c) = later (♯ (associative (now a) (♭ b) (now c)))
    associative (now a) (later b) (later c) = later (♯ (associative (now a) (♭ b) (♭ c)))
    associative (later a) (now b) (now c) = later (♯ (associative (♭ a) (now b) (now c)))
    associative (later a) (now b) (later c) = later (♯ (associative (♭ a) (now b) (♭ c)))
    associative (later a) (later b) (now c) = later (♯ (associative (♭ a) (♭ b) (now c)))
    associative (later a) (later b) (later c) = later (♯ (associative (♭ a) (♭ b) (♭ c)))


  module _ {A : Set} {_∼_ : (A × ⊤) → (A × ⊤) → Set}
           (reflA×⊤ : Reflexive _∼_) where

    open Equality {A × ⊤} _∼_ using (_≅_)
    open Equality.Rel

    rid : (a : A ⊥) → (merge a unit) ≅ (fmap (λ a → (a , tt)) a)
    rid (now x) = now reflA×⊤
    rid (later x) = later (♯ (rid (♭ x)))

  module _ {A : Set} {_∼_ : (⊤ × A) → (⊤ × A) → Set}
           (refl⊤×A : Reflexive _∼_) where

    open Equality {⊤ × A} _∼_ using (_≅_)
    open Equality.Rel
```

```
  lid : (a : A ⊥) → (merge unit a) ≅ (fmap (λ a → (tt , a)) a)
  lid (now x)   = now refl⊤×A
  lid (later x) = later (♯ (lid (♭ 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∼))
```

A.2. Lemas para la prueba de interchange para conaturales con notación musical

Los siguientes lemas se utilizan en la sección 6.5.2 para mostrar los intentos de prueba de la ley de intercambio para números conaturales definidos con notación musical.

≡⇒≳ prueba que la desigualdad es reflexiva respecto de la igualdad proposicional:

```
≡⇒≳ : {n₁ n₂ : Coℕ} → n₁ ≡ n₂ → n₁ ≳ n₂
≡⇒≳ {zero}    {zero}    n≡ = zero
≡⇒≳ {suc n₁} {suc .n₁} refl = refl≳ refl≈
```

Los lemas maxzero_1, maxzero_2, sumzero_1 y sumzero_2 prueban que zero es neutro a derecha de max y sum probando las dos desigualdades (ya que en los demás lemas se necesitaba de esta manera):

```
maxzero₁ : {n : Coℕ} → n ≳ max n zero
maxzero₁ {zero}  = zero
maxzero₁ {suc n} = refl≳ refl≈

maxzero₂ : {n : Coℕ} → max n zero ≳ n
```

```
maxzero₂ {zero}  = zero
maxzero₂ {suc n} = refl≳ refl≈

sumzero₁ : {m : Co𝕟} → m ≳ sum m zero
sumzero₁ {zero}  = zero
sumzero₁ {suc m} = refl≳ refl≈

sumzero₂ : {m : Co𝕟} → sum m zero ≳ m
sumzero₂ {zero}  = zero
sumzero₂ {suc x} = refl≳ refl≈
```

sym-sum y sym-max prueban que ambas operaciones son simétricas:

```
sym-sum : {m n : Co𝕟} → sum m n ≳ sum n m
sym-sum {zero}  {zero}  = zero
sym-sum {zero}  {suc n} = refl≳ refl≈
sym-sum {suc m} {zero}  = refl≳ refl≈
sym-sum {suc m} {suc n} = suc (♯ suc (♯ (sym-sum {♭ m} {♭ n})))

sym-max : {m n : Co𝕟} → max m n ≳ max n m
sym-max {zero}  {zero}  = zero
sym-max {zero}  {suc n} = refl≳ refl≈
sym-max {suc m} {zero}  = refl≳ refl≈
sym-max {suc m} {suc n} = suc (♯ sym-max {♭ m} {♭ n})
```

≳sum prueba que la suma es compatible con la relación de orden, para ello se prueba también ≳sumzero que prueba lo mismo para el caso en que el argumento n_2 es zero:

```
≳sumzero : {m₁ m₂ n : Co𝕟} → m₁ ≳ m₂ → n ≳ zero → sum m₁ n ≳ sum m₂ zero
≳sumzero zero    zero    = zero
≳sumzero zero    (suc^l q) = suc^l q
≳sumzero (suc x) zero    = suc (♯ (♭ x))
≳sumzero (suc x) (suc^l q) = suc (♯ (suc^l (trans≳ (≳sumzero (♭ x) q) sumzero₂)))
≳sumzero (suc^l p) zero    = suc^l (trans≳ p sumzero₁)
≳sumzero (suc^l p) (suc^l q) = suc^l (suc^l (≳sumzero p q))

≳sum : {m₁ m₂ n₁ n₂ : Co𝕟} → m₁ ≳ m₂ → n₁ ≳ n₂ → sum m₁ n₁ ≳ sum m₂ n₂
≳sum zero    ≳n      = ≳n
≳sum (suc x) zero    = suc x
≳sum (suc x) (suc x₁) = suc (♯ (suc (♯ (≳sum (♭ x) (♭ x₁)))))
≳sum (suc^l ≳m) (suc^l ≳n) = suc^l (suc^l (≳sum ≳m ≳n))
≳sum (suc^l ≳m) zero    = trans≳ (trans≳ ≳suc ≳m) sumzero₁
≳sum {n₂ = zero} (suc x) (suc^l ≳n) = suc (♯ (suc^l (trans≳ (≳sumzero (♭ x) ≳n) sumzero₂)))
≳sum {n₂ = suc n} (suc x) (suc^l ≳n) = suc (♯ suc (♯ ≳sum (♭ x) (trans≳ ≳n ≳suc)))
≳sum {suc m} {zero} {suc n} (suc^l ≳m) (suc x) = suc (♯ (suc^l (trans≳ (trans≳ (sym-sum {♭ m} {♭ n})
                                                  (≳sumzero (♭ x) ≳m)) sumzero₂)))
≳sum {m₂ = suc m} (suc^l ≳m) (suc x) = suc (♯ (suc (♯ (≳sum (trans≳ ≳m ≳suc) (♭ x)))))
```

Análogamente, ≳max prueba que max es compatible con la relación de orden, utilizando ≳maxzero para el caso en que n_2 = zero:

```
≳maxzero : {m₁ m₂ n : CoℕS} → m₁ ≳ m₂ → n ≳ zero → max m₁ n ≳ max m₂ zero
≳maxzero zero     zero     = zero
≳maxzero zero     (suc^l q) = suc^l q
≳maxzero (suc x)  zero     = suc x
≳maxzero (suc x)  (suc^l q) = suc (♯ trans≳ (≳maxzero (♭ x) q) maxzero₂)
≳maxzero (suc^l p) zero     = suc^l (trans≳ p maxzero₁)
≳maxzero (suc^l p) (suc^l q) = suc^l (≳maxzero p q)

≳max : {m₁ m₂ n₁ n₂ : CoℕS} → m₁ ≳ m₂ → n₁ ≳ n₂ → max m₁ n₁ ≳ max m₂ n₂
≳max zero     zero     = zero
≳max zero     (suc x)  = suc x
≳max zero     (suc^l q) = suc^l q
≳max (suc x)  zero     = suc x
≳max (suc x)  (suc x₁) = suc (♯ (≳max (♭ x) (♭ x₁)))
≳max (suc^l p) zero     = suc^l (trans≳ p maxzero₁)
≳max (suc^l p) (suc^l q) = suc^l (≳max p q)
≳max {n₂ = zero} (suc x) (suc^l q) = suc (♯ (trans≳ (≳maxzero (♭ x) q) maxzero₂))
≳max {n₂ = suc n} (suc x) (suc^l q) = suc (♯ (≳max (♭ x) (suc^r-1 q)))
≳max {suc m} {zero} {suc n} (suc^l p) (suc x) = suc (♯ trans≳ (trans≳ (sym-max {♭ m} {♭ n})
                                                    (≳maxzero (♭ x) p)) maxzero₂)
≳max {m₂ = suc m} (suc^l p) (suc x) = suc (♯ (≳max (suc^r-1 p) (♭ x)))
```

A.3. Lemas para probar que los conaturales con *sized types* forman un monoide concurrente

Los lemas que se presentan a continuación se utlizan en la sección 6.6.2 para probar que los números conaturales bajo la representación que utiliza *sized types* forman un monoide concurrente.

El primer lema representa la conmutatividad de la suma. Para probar tal propiedad se requiere un lema extra que postula que sumar 1 a la suma m + force n es igual a realizar la suma m + suc m, lo cual es similar a definir la suma al revés en el sentido de ir reduciendo el conúmero de la derecha en lugar del de la izquierda. Este lema se prueba junto con otro que permite cambiar el constructor suc de lado en la suma de manera mutuamente recursiva.

```
mutual

  suc+∼+suc : ∀ {m n i} → [ i ] suc m + force n ∼ force m + suc n
  suc+∼+suc {m} {n} = transitive-∼ (suc λ { .force → reflexive-∼ _ }) (1++∼+suc _)

  1++∼+suc : ∀ m {n i} → [ i ] ⌜ 1 ⌝ + m + force n ∼ m + suc n
  1++∼+suc zero = suc λ { .force → reflexive-∼ _ }
  1++∼+suc (suc _) = suc λ { .force → suc+∼+suc }

+-comm : ∀ m {n i} → [ i ] m + n ∼ n + m
+-comm zero {n} = symmetric-∼ (+-right-identity _)
+-comm (suc m) {n} = transitive-∼ (suc λ { .force → +-comm (force m) }) (1++∼+suc _)
```

El lema _+-cong_ representa la propiedad que indica que si se tienen dos pares de conúmeros bisemejantes $m_1 \sim m_2$ y $n_1 \sim n_2$, la suma de los dos primeros es bisemejante a la suma de los dos segundos.

```
infixl 6 _+-cong_

_+-cong_ : ∀ {i m₁ m₂ n₁ n₂} → [ i ] m₁ ∼ m₂ → [ i ] n₁ ∼ n₂
             → [ i ] m₁ + n₁ ∼ m₂ + n₂
zero +-cong q = q
suc p +-cong q = suc λ { .force → force p +-cong q }
```

El lema _max-cong_ representa la propiedad análoga a la anterior para el operador max.

```
infixl 6 _max-cong_

_max-cong_ :
  ∀ {i m₁ m₂ n₁ n₂} →
  [ i ] m₁ ∼ m₂ → [ i ] n₁ ∼ n₂ → [ i ] max m₁ n₁ ∼ max m₂ n₂
zero max-cong q = q
_max-cong_ {i} {.suc m} {.suc n} {zero} {zero} (suc p) zero =
                              suc λ { .force → transitive-∼ (max-right-identity (force m))
                                                (transitive-∼ (force p)
                                                (symmetric-∼ (max-right-identity (force n)))) }
suc p max-cong suc q = suc λ { .force → (force p) max-cong (force q) }
```

El siguiente lema, pred-max, demuestra que el máximo de los predecesores de dos conúmeros es igual al predecesor del máximo de los mismos.

```
pred-max : ∀ {i} m n → [ i ] max (pred m) (pred n) ∼ pred (max m n)
pred-max zero n = reflexive-∼ (pred n)
pred-max (suc m) n = reflexive-∼ (max (force m) (pred n))
```

Por último, el lema max≤+ prueba que el máximo de dos conúmeros siempre es menor o igual a su suma.

```
max≤+ : ∀ {i m n} → [ i ] max m n ≤ m + n
max≤+ {i} {zero} {n} = reflexive-≤ (max zero n)
max≤+ {i} {suc x} {n} = suc λ { .force → transitive-≤ (max≤+ {m = force x} {n = pred n})
                                          (reflexive-≤ (force x) +-mono pred≤) }
```

Apéndice B

Referencia al código fuente