Universidad Nacional de Rosario

Departamento de Ciencias de la Computación

Formalización de Containers

Tesina de grado

Autora:
Eugenia Simich

Director:
Dr. Mauro Jaskelioff

31 de octubre de 2016

Resumen

Muchos de los tipos de datos más utilizados comparten la característica de poder ser pensados como una serie de esqueletos, o plantillas, para contener otros datos. Las listas, los árboles, los streams, son ejemplos de tipos de datos de esta clase. Esta propiedad que los integra permite analizarlos de forma segregada: por una lado se encuentra su estructura y por otro la información a almacenar. Los *containers* se presentan como una buena alternativa de representación de esta clase de constructores de tipos, explotando esta posibilidad de separar estructura de contenido y proveyendo la posibilidad de estudiar la estructura de forma aislada.

Con el fin principal de evitar la reescritura de código, el paradigma conocido como *programación genérica* se dedica a la construcción de programas definidos de forma abstracta sobre un conjunto de tipos de datos. Es decir, en lugar de escribir algoritmos que trabajen sobre una estructura en particular, se definen sobre un conjunto amplio de tipos de datos. La delimitación y posterior inspección de este conjunto es una problemática clave a resolver a la hora de programar genéricamente. Existe una gran variedad de lenguajes de programación donde las herramientas para cumplir este objetivo son provistas de antemano. Cuando este no es el caso, o cuando trabajamos teóricamente, la construcción de universos es un recurso útil y válido.

Desde este punto de vista, los containers se presentan como un universo particular donde es posible razonar de forma abstracta y programar genéricamente con el mencionado conjunto de constructores de tipos.

Esta tesina presenta una implementación del universo definido por los containers y una serie de propiedades interesantes acerca de ellos en el lenguaje de programación *Agda*. Se dará a su vez pruebas formales del cumplimiento de estas propiedades, extendiendo así las bibliotecas de containers existentes a la fecha. Este trabajo explota y reúne principalmente dos propiedades interesantes de los lenguajes con tipos dependientes como Agda. Por un lado, la posibilidad de construir de forma muy precisa universos que limiten la expresividad de los programas de una forma útil, dejando afuera comportamientos no deseados y posibilitando la programación genérica. Por otro lado, hace uso de la posibilidad de utilizar el mismo lenguaje para construir pruebas de propiedades sobre los programas que escribimos. En un lenguaje de tipos dependientes, implementar y demostrar resultan ser la misma tarea.

Índice general

Introducción 1

I Preliminares 7

1. Introducción a Agda 9

1.1. Tipos de datos . . . 10
1.2. Pattern matching y funciones anónimas . . . 11
1.3. Polimorfismo y argumentos implícitos . . . 12
1.4. Otro tipo de datos y operadores infijos . . . 13
1.5. Records . . . 14
1.6. Inducción, recursión y terminación . . . 14
1.7. Tipo de datos dependientes . . . 15
1.8. Proposiciones como tipos y la equivalencia proposicional . . . 16
1.9. Suma dependiente . . . 18
1.10. Equivalencia heterogénea . . . 19
1.11. Extensionalidad . . . 21
1.12. Universos . . . 22

2. Teoría de Categorías 25

2.1. Categoría . . . 26
2.2. Isomorfismo . . . 28
2.3. Funtor . . . 29
2.4. Transformación natural . . . 32
2.5. Último ejemplo y motivación de Containers . . . 33

II Formalización de Containers 35

3. Containers 37

3.1. Motivación . . . 38
3.2. Definición y extensión . . . 39
3.3. Morfismos de containers . . . 47
3.4. Categoría de containers . . . 50

4. Construcciones con containers **51**

4.1. Coproducto 52

4.2. Producto 58

4.3. Inicial y terminal 64

4.4. Exponencial 68

Conclusiones **79**

A. Hoja de ruta del código fuente **81**

Bibliografía **83**

Referencias históricas 84

Bibliografía general 86

Introducción

> El otro proyecto era un plan para abolir por completo todas las palabras, cualesquiera que fuesen; y se defendía como una gran ventaja, tanto respecto de la salud como de la brevedad. Es evidente que cada palabra que hablamos supone, en cierto grado, una disminución de nuestros pulmones por corrosión, y, por lo tanto, contribuye a acortarnos la vida; en consecuencia, se ideó que, siendo las palabras simplemente los nombres de las cosas, sería más conveniente que cada persona llevase consigo todas aquellas cosas de que fuese necesario hablar en el asunto especial sobre que había de discurrir.
> (...) muchos de los más sabios y eruditos se adhirieron al nuevo método de expresarse por medio de cosas: lo que presenta como único inconveniente el de que cuando un hombre se ocupa en grandes y diversos asuntos se ve obligado, en proporción, a llevar a espaldas un gran talego de cosas.
>
> Los Viajes de Gulliver
> Jonathan Swift

Sin dudas, en un mundo como el que plantea el exergo, las ciencias de la computación no tendrían lugar. La historia de esta disciplina está conformada, en gran parte, de sucesivos intentos de formalizar el razonamiento. Como resultado se obtuvo la posibilidad de crear cosas con palabras, y la posibilidad de elegir qué cosas creamos a partir de enriquecer o limitar el lenguaje utilizado.

En ese sentido, este trabajo no es la excepción, puesto que a grandes rasgos trabajaremos con el objetivo de formalizar una serie de construcciones que, por su parte, funcionan como un lenguaje para expresar de forma alternativa ciertos constructores de tipos de datos.

Los *containers* [Abb03, AAG03] se presentan como una opción de representación de un conjunto de tipos de datos paramétricos, aquellos que pueden ser pensados como estructuras, esqueletos para contener otros datos. Segregando estructura de contenido, esta forma de representación posibilita el análisis aislado de la estructura.

Se conoce como paradigma de programación genérica [BGHJ07] a aquella disciplina que problematiza, entre otras cuestiones, la construcción de algoritmos cuyos argumentos puedan pertenecer no sólo a un tipo de datos particular, sino a un conjunto amplio. Entre sus objetivos se encuentran el ahorrar código y favorecer la claridad, expandiendo las posibilidades de parametrización. En este sentido, al identificar a un extenso grupo de tipos de datos, los containers promueven el análisis abstracto y se postulan como una buena alternativa hacia la construcción de programas genéricos [AMM07].

Para comenzar, se realizará un repaso amplio del contexto histórico de esta disciplina, haciendo énfasis particular en algunos puntos con mayor relevancia para este trabajo.

Un siglo de historia en 1400 palabras (incluidas estas tres)

Hacia finales de siglo XIX y principios del XX, la matemática estaba pasando por una etapa predominantemente formalista. Muchos estudiosos de la época veían la necesidad de refundar toda la matemática sobre una base axiomática en la que se pudieran probar formalmente, dentro del sistema, todos los teoremas posibles. Más aún, estaban los que opinaban que la matemática era reducible a la lógica, siendo toda verdad matemática una verdad lógica. Uno de los primeros intentos de llevar a la práctica estas ideas, si bien de gran influencia posterior, será célebre por un error. Gottlob Fregue publica *Begriffsschrift* [Fre79] en 1879, donde presenta un sistema formal para la lógica de predicados. Este estaba pensado no sólo como modelo de la aritmética, sino que Fregue va más allá y lo propone como modelo del pensamiento y posible instrumento para filósofos.

Veinte años más tarde, Bertrand Russell encuentra un problema en el sistema de Fregue y le escribe una carta. Se trataba de lo que hoy se conoce como la paradoja de Russell, que se resume en la posibilidad de definir un conjunto, llamémosle R, tal que

$$s \in R \Leftrightarrow s \notin s$$

En particular, substituyendo R por s, obtenemos la siguiente inconsistencia

$$R \in R \Leftrightarrow R \notin R$$

La impredicatividad, es decir, la posibilidad de definir predicados cuantificados sobre un conjunto al que el predicado pertenece, es una característica que puede traer paradojas de esta índole.

A pesar del tropiezo, el impulso formalista continúa. Russell junto con Whitehead se proponen seguir el legado de Fregue y publican entre 1910 y 1913 un tratado de tres volúmenes llamado *Principia Mathematica* [WR13]. Allí postulan un sistema tan preciso como difícil de ser llevado a cabo, a tal punto que la formulación de $1 + 1 = 2$ aparece recién en el segundo volumen y su prueba, en el tercero. Sin embargo, fue muy bien aceptado por la comunidad científica, instaurando aún más la idea de que la formalización lógica de la matemática era posible. El objetivo era ahora lograr derivar toda la matemática y probar toda proposición verdadera dentro de la formalización de los *Principia*.

En esta escena es donde aparece Kurt Gödel a traer malas noticias. En 1931, con su primer teorema de incompletitud [God31], demuestra que en cualquier sistema consistente, tan poderoso como el de los *Principia*, donde se pudiera modelar aritmética básica, existen juicios trivialmente verdaderos que resultan ser imposibles de demostrar. Es decir, demuestra que no se puede tener completitud y consistencia a la vez.

Segundo traspié y los deseos formalistas empiezan a ser un poco más conservadores. ¿Podremos tener al menos un mecanismo de decisión que diga si un juicio es o no demostrable? Es decir, ¿existe un *algoritmo* que decida si una fórmula es o no un teorema? A esta pregunta se la conoce como el *problema de la decisión*, traducción del vocablo alemán *Entscheidungsproblem* [HA28]. La respuesta fue otra vez por la negativa. De forma independiente, Alonzo Church en 1936 [Chu36] y muy pocos meses después Alan Turing [Tur36], proponen modelos de computación –el lambda cálculo y las máquinas de Turing, respectivamente– y demuestran la imposibilidad de construir dicho algoritmo en sus sistemas. Turing, por su parte, también prueba la equivalencia de su sistema con el previamente propuesto por Church y lo publica de todas formas, ya que resultan ser lo

suficientemente diferentes en cuanto a lo que filosofía se refiere [Wad15]. La formulación de Turing preveía la posibilidad de la existencia de máquinas de computar, mientras que la formulación de Church fue originalmente pensada como una nueva notación para la lógica.

Hasta ahora podemos concluir que cada tropiezo trajo consigo sus frutos. En este caso, de la respuesta al Entscheidungsproblem nacen la formulación de los primeros lenguajes de computación y la noción de lo *efectivamente calculable*. Una función es efectivamente calculable si puede ser formulada como un término del lambda cálculo, o equivalentemente, como un procedimiento en la máquina de Turing.

El desarrollo de la lógica, la matemática y este nuevo estudio de lo computable, continuó. Dentro del ámbito de la primera, Gerhard Gentzen introduce la deducción natural [Gen35], entre muchas otras formulaciones y estilos de notación que perduran hoy en día. Por el lado de las ciencias de la computación, Kleene y Rosser, alumnos de Church, encuentran un problema en el lambda cálculo [KR35]. Así como la impredicatividad del sistema propuesto por Fregue traía aparejada la inconsistencia del sistema, en el caso del lambda cálculo, la posibilidad de aplicar funciones a sí mismas significaba tener algoritmos que no terminasen. Esto podía ser o no un problema, pero con la intención de poder asegurar la terminación de las funciones, Church hace uso de una solución equivalente a la de Russell y agrega tipos al lambda cálculo [Chu40], asegurando su terminación. Por el lado de la matemática, hacia la década del 40, Samuel Eilenberg y Saunders Mac Lane introducen nociones fundantes de la nueva teoría de categorías [EML45], como ser la noción de categoría, funtor y transformación natural como parte de su trabajo en álgebra topológica.

En 1934, Haskell Curry observa un hecho curioso relativo al tipo funcional $A \to B$ y la implicación $A \supset B$; cada función de dicho tipo podía ser entendida como la prueba de la implicancia. Es así como Curry, junto con Robert Feys publican esta idea en sus trabajos sobre lógica combinatoria [CF58], donde se hace explícita la correspondencia entre tipos y proposiciones, pruebas y funciones. Tiempo más tarde, finalizando la década del 60, motivado por estas observaciones, William Howard [How80], alumno de Mac Lane, encuentra una correspondencia entre pruebas en deducción natural y programas en lambda cálculo simplemente tipado, agregando que la normalización de pruebas en deducción natural se corresponde con la beta reducción del cálculo lambda. Además, incluye otras correspondencias estructurales como ser la del producto y la conjunción lógica o la unión disjunta y la disyunción. Esta correspondencia se conoce hoy en día como el isomorfismo de Curry-Howard. Esta correlación ya había sido vislumbrada por los lógicos intuicionistas Brower, Heyting e independientemente por Kolmogorov, aunque no ligándola de forma explícita con el lambda cálculo.

Sería también un lógico intuicionista, el sueco Per Martin Löf, el que encontraría en estas ideas la inspiración para la formulación de la teoría de tipos intuicionista, base de muchos lenguajes de programación/asistentes de prueba como lo es Agda, lenguaje que utilizaremos en este trabajo; o también *Epigram* [McB04], *Idris* [Bra13, Bra] y *Coq* [Coq04]. Hacia 1971 el ámbito científico ya conoce un primer manuscrito de Per Martin-Löf donde describe su teoría de tipos intuicionista, fuertemente influenciado por los trabajos de Curry y Feys. En ese manuscrito, Martin Löf expone una teoría de tipos dependientes con la intención de utilizar la lógica como fundamento de las matemáticas, continuando con las intenciones formalistas de principios de siglo.

Una de las modificaciones que se realizarían a ese primer manuscrito nunca publicado

es la introducción de universos de tipos. La teoría de tipos en esa primera versión incluía un universo V al que pertenecían todos los tipos de datos, incluyendo el axioma $V \in V$. Es decir, el axioma asevera que el tipo de datos que aglutina a todos los tipos de datos, también es un tipo de datos. En su tesis doctoral, Jean-Yves Girard demuestra que este axioma introduce una inconsistencia [Gir72], producto de una paradoja similar a la paradoja de Russell. Para evitar este problema, se introduce a la teoría una jerarquía de universos [ML98], tomando de la teoría de categorías la idea de categorías pequeñas y categorías grandes. El universo V será entonces, el universo al que pertenecen los *tipos pequeños*. El elemento V en sí y todos los tipos construidos a partir de él, pertenecerán al siguiente nivel, V_1. Queda así establecida una jerarquía de universos:

$$V = V_0 \in V_1 \in \ldots \in V_n \in \ldots$$

Si bien la introducción de universos en la teoría de tipos fue necesaria para mantener la consistencia del sistema, es un recurso muy útil si buscamos parametricidad en funciones y tipos o programar genéricamente.

También a comienzos de la década del 70 aparecería un resultado que uniría la lógica y las ciencias de la computación con la teoría de categorías de una forma muy interesante. Joachim Lambek [Lam72] muestra que el isomorfismo de Curry-Howard tiene una tercera pata: las categorías cartesianas cerradas muestran una correspondencia ecuacional con la lógica y el lambda cálculo tipado, siendo posible interpretar a los tipos (o proposiciones) como objetos de la categoría y a los términos (o pruebas) como los morfismos. Es por dicha razón que en la actualidad muchos se refieren a la correspondencia expuesta como de *Curry–Howard–Lambek*.

Estado del arte

Los containers son concebidos por Abbott, Altenkirch y Ghani en 2003, en un primer trabajo denominado *Categories of Containers* [AAG03] y profundizado luego en la tesis doctoral del primero [Abb03]. Allí, exponen una formalización matemática de los containers, entendiéndolos como una forma de representar aquellos tipos de datos que consisten en *plantillas* a llenar con otros datos. Muestran también que los containers resultan ser cerrados bajo una serie de construcciones; en particular, prueban que es posible representar a todos los tipos estrictamente positivos. Asimismo, introducen la extensión de los containers como una forma de reinterpretarlos como constructores funtoriales de tipos de datos. Siguiendo esta línea, presentan lo que llamaremos morfismos de containers como el conjunto de funciones polimórficas entre estos tipos de datos, que categóricamente hablando resultan ser las transformaciones naturales entre los funtores asociados.

En el trabajo *Higher Order Containers*, Altenkirch, Levy y Stanton [ALS10] exponen, entre otros resultados, la existencia de exponenciales para todo par de containers, convirtiendo a la categoría en cartesiana cerrada. A pesar de que la categoría de containers fue en un principio pensada como modelo para tipos de datos, este resultado implica que también se puedan interpretar construcciones de alto orden.

En los trabajos recientemente mencionados se prueba adicionalmente que los containers –ampliando la formulación para representar constructores con múltiples argumentos– son cerrados bajo álgebra inicial y coálgebra final, convirtiéndolos en una buena alternativa de semántica composicional. En particular, resultan ser un buen modelo semántico para los denominados tipos estrictamente positivos.

Luego de introducir el universo de containers y realizar aportes relacionados, como ser la exposición de sus exponenciales, Altenkirch *et al* presentan los containers indexados [AGH+15], una generalización que pasa de representar tipos de datos a familias de tipos, haciendo énfasis particular en la utilización de los containers indexados como modelo de las familias estrictamente positivas. Como punto de vista alternativa a un mismo objeto de estudio se pueden citar los trabajos acerca de funtores polinomiales [GK10] realizados por Kock y Gambino.

También en lo que refiere a semántica de tipos de datos, los *shapely types* –i.e. tipos con forma– son introducidos por Jay y Cockett [JC94, Jay95]. Estos muestran una conexión interesante puesto que su construcción se basa en una segregación de estructura y contenido, igual que el universo que se analiza en el presente trabajo. Puede demostrarse que todo shapely type es factible de ser representado como container.

Remontándonos más en el tiempo se puede encontrar otro trabajo relacionado que vale mencionarse: es el que refiere a las denominadas especies combinatorias, originalmente propuestas por Joyal en los años 80 [Joy81]. Para ese momento, la teoría sobre tipos datos algebraicos en lenguajes funcionales estaba siendo desarrollada de forma independiente; sin embargo, la relación es evidente. En el citado trabajo de 1981 se exponen a las especies combinatorias como una categoría particular de endofuntores en **Set**.

Contribuciones

En esta tesina se analizan algunas nociones presentadas en los trabajos citados, dejando otras de lado. Particularmente, se trabajará con el universo de containers y sus morfismos, con sus respectivas funciones de extensión y con las construcciones de producto, coproducto, exponencial, objeto inicial y objeto terminal.

Como contribución original, en el presente trabajo se presenta una formalización en Agda, en términos categóricos, del universo de containers y sus construcciones. Se proveen pruebas formales de que los containers en efecto conforman una categoría y se demuestra que esta cuenta con coproductos, productos, objetos inicial y terminal y exponenciales. En consecuencia, se extienden las librerías de containers existentes a la fecha a la vez que se proveen las garantías de estar construyendo los elementos correctos.

Por otro lado, se provee otra contribución en cuanto a lo que presentación de la temática se refiere. Debido a que este trabajo posee el formato de una tesina de grado y a que no se asumen avanzados conocimientos, la exposición detallada y muchas veces intuitiva resulta un aporte a la comprensión y la difusión del tema expuesto hacia un público más amplio; no sólo en lo relativo al caso particular de los containers, sino también a modo de ejemplo de formalización y construcción de un universo para la programación genérica.

Panorama

El presente trabajo se encuentra organizado en dos partes. La primera parte estará avocada a presentar sucintamente nociones básicas de dos lenguajes que utilizaremos a lo largo de toda la tesina. Se trata de, por un lado, el lenguaje de programación *Agda*, presentado en el capítulo 1 y por otro lado, la teoría de categorías, en el capítulo 2.

Para el primero seguiremos un camino muchas veces transitado por diversos tutoriales e introducciones a la programación en este lenguaje, por lo que puede obviarse si se poseen conocimientos previos. En el caso del segundo capítulo, por lo contrario, ya se asumirá un manejo del lenguaje Agda y se secundarán todas las definiciones categóricas con formalizaciones dadas en dicho lenguaje. Dichas implementaciones se retomarán en la segunda parte, razón por la cual puede resultar necesaria su lectura para la correcta comprensión de la parte subsiguiente.

La segunda parte nos sumerge ya en el mundo de los containers. En el capítulo 3 presentaremos formalmente este universo y daremos cuenta de su potencialidad, a partir de analizar de forma intuitiva cada construcción y de proveer múltiples ejemplos. En el capítulo 4 retomamos la exposición del lenguaje categórico con el objetivo de presentar propiedades sobre containers en estos términos. En contrapunto, veremos la formalización de cada propiedad en Agda y la exposición y prueba de su cumplimiento para el caso particular de la categoría de containers.

Cada capítulo incluye porciones de código Agda en contexto y con la explicación correspondiente. Sin embargo, el código completo puede obtenerse en `http://www.github.com/eugeniasimich/containers`[1].

[1] ver apéndice A, pág. 81

Parte I

Preliminares

Capítulo 1

Introducción a Agda

> Mas para conversaciones cortas, un hombre puede llevar los necesarios utensilios en los bolsillos o debajo del brazo, y en su casa no puede faltarle lo que precise. Así, en la estancia donde se reúnen quienes practican este arte hay siempre a mano todas las cosas indispensables para alimentar este género artificial de conversaciones.
>
> Los Viajes de Gulliver
> Jonathan Swift

Agda [Nor07] es un lenguaje de programación desarrollado por la Universidad de Chalmers, que se caracteriza principalmente por poseer lo que se denominan tipos dependientes. A diferencia de algunos lenguajes de programación, como ser Haskell o ML, donde se diferencia claramente el lenguaje de los términos y el lenguaje de los tipos de datos, en los lenguajes con tipos dependientes los tipos son también valores. Esto implica que pueden depender de otros valores, ser tomados como argumento o retornados por funciones.

Gracias al isomorfismo de Curry-Howard, Agda es no sólo un lenguaje de programación sino también un asistente de prueba. Coloquialmente hablando, el isomorfismo de Curry-Howard es una correspondencia entre la teoría de tipos y la lógica, equiparando tipos con proposiciones, términos con pruebas y evaluación de términos a forma normal, con normalización de pruebas. Como los tipos dependientes permiten hablar sobre valores dentro del mismo tipo, podemos expresar proposiciones que se refieran a los valores, como tipos de datos. Dichas proposiciones serán teoremas una vez que se provean habitantes del tipo de datos. Es decir que los términos son pruebas formales de la veracidad de las proposiciones.

Agda es una implementación de la teoría de tipos de Martin-Löf [ML98, ML75, ML82, ML84]. Este introduce una lógica intuicionista, que a diferencia del lenguaje tradicional utilizado previamente en sistemas intuicionistas, permite incluir pruebas como parte de las proposiciones, utilizando el mismo lenguaje. Consciente de la relación, Martin-Löf propone el uso de su lógica intuicionista como un lenguaje de programación. Para que la lógica sea consistente, los programas deberán ser totales, no pudiendo fallar o no terminar, es por esta razón que Agda incluye mecanismos para la comprobación de la terminación de los programas.

Esta sección está dedicada a presentar una introducción a Agda, prestando especial atención a cuestiones necesarias para la exposición de la temática principal de esta tesina.

1.1. Tipos de datos

Para comenzar esta breve introducción al lenguaje de programación Agda, veremos cómo definir tipos de datos. La principal forma de hacerlo se muestra en el ejemplo del código 1.1.

Comenzamos definiendo nombre, parámetros y signatura del nuevo tipo de datos, entre las palabras reservadas data y where:

data *<nombre>* *<parámetros>* : *<signatura>* where.

En el siguiente ejemplo definimos al tipo Bool, que carece de parámetros y cuya signatura es Set; es decir, es un conjunto.

Código 1.1. *Booleanos*

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

En Agda, el lenguaje de términos y el lenguaje de tipos es el mismo. El elemento Set es el tipo que tienen los tipos de datos. Dicho esto, una pregunta sensata resulta ser: ¿eso significa que Set tiene tipo Set? La respuesta a esta pregunta es no y la razón es evitar la paradoja de Girard, mencionada en la sección de introducción. Para explicarlo someramente podemos decir que $\text{Set} : \text{Set}_1$ y proceder a corregir una aseveración previa: Set es el tipo que tienen los tipos de datos *pequeños*. El tipo Set_1, por su parte, tendrá tipo Set_2. Se genera así la siguiente jerarquía de tipos:

$$\text{Set} : \text{Set}_1 : \text{Set}_2 : \text{Set}_3 \ldots$$

En esta tesina tomaremos el criterio general de pasar por alto el detalle de a qué nivel de universo pertenecen nuestras construcciones, con el objetivo de simplificar las exposiciones. En su lugar, a menos que se indique explícitamente de otra forma, consideraremos **Set** : **Set**. Sin embargo, en el código Agda que acompaña a esta tesina, los distintos niveles se respetan.

Retomando el análisis del código expuesto en 1.1, observemos que la definición continúa con la declaración de los constructores de términos. La indentación no es una mera cuestión de estilo sino que forma parte del lenguaje. En este caso decimos que los únicos habitantes del tipo Bool son los elementos true y false. Es decir, implícitamente esta forma de definir datos nos asegura que los constructores que declaremos sean todas las formas de construcción posibles. De esta forma, si quisiéramos definir un tipo vacío simplemente no declararíamos constructor alguno, como se puede observar en el código 1.2 donde definimos el tipo vacío $\bot$.

Código 1.2. *Tipo vacío* $\bot$

```
data ⊥ : Set where
```

Podría resultarnos curioso a primera vista la utilización de símbolos como ser "$\bot$" o subíndices "$_1$". En efecto, otra característica particular de Agda es que permite la utilización de caracteres unicode, haciendo más natural la introducción de conceptos matemáticos.

1.2. Pattern matching y funciones anónimas

Ahora que contamos con nuestros dos primeros tipos de datos, pasemos a construir funciones sobre ellos. Una función muy simple que podemos construir sobre el tipo Bool es la función identidad.

Código 1.3. *Función identidad para booleanos*

```
idBool : Bool → Bool
idBool x = x
```

Notemos la función identidad no opera sobre el valor que recibe, simplemente lo retorna. Si quisiéramos operar sobre valores booleanos, tendríamos que diferenciar los casos donde x es true de los casos donde es false. La función not de negación toma un valor booleano y retorna otro, siendo este falso cuando el argumento es verdadero y viceversa. Gracias a que la construcción de un tipo de datos se realiza mediante la declaración de un conjunto determinado de constructores, resulta muy sencillo garantizar la totalidad de las funciones, asegurando que la función se defina para cada constructor. Este recurso tiene el nombre de coincidencia de patrones, más conocido por su versión en inglés *pattern matching* y es la forma que tiene Agda para asegurar la totalidad. En este caso, la función not puede aceptar sólo valores de la forma true o false, dedicándosele una línea de código a cada posibilidad.

Código 1.4. *Función de negación*

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

A diferencia de lenguajes funcionales como Haskell o ML, Agda no siempre puede inferir el tipo de las funciones, por lo que la declaración de signatura, i.e., la primera línea de la definición de not del código 1.4, no puede obviarse.

Otro tipo de pattern matching que permite la sintaxis de Agda es sobre funciones anónimas. Una función anónima es una función sin nombre. Por ejemplo, podemos definir la función identidad de forma anónima como $(\lambda\ x \to x)$. Es decir, dado un valor cualquiera, lo retornamos.

Pero como hemos observado antes, la función idBool en particular no observa nada del argumento. ¿Cómo haríamos si quisiéramos retornar algo en función del valor del argumento? Agda permite hacer pattern matching dentro de funciones anónimas utilizando llaves para englobar la función, y punto y coma para separar los casos. La versión anónima de not es entonces: (λ {true → false; false → true}).

Otro ejemplo de uso de pattern matching en funciones anónimas se muestra en el código 1.5, donde definimos la función and de conjunción de booleanos. Observar que en el caso donde el primer argumento es falso, no importa el valor del segundo y lo marcamos con guión bajo. El guión bajo ubicado en la posición de una variable indica que dicha variable no se utiliza y por lo tanto no es necesario darle un nombre.

Código 1.5. *Conjunción de booleanos*

```
and : Bool → Bool → Bool
and = λ { true  y → y
        ; false _ → false }
```

Podríamos necesitar también hacer pattern matching sobre más de un argumento. Lógicamente esto es igualmente posible, por ejemplo, para definir la función xor de booleanos:

```
xor : Bool → Bool → Bool
xor = λ { true  true  → false
        ; false false → false
        ; _     _     → true}
```

Hay un caso especial de pattern matching al que le queremos prestar especial atención: el patrón absurdo. Digamos que queremos definir una función que vaya del tipo vacío hacia los booleanos. ¿Es posible construir tal función, considerando que no tenemos elementos en el dominio? Justamente, resulta trivial definirla, es la función vacía. En Agda se indica al patrón absurdo con paréntesis que abren y cierran, como se observa en el código 1.6.

Código 1.6. *Función vacía hacia los booleanos*

```
emptyToBool : ⊥ → Bool
emptyToBool ()
```

Hemos visto entonces algunas formas de escribir funciones en Agda haciendo uso del pattern matching. La variable x en la definición de la función identidad, en el código 1.3, es un patrón que asocia cualquier valor posible de booleano al nombre x; el guión bajo es un patrón que también asocia todo valor del tipo dado, pero sin asignarle un nombre; los constructores true y false se asocian con las dos posibles formas que puede tener un habitante del tipo de los booleanos mientras que el patrón absurdo indica la ausencia de valor posible para dicho argumento.

1.3. Polimorfismo y argumentos implícitos

Observemos que la función idBool definida en el código 1.3 podría haber sido definida para cualquier tipo de datos en lugar de los booleanos. Esto es posible porque simplemente retorna inmaculado el valor que se le pasó como argumento y no necesita conocer detalles del tipo al que pertenece. Quisiéramos modificar dicha función para expresar este hecho, es decir, quisiéramos poder decir que la función identidad toma un elemento de un tipo cualquiera y retorna otro del mismo tipo. Un programador de Haskell seguramente propondría escribir el siguiente código, siendo X una variable que representa cualquier tipo de datos.

```
id₀ : X → X
id₀ x = x
```

Ante dicho código, Agda dirá que hay un error, alegando que la variable X no está definida. Para remediar esta cuestión, simplemente hay que informar que X es un tipo de datos, agregándolo como parámetro de la función, como se muestra en el código 1.7.

Código 1.7. *Función identidad polimórfica*

```
id₁ : (X : Set) → X → X
id₁ _ x = x
```

Contamos también con la posibilidad de definir funciones con *argumento implícito*. Un argumento que se define entre llaves en la signatura se denomina argumento implícito

y puede obviarse su escritura siempre que Agda sea capaz de inferirlo.

Código 1.8. *Función identidad polimórfica con argumento implícito*

```
id : {X : Set} → X → X
id x = x
```

Notemos que si bien incluimos a X como parámetro en la signatura de la función id, aclarando también su pertenencia a Set, no tenemos que incluirlo en la definición. Siempre que sea inferible, tampoco tendremos que hacerlo al momento de usar la función identidad.

Código 1.9. *Función vacía polimórfica*

```
empty : {X : Set} → ⊥ → X
empty ()
```

En el código 1.9 recién expuesto vemos otro ejemplo de función que puede ser definida para cualquier tipo de datos en lugar de hacerlo exclusivamente para los booleanos; se trata de la versión polimórfica de la función emptyToBool presentada en el código 1.6. Notar que el argumento X se define de forma implícita y por lo tanto no se incluye en la definición pues no resulta necesario.

Para tener en cuenta en las futuras secciones, la versión anónima de la función vacía es sencillamente λ () o bien λ {} si el argumento es implícito.

1.4. Otro tipo de datos y operadores infijos

Vemos a continuación un ejemplo donde el tipo de datos a construir tiene como parámetros a otros tipos de datos.

Código 1.10. *Producto cartesiano de conjuntos*

```
data _×_ (X Y : Set) : Set where
  _,_ : X → Y → X × Y
```

El tipo de los pares, también conocido como producto cartesiano de conjuntos, es un tipo construido a partir de otros dos. En estos casos decimos que el tipo _×_ se encuentra *parametrizado* por los tipos X e Y. El elemento _,_ es una función que construye un valor del producto cartesiano de X e Y a partir de un elemento de cada uno de ellos.

Utilizar guiones bajos en una definición nos permite luego ubicar los argumentos en los lugares donde se encontraban los guiones. En particular, obtenemos una notación infija, como podemos observar en la última línea de la definición de tipo, donde aparece $X \times Y$. De la misma forma podemos utilizar el constructor de datos _,_. Por ejemplo, observar la siguiente función que proyecta el primer elemento de un par, definida en el código 1.11. Este recurso puede asimismo utilizarse para definir funciones posfijas o multifijas.

Código 1.11. *Proyección izquierda de un par*

```
proj₁ : ∀{X Y} → X × Y → X
proj₁ (x , y) = x
```

Siempre que Agda pueda inferir el tipo de un elemento, será posible utilizar la notación $\forall\{a\}$ en lugar de $\{a : A\}$. Es decir, sin indicar explícitamente el tipo. Esto hacemos en la signatura de proj_1 ya que en este caso, Agda puede inferir a partir de la definición del tipo producto dada en 1.10 que tanto X como Y deberán ser habitantes de Set.

1.5. Records

Una forma interesante de definir tuplas de datos como lo es el producto, es mediante un record. De la misma forma que ciertas construcciones de muchos lenguajes de programación como ser las estructuras de C o los homónimos records de Haskell, los records en Agda son valores compuestos por otros de distinto tipo: sus campos, listados por nombre debajo de la palabra reservada field. En este caso, nombramos proj_1 y proj_2 al valor izquierdo y derecho del par, respectivamente. Opcionalmente podemos definir un símbolo para el constructor mediante el uso de la cláusula constructor. En este caso decidimos usar un símbolo de coma.

Código 1.12. *Redefiniendo el producto cartesiano de conjuntos utilizando records*

```
record _×_ (X Y : Set) : Set where
  constructor _,_
  field
    proj₁ : X
    proj₂ : Y
```

Los nombres de los campos del record hacen las veces de funciones de acceso al mismo. Toman como argumento una instancia del record y retornan su primer y segundo elemento, respectivamente:

```
proj₁ : ∀{X Y} → X × Y → X
proj₂ : ∀{X Y} → X × Y → Y
```

1.6. Inducción, recursión y terminación

Otro tipo de datos que nos será de utilidad es el de los números naturales, es decir, los números enteros no negativos. Lo construimos de forma inductiva, basándonos en la formalización de Peano. Declaramos primero un constructor zero que represente al número natural cero y luego una función suc que asegure que para todo número natural existe su sucesor.

Código 1.13. *Conjunto de números naturales*

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

La suma sobre números naturales puede ser definida haciendo recursión primitiva sobre el primer argumento.

Código 1.14. *Suma de naturales*

```
suma : ℕ → ℕ → ℕ
suma  zero      y = y
suma  (suc x)   y = suc (suma x y)
```

Tenemos dos casos: el caso base, donde decimos que la suma entre cero y cualquier otro número resulta en este último; y por otro lado el caso recursivo, donde decimos que sumar el sucesor de un número con cualquier otro es igual al sucesor de la suma de ellos dos.

Como los programas en Agda son totales, se deberá garantizar su terminación. Seguramente el lector se encontrará sorprendido ante tal aseveración. ¿Cómo es esto posible, si estamos frente al famoso problema de la parada, que sabemos indecidible? Pues bien, Agda es un lenguaje para el cual el chequeo de terminación resulta decidible, pero lo logra perdiendo expresividad en el camino. Usando Agda, obtenemos garantía de terminación en detrimento de turing-completitud. Las llamadas recursivas a las funciones deben realizarse sobre argumentos estructuralmente más pequeños para así poder garantizar que terminan. En este sentido, Agda considera que x es menor en estructura que suc x y acepta a suma como válida.

1.7. Tipo de datos dependientes

La primera característica del lenguaje de programación Agda que suele mencionarse a la hora de describirlo es la de poseer *tipos dependientes*. Un tipo dependiente es aquél que depende de valores de otro tipo. Más precisamente, que se encuentra indexado por otro tipo. Un ejemplo paradigmático de tipo de datos dependiente es el de los vectores, listas de tamaño fijo.

Código 1.15. *Vectores*

```
data Vec (A : Set) : ℕ → Set where
  nil   : Vec A zero
  cons  : {n : ℕ} → A → Vec A n → Vec A (suc n)
```

Se puede observar en la definición que Vec A : ℕ → Set, es decir, queda definida una familia de tipos de datos indexada por los naturales. Para cada número natural, obtenemos el tipo de los vectores de dicha longitud.

Una forma sencilla de tipo dependiente es la de las funciones donde el tipo resultado depende de los valores del resto de los argumentos. Por ejemplo, el tipo $(x : A) \to B$ construye aquellas funciones que toman un valor x de tipo A y retornan un valor de tipo B, donde se permite que B dependa de x. Particularmente, x podría ser a su vez un tipo, siendo A el conjunto Set. Este caso lo hemos visto en reiteradas ocasiones hasta ahora; en la definición de la función identidad, la función vacía polimórfica, las proyecciones del producto.

1.8. Proposiciones como tipos, programas como pruebas y la equivalencia proposicional

Hemos mencionado en la introducción que el sistema de tipos de Agda es lo suficientemente poderoso como para poder representar proposiciones en la forma de tipos cuyos habitantes serán las pruebas de dichas proposiciones. Vimos además, quizás sin darnos cuenta, una proposición en la forma de tipo de datos: la proposición falsa. El tipo de datos ⊥ no tiene habitantes. Dado que no existen pruebas para la proposición que es siempre falsa, este tipo resulta ser un buen modelo.

De forma similar, la proposición que es siempre verdadera tiene un elemento trivial.

```
data ⊤ : Set where
  tt : ⊤
```

Además, podemos construir la proposición de la negación. Como vemos en el código 1.16, la negación de una proposición modelada en el tipo A es la función de tipo $A \to \bot$. Esto se debe a que cuando una proposición es falsa carece de pruebas que la habiten y es sólo en ese caso posible construir una función (la función vacía) que vaya hacia el tipo vacío. En contrapartida, si queremos probar la negación de una proposición verdadera nos resulta imposible, puesto que tendríamos que proveer un elemento de ⊥. También resulta sensato desde el punto de vista de la lógica, donde la negación de un juicio suele expresarse como su implicancia a falso.

Código 1.16. *Negación de una proposición*

```
¬ : Set → Set
¬ A = A → ⊥
```

Una proposición más interesante es la de la equivalencia proposicional, introducida por Martin Löf [ML98, ML75], expuesta en el código 1.17. El tipo de datos _≡_ construye, a partir de un tipo de datos A y un habitante x, una familia de tipos indexada en A. Es decir, por cada elemento y de A, obtenemos el conjunto $x \equiv y$, que estará habitado por el elemento refl sólo cuando y sea igual a x.

Código 1.17. *Equivalencia proposicional*

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

A primera vista esta equivalencia puede parecer insuficiente pues a priori sólo nos permite probar la igualdad de elementos que son trivialmente iguales. Para entender cómo esta definición resulta útil para probar equivalencias no triviales, tenemos primero que comprender la forma en que Agda entiende a la igualdad *definicional* y su mecanismo de *unificación*.

Recordemos la definición de suma dada en el código 1.14. Allí dijimos que sumar el sucesor de un número con otro es *igual* al sucesor de la suma de ellos dos. Dicha igualdad es una igualdad *definicional* y Agda la considera trivial. Entonces podemos proveer una prueba de dicha aseveración de forma muy sencilla, simplemente con el constructor de la reflexividad, como se muestra en el siguiente código:

```
sumasuc₁ : ∀{x y : ℕ} → suma (suc x) y ≡ suc (suma x y)
sumasuc₁ = refl
```

Si quisiéramos probar, por el contrario, el siguiente juicio levemente diferente a la forma definicional, donde el sucesor se aplica al segundo argumento, Agda no considera que la prueba sea posible simplemente por reflexividad.

```
sumasuc₂ : ∀{x y : ℕ} → suma x (suc y) ≡ suc (suma x y)
```

En estos casos sacaremos provecho del mecanismo de unificación de Agda, y probaremos lemas que nos ayuden con las pruebas. A continuación vemos el llamado lema de congruencia, que asevera que dados dos elementos proposicionalmente iguales, lo serán también las respectivas aplicaciones de una misma función. Al hacer pattern matching sobre $x \equiv y$ nos encontramos con que el único habitante de dicho tipo será el constructor refl. En ese punto, Agda *unifica* dicha equivalencia y pasa considerar a x e y indistintos. Queda entonces pendiente una prueba trivial de la igualdad de $f\,x$ con $f\,y$.

Código 1.18. *Lema de congruencia proposicional*

```
cong : ∀{A B}{x y : A} → (f : A → B) → x ≡ y → f x ≡ f y
cong f refl = refl
```

Finalmente, podemos utilizar este lema en el segundo caso de pattern matching –dado que el primero resulta trivial– junto con la llamada recursiva sobre un elemento estructuralmente menor, para concluir la definición de sumasuc₂.

```
sumasuc₂ {zero} = refl
sumasuc₂ {suc x} = cong suc (sumasuc₂ {x})
```

Observemos que esto se realiza utilizando la llamada recursiva como hipótesis inductiva. Es decir, el valor sumasuc₂ {x} es prueba de la siguiente proposición:

$$\text{suma}\ x\ (\text{suc}\ y) \equiv \text{suc}\ (\text{suma}\ x\ y)$$

Al aplicar cong suc a dicho valor demostramos la siguiente equivalencia, que ya es definicionalmente equivalente a lo que buscamos probar:

$$\text{suc}\ (\text{suma}\ x\ (\text{suc}\ y)) \equiv \text{suc}\ (\text{suc}\ (\text{suma}\ x\ y))$$

Utilizando unificación es posible probar también que _≡_ es en efecto una relación de equivalencia. La prueba de la reflexividad no es nada más ni nada menos que el constructor canónico del tipo de datos. Queda pendiente probar la simetría y la transitividad. En el primer caso, una vez unificada la equivalencia entre x e y, resulta obvia la equivalencia simétrica. En el segundo caso, luego de unificar las equivalencias entre x e y y entre y y z y hacer que internamente Agda los considere tres elementos indistintos, la prueba de $x \equiv z$ resulta trivial.

```
sym : ∀{A : Set}{x y : A} → x ≡ y → y ≡ x
sym refl = refl

trans : ∀{A : Set}{x y z : A} → x ≡ y → y ≡ z → x ≡ z
trans refl refl = refl
```

Hemos demostrado cómo en un lenguaje de tipos dependientes como lo es Agda es posible expresar propiedades en la forma de tipos, cuyos habitantes serán pruebas de la veracidad de la proposición. Esta particularidad resulta ser un recurso muy útil para expresar propiedades genéricas sobre las construcciones. La metodología consiste en definir un tipo de datos parametrizado que exprese la propiedad o conjunto de propiedades que

se quiera modelar. Probar que la propiedad se cumple en una construcción en particular se limitará a proveer instancias de dicho tipo de datos genérico.

Para mayor claridad, a continuación vemos una forma de modelar la propiedad de «ser una relación de equivalencia», para una relación dada _≈_. El modelo consiste en un record de tres campos, uno por cada una de las leyes a cumplirse, i.e. reflexividad, simetría y transitividad.

Código 1.19. *Formalización del concepto de relación de equivalencia*

```
record IsEquivalence {A : Set} (_≈_ : A → A → Set) : Set where
  field
    refl  : ∀{x} → x ≈ x
    sym   : ∀{x y} → x ≈ y → y ≈ x
    trans : ∀{x y z} → x ≈ y → y ≈ z → x ≈ z
```

Una habitante de dicho tipo de datos, instanciado en la relación _≡_, es prueba de que dicha relación es de equivalencia y se expone a continuación.

```
isEquivalence≡ : ∀{A} → IsEquivalence {A} _≡_
isEquivalence≡ = record
  { refl  = refl
  ; sym   = sym
  ; trans = trans }
```

1.9. Suma dependiente

La suma dependiente es un tipo de datos que se asemeja mucho al tipo de producto presentado en 1.12 pero donde el tipo de la segunda componente puede depender del valor de la primera.

Otra forma de entender a este tipo de datos es como la contracara del tipo de las funciones dependientes, que hemos presentado y utilizado ampliamente en esta sección. Una función dependiente retorna un valor de un tipo que depende del elemento argumento. Lo expresamos en Agda como $(x : A) \to B$ donde B podía depender de x. Para hacer esto más expreso aún, podríamos escribirlo como: $(x : A) \to B(x)$. Es decir que *para todo* valor de entrada computa un valor de salida con tipo dependiendo de ella. En teoría de tipos encontramos a esta construcción con el nombre de *producto dependiente*, simbolizado $\Pi x : A.B(x)$.

La *suma dependiente*, por su parte, codifica un tipo de datos para el cual *existe* un valor que lo conforme. Dicha existencia se expresa como un par, donde el primer valor es el testigo de la existencia del tipo de la segunda componente del par.

```
record Σ (A : Set) (B : A → Set) : Set where
  constructor _,_
  field
    proj₁ : A
    proj₂ : B proj₁
```

La cláusula syntax define sinónimos de tipos. En el código expuesto a continuación expresamos que en lugar de escribir $\Sigma\ A\ (\lambda x \to B)$, podemos escribir $\Sigma[x \in A]\ B$ y así

hacer explícito el hecho de que el elemento x tiene tipo A.

```
syntax Σ A (λ x → B) = Σ[ x ∈ A ] B
```

Para esclarecer el asunto, veamos el siguiente ejemplo que construye los elementos de tipo ℕ para los cuales existe una prueba de ser no negativos. Un habitante de dicho tipo será, por ejemplo, el número uno, junto con la prueba de ser positivo.

```
positiveNat : Set
positiveNat = Σ[ n ∈ ℕ ] (¬ (n ≡ zero))

one : positiveNat
one = (suc zero , λ ())
```

1.10. Equivalencia heterogénea

Cuando queremos referirnos a la equivalencia entre dos valores cuyos tipos son proposicionalmente pero no definicionalmente equivalentes nos encontramos en un problema. Para ejemplificar la cuestión, digamos que tenemos una función dependiente $f : (x : A) \to B(x)$ y dos valores $x, y : A$ proposicionalmente equivalentes. ¿Cómo probar para f el lema de congruencia si el tipo de retorno diferirá en cada lado de la igualdad? Es decir, a pesar de poseer una prueba de $x \equiv y$, no podremos probar que $f x \equiv f y$ pues estos dos elementos tienen tipos definicionalmente distintos: $B(x)$ y $B(y)$ respectivamente.

Una forma de lograr construir dicha prueba es a partir de una función de substitución como la que se muestra en el código 1.20. Esta función se encarga de retornar el mismo valor, pero con el tipo cambiado a la versión proposicionalmente equivalente.

Código 1.20. *Substitución para la equivalencia proposicional*

```
subst  :   ∀{A : Set}{P : A → Set}{x y}
         → x ≡ y → P x → P y
subst {P} refl p = p
```

Utilizar este recurso implica probar algo levemente diferente: siendo $pr : x \equiv y$, probamos la equivalencia subst pr $(f x) \equiv f y$, como podemos observar en el siguiente código.

Código 1.21. *Lema de congruencia con substitución*

```
scong  :   ∀ {A}{B : A → Set}{x y : A}
         → (pr : x ≡ y)
         → (f : (x : A) → B x ) → subst pr (f x) ≡  f y
scong refl f = refl
```

La substitución es un recurso válido, pero incómodo, sobre todo en definiciones encadenadas y recurrentes. Una forma más cómoda y elegante para lidiar con dicho inconveniente es utilizar una noción de equivalencia a priori más laxa: la equivalencia heterogénea[McB02].

Código 1.22. *Equivalencia heterogénea*

```
data _≅_ {A : Set} (x : A) : {B : Set} → B → Set where
  refl : x ≅ x
```

Observemos que el truco de esta definición es engañar al chequeador de tipos del lenguaje, promulgando definir la equivalencia entre elementos de tipos distintos, para luego definir como constructor a la misma cláusula de reflexividad que la equivalencia de Martin Löf, donde los tipos son el mismo.

El lema de congruencia para funciones de tipo dependiente es ahora posible de ser probado, como se muestra a continuación

Código 1.23. *Lema de congruencia heterogénea*

```
cong  :  ∀{A : Set}{B : A → Set}{x y}
      →  (f : (x : A) → B x) → x ≅ y → f x ≅ f y
cong f refl = refl
```

Podemos incluso probar una versión aún más general de congruencia: cuando las funciones a aplicar a cada lado de la equivalencia tienen tipos definicionalmente diferentes. Se prueba dicho lema más general en el código dcong.

Código 1.24. *Lema de congruencia heterogénea general*

```
dcong  :   {A A' : Set}{B : A → Set}{B' : A' → Set}
           (f : (a : A) → B a){f' : (a : A') → B' a}{a : A}{a' : A'}
       →   a ≅ a' → B ≅ B' → f ≅ f'
       →   f a ≅ f' a'
dcong f refl refl refl = refl
```

Otro ejemplo de lema posible de probar gracias a la introducción de la equivalencia heterogénea es aquél que nos ayuda con la prueba de la igualdad de para sumas dependientes. Para demostrar que un elemento de tipo $\Sigma\ A\ B$ es equivalente a otro de tipo $\Sigma\ A'\ B'$, nos basta probar que son equivalentes sus primeras componentes y el tipo y valor de las segundas, como se muestra en el código 1.25.

Código 1.25. *Equivalencia de habitantes de sumas dependientes*

```
dSumEq  :   {A A' : Set}{B : A → Set}{B' : A' → Set}
            {x : Σ A B}{y : Σ A' B'}
        →   proj₁ x ≅ proj₁ y → B ≅ B' → proj₂ x ≅ proj₂ y
        →   x ≅ y
dSumEq refl refl refl = refl
```

Al igual que hicimos para la equivalencia proposicional, podemos proveer una instancia del record IsEquivalence para el caso de la relación heterogénea, quedando así demostrado que dicha relación es en efecto una relación de equivalencia.

```
isEquivalence≅ : ∀{A} → IsEquivalence {A} (λ x y → x ≅ y)
isEquivalence≅ = record
  { refl   = refl
  ; sym    = sym
  ; trans  = trans }
```

Las pruebas de la validez de la simetría y transitividad son similares a las correspondientes al caso proposicional y se exponen a continuación:

```
sym : ∀{A B}{x : A}{y : B} →  x ≅ y → y ≅ x
sym refl = refl

trans : ∀{A B C}{x : A}{y : B}{z : C} → x ≅ y → y ≅ z → x ≅ z
trans refl refl = refl
```

1.11. Extensionalidad

Otro caso que nos interesa examinar es aquel caso donde se quiera probar equivalencia de funciones. Luego de haber definido cong para la equivalencia heterogénea, queda claro que es posible probar que $f x \cong f y$ siendo $x \cong y$. En un problema nos encontramos cuando queremos probar lo mismo sobre las funciones en lugar de los argumentos. Es decir, probar que $f \cong g$ a partir de saber que para todo x vale $f x \cong g\, x$. En el marco de la teoría de tipos de Agda no nos será posible demostrar esto y nos será necesario apelar al axioma de extensionalidad. Es decir, aquél que indica que dos funciones son iguales si siempre lo son al ser aplicadas al mismo argumento. Esto no es más ni menos que la definición de la igualdad entre funciones, lo que implica que resulte sensato agregar dicha propiedad en forma de axioma.

En Agda un axioma se expresa con la palabra reservada postulate. El siguiente código expone los postulados de extensionalidad para funciones con argumento explícito (ext) e implícito (iext):

Código 1.26. *Axiomas de extensionalidad*

```
postulate ext  :   {A : Set}{B B' : A → Set}
                   {f : ∀ a → B a}{g : ∀ a → B' a}
               →   (∀ a → f a ≅ g a)
               →   f ≅ g

postulate iext  :   {A : Set}{B B' : A → Set}
                    {f : ∀ {a} → B a}{g : ∀ {a} → B' a}
                →   (∀ {a} → f {a} ≅ g {a})
                →   (λ {a} → f {a}) ≅ (λ {a} → g {a})
```

Con el objetivo de ver en acción estos postulados, demostraremos que la función suma es equivalente a su versión infija expresada en el código 1.27

Código 1.27. *Versión infija de la suma de naturales*

```
_+_ : ℕ → ℕ → ℕ
zero     + y  = y
(suc x)  + y  = suc (x + y)
```

Probamos primero el requerimiento del postulado de extensionalidad, es decir, que las funciones son equivalentes al ser aplicadas a los mismos argumentos:

```
sumaEquivExt : ∀{x y} → suma x y ≅ x + y
sumaEquivExt {zero}   = refl
sumaEquivExt {suc x}  = cong suc (sumaEquivExt {x})
```

Finalmente, probamos su equivalencia, haciendo uso del postulado de extensionalidad para cada uno de sus argumentos:

```
sumaEquiv : suma ≅ _+_
sumaEquiv = ext (λ x → ext (λ y → sumaEquivExt {x} {y}))
```

El siguiente postulado de extensionalidad es un poco más general que el presentado previamente, dado que permite que las funciones que se desean probar equivalentes tengan tipos definicionalmente distintos, aunque proposicionalmente equivalentes.

Código 1.28. *Extensionalidad dependiente general*

```
postulate dext  :   {A A' : Set}{B : A → Set}{B' : A' → Set}
                    {f : ∀ a → B a}{g : ∀ a → B' a}
                →   (∀ {a a'} → a ≅ a' → f a ≅ g a')
                →   f ≅ g
```

1.12. Universos

En esta sección introduciremos la noción de universo con el objetivo de dar fundamento a la temática principal de este trabajo, que consiste ni más ni menos que en un universo en particular, el universo de containers. Comenzaremos con la definición conceptual, para luego exponer ejemplos que motivarán las ventajas y posibles usos de estas construcciones.

La posibilidad de crear universos es una característica que poseen los lenguajes con tipos dependientes de la que carecen lenguajes con tipos simples. En términos generales, un universo de tipos es simplemente una colección de tipos, cerrada bajo ciertos constructores.

Un claro ejemplo de universo de tipos ya fue presentado en la introducción y vuelto a mencionar en la sección 1.1. Para evitar una paradoja similar a la paradoja de Russell, los tipos de Agda se encuentran subdivididos en universos. Todo tipo pertenece a un universo en particular. Los tipos básicos, construidos de forma inductiva, pertenecen al primer nivel, Set_0 (o simplemente Set). Por su parte, Set_0 y los tipos construidos a partir de él, tienen tipo Set_1. Es así como se forma la siguiente jerarquía de universos:

$$\mathsf{Set} : \mathsf{Set}_1 : \mathsf{Set}_2 : \mathsf{Set}_3 \ldots$$

Utilizando notación Agda, diremos que un *universo* es un tipo U y una función de interpretación o *extensión* ⟦_⟧ tal que

```
U    : Set
⟦_⟧  : U → Set
```

La programación genérica tiene como objeto la construcción de algoritmos que actúen de forma genérica sobre un conjunto determinado de tipos. Pero el problema que surge al querer programar genéricamente, es que no podemos hacer pattern matching sobre el tipo Set. Por otra parte, el conjunto de tipos sobre el que se pretende trabajar es muchas veces un subconjunto de todos los tipos posibles. Podemos obtener ese subconjunto de forma precisa a partir de definir un universo de códigos sintácticos, uno por cada tipo a incluir, y una función de interpretación que vaya del tipo de los códigos hacia el tipo de los tipos, asignando a cada código, el tipo que representa.

A continuación vemos un ejemplo ilustrativo sencillo, un universo que contiene a los booleanos y a los números naturales.

Código 1.29. *Códigos para la definición del universo de tipos naturales y booleanos*

```
data U₁ : Set where
  booleanos  : U₁
  naturales  : U₁
```

Es posible definir funciones genéricas sobre un universo simplemente haciendo pattern matching en los códigos sintácticos. Definimos la extensión del universo de naturales y booleanos en el código 1.30.

Código 1.30. *Extensión del universo de los tipos naturales y booleanos*

```
⟦_⟧₁ : U₁ → Set
⟦ booleanos  ⟧₁ = Bool
⟦ naturales  ⟧₁ = ℕ
```

Podemos generalizar esta definición para incluir también la posibilidad de definir universos de constructores de tipos. Observemos el siguiente ejemplo de universo que nuclea a los constructores de listas y de árboles[1]

Código 1.31. *Códigos para la definición del universo de constructores*

```
data U₂ : Set where
  listas   : U₂
  arboles  : U₂
```

Código 1.32. *Extensión del universo de constructores*

```
⟦_⟧₂ : U₂ → Set → Set
⟦ listas   ⟧₂ = List
⟦ arboles  ⟧₂ = Tree
```

[1]Ver códigos 3.1 y 3.17 para la definición de los elementos List y Tree

Capítulo 2

Teoría de Categorías

> Otra ventaja que se buscaba con este invento era que sirviese como idioma universal para todas las naciones civilizadas, cuyos muebles y útiles son, por regla general, iguales o tan parecidos, que puede comprenderse fácilmente cuál es su destino. Y de este modo los embajadores estarían en condiciones de tratar con príncipes o ministros de Estado extranjeros para quienes su lengua fuese por completo desconocida.
>
> Los Viajes de Gulliver
> Jonathan Swift

Una primera aproximación a definir de qué trata la teoría de categorías es, en pocas palabras, la de álgebra abstracta de funciones abstractas [Awo10]. En términos epistemológicos se puede decir que la teoría de categorías se ha constituido a partir de un giro copernicano. Donde los conjuntos y sus elementos eran el foco de atención, ahora lo son las relaciones entre ellos.

Este cambio en el punto de vista vino acompañado de un nuevo lenguaje que posibilita hablar de cada construcción en términos más abstractos, constituyéndose así en una buena herramienta de entendimiento común. Así como la teoría de álgebras ha sabido instaurar su propio lenguaje basado en tuplas equipadas con reglas, o la teoría de conjuntos o de grafos instauraron una jerga propia, la teoría de categorías es un lenguaje en sí mismo; lenguaje que promueve un punto de vista distinto para analizar muchas construcciones de diversas ramas de la matemática, la lógica, las ciencias de la computación o la física teórica, como para mencionar algunas.

La razón por cual resulta interesante hablar en el lenguaje de las categorías es la fuerte conexión que encontramos entre dicha teoría, la teoría de tipos y el lambda cálculo, subyacentes a los lenguajes funcionales, Agda en particular. Una forma de percibir a grandes rasgos que dicha conexión no es casual, es observar que el lambda cálculo también hace énfasis en las funciones, es un cálculo de funciones que instaura reglas formales para su manipulación.

En esta sección expondremos el vocabulario básico de la teoría de categorías. A diferencia de lo que podríamos encontrar en cualquier libro introductorio al tema, secundaremos la exposición de cada nuevo concepto con formalizaciones en Agda y ejemplos que apunten a una mayor comprensión de las secciones subsiguientes. La formalización está basada en la realizada por Altenkirch *et al* [ACU14].

2.1. Categoría

Definición 2.1. Una *categoría* $\mathscr{C}$ consiste de los siguientes elementos

- Una colección de **objetos**, denominada $|\mathscr{C}|$.
- Por cada par de objetos A, B, un conjunto $\mathscr{C}(A, B)$, a cuyos miembros denominamos **morfismos** de A en B. En general notaremos $A \xrightarrow{f} B$ o bien $f : A \to B$ a la sentencia $f \in \mathscr{C}(A, B)$.

  Para un morfismo $A \xrightarrow{f} B$ decimos que A es el *dominio* y B el *codominio* de f.
- Por cada objeto A, un morfismo $A \xrightarrow{id_A} A$ denominado **identidad** de A.
- Por cada terna de objetos A, B, C una **ley de composición** que asocia a cada par de morfismos $A \xrightarrow{f} B, B \xrightarrow{g} C$ un morfismo $A \xrightarrow{g \circ f} C$ denominado composición de g con f.

satisfaciendo las leyes:

⋆ **Leyes de identidad:** Dado un morfismo $A \xrightarrow{f} B$, se cumple:

$$id_B \circ f = f \qquad f \circ id_A = f$$

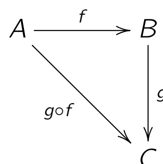
⋆ **Ley de asociatividad:** Dados morfismos $A \xrightarrow{f} B, B \xrightarrow{g} C$ y $C \xrightarrow{h} D$ se cumple:

$$h \circ (g \circ f) = (h \circ g) \circ f$$

Ejemplo 2.2. Un ejemplo importante de categoría, debido a su ubicuidad y a ser fuente de intuición a lo largo del trabajo, es el de la categoría cuyos objetos son conjuntos y cuyos morfismos son funciones totales. La llamamos **Set**. Los morfismos identidad son las funciones identidad y la composición de morfismos es la composición de funciones. Para completar la presentación de esta categoría, resta mostrar que las leyes de identidad y asociatividad se cumplen.

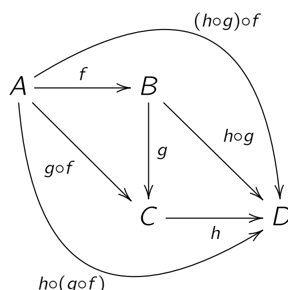
Ejemplo 2.3. Otro ejemplo que nos resultará de utilidad en las próximas secciones es el de la categoría *dual*, que se obtiene de invertir todos los morfismos. Es decir, dada una categoría $\mathscr{C}$, su categoría dual $\mathscr{C}^{op}$ tiene los mismos objetos que $\mathscr{C}$ y por cada morfismo $f \in \mathscr{C}(A, B)$, un morfismo
$f \in \mathscr{C}^{op}(B, A)$.

Para expresar leyes en teoría de categorías, es usual utilizar diagramas en forma de grafos, donde los objetos de la categoría involucrados se representan como vértices del grafo y los morfismos como aristas. Por ejemplo, dados los objetos A, B, C y los morfismos $A \xrightarrow{f} B$ y $B \xrightarrow{g} C$ en una categoría $\mathscr{C}$, sabemos que es posible componerlos obteniendo un morfismo $g \circ f$. En forma de diagrama:



Diremos que un diagrama es *conmutativo* cuando para cada par de vértices A, B del diagrama, todos los caminos posibles entre A y B son iguales.

Por ejemplo, el siguiente diagrama expresa la ley de asociatividad expuesta en la definición 2.1.



2.1.1. Formalización

A continuación exponemos el concepto de categoría formalizado en Agda, como un tipo de datos en forma de record. Como ya hemos expuesto en la sección dedicada a presentar el lenguaje Agda, es posible representar estructuras algebraicas abstractas como tipos de datos que incluyen no sólo el conjunto portador del álgebra sino también las operaciones y leyes que todos ellos deben cumplir. En el caso de las categorías, tendremos el tipo de los objetos Obj y para cada par de objetos A y B, el conjunto de morfismos con origen en A y destino en B, Hom A B. Además, contamos con un morfismo identidad iden para cada objeto y un operador de composición para cada par de morfismos posibles de ser compuestos. Finalmente, deben cumplirse las dos leyes de identidad idl e idr y la ley de asociatividad ass.

Código 2.4. *Formalización en Agda del concepto de categoría*

```
record Category : Set where
  field Obj  :  Set
        Hom  :  Obj → Obj → Set
        iden :  ∀{A} → Hom A A
        comp :  ∀{A B C} → Hom B C → Hom A B → Hom A C
        idl  :  ∀{A B}{f : Hom A B} → comp iden f ≅ f
        idr  :  ∀{A B}{f : Hom A B} → comp f iden ≅ f
        ass  :  ∀{A B C D}{f : Hom A B}{g : Hom B C}{h : Hom C D}
             →  comp h (comp g f) ≅ comp (comp h g) f
```

Los ejemplos 2.2 y 2.3 expuestos informalmente los podemos expresar como habitantes del tipo de datos Cat y así garantizar formalmente que dichas construcciones son en efecto categorías. En el código 2.5 vemos la categoría **Set** de conjuntos y funciones totales. Observemos que ahora sí exponemos las demostraciones de las leyes que requiere la definición de categoría. En este caso, las pruebas resultan triviales.

Código 2.5. *Categoría* **Set** *de conjuntos y funciones*

```
Set : Category
Set = record
```

```
  { Obj    = Set
  ; Hom    = λ R S → R → S
  ; iden   = id
  ; comp   = _∘_
  ; idl    = refl
  ; idr    = refl
  ; ass    = refl
  }
```

En el código 2.6 se muestra la construcción de la categoría dual de una categoría dada $\mathscr{C}$. Podemos ver que tanto los objetos como las identidades y sus leyes se mantienen iguales a la categoría $\mathscr{C}$, mientras que los morfismos se revierten, las composiciones se espejan y la ley de asociatividad se prueba por simetría a partir de la asociatividad en $\mathscr{C}$.

Código 2.6. *Categoría dual*

```
_ ᵒᵖ : Category → Category
𝒞 ᵒᵖ = record
  { Obj    = Obj 𝒞
  ; Hom    = λ A B → Hom 𝒞 B A
  ; iden   = iden 𝒞
  ; comp   = λ {A} {B} {C} f g → comp 𝒞 g f
  ; idl    = idr 𝒞
  ; idr    = idl 𝒞
  ; ass    = sym (ass 𝒞)
  }
```

2.2. Isomorfismo

Definición 2.7. Un morfismo $A \xrightarrow{f} B$ en una categoría $\mathscr{C}$ es un *isomorfismo* siempre que exista un morfismo $B \xrightarrow{g} A$ de $\mathscr{C}$ que satisfaga

$$f \circ g = id_B \qquad g \circ f = id_A$$

Si existe un isomorfismo entre dos objetos A, B de una categoría, decimos que son *isomorfos* y notamos $A \simeq B$.

2.2.1. Formalización

Código 2.8.

```
record IsIsomorphism {𝒞 : Category}{A B : Obj 𝒞}
                     (f : Hom 𝒞 A B) : Set where
  field
    inverse : Hom 𝒞 B A
    proof : isInverse {𝒞} f inverse
```

Es decir, un morfismo f es un isomorfismo entre los objetos A y B si existe un morfismo inverse, tal que las diversas composiciones resultan ser las respectivas identidades. Este

último requerimiento se expresa en el campo proof, instancia del record isInverse, definido por el siguiente código:

```
record isInverse  {𝒞 : Category}{A B}
                  (f : Hom 𝒞 A B)
                  (g : Hom 𝒞 B A) : Set where
  field
    invl : comp 𝒞 g f ≅ iden 𝒞 {A}
    invr : comp 𝒞 f g ≅ iden 𝒞 {B}
```

2.3. Funtor

Si hubiéramos elegido utilizar un lenguaje netamente algebraico y haber descripto a las categorías como un álgebra con dos portadores, dos operaciones y tres leyes, podríamos decir que esta sección se dedica a definir los homomorfismos de categorías. Definiremos a continuación el concepto de funtor como aquella operación entre categorías que preserva su estructura interna.

Definición 2.9. Sean las categorías $\mathscr{C}$ y $\mathscr{D}$. Un *funtor* F de $\mathscr{C}$ a $\mathscr{D}$, que notamos $F : \mathscr{C} \to \mathscr{D}$ consiste de:

- Un mapeo que asigna a cada objeto A de $\mathscr{C}$ un objeto $F(A)$ de $\mathscr{D}$.
- Un mapeo que asigna a cada morfismo $A \xrightarrow{f} B$ de $\mathscr{C}$ un morfismo $F(A) \xrightarrow{F(f)} F(B)$ de $\mathscr{D}$.

satisfaciendo las leyes:

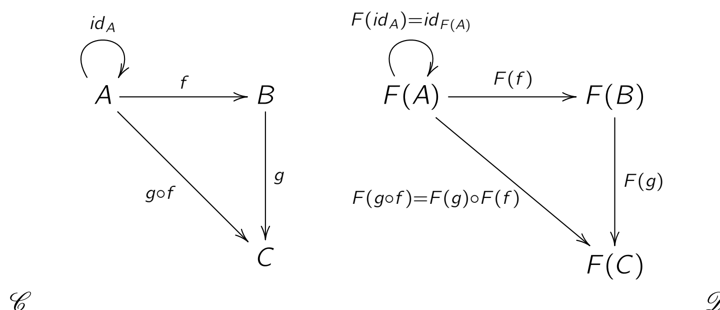
⋆ **Ley de preservación de la identidad:** Por cada objeto A de $\mathscr{C}$ tenemos la siguiente igualdad entre morfismos en $\mathscr{D}$:

$$F(id_A) = id_{F(A)}$$

⋆ **Ley de preservación de la composición:** Por cada par de morfismos $A \xrightarrow{f} B, B \xrightarrow{g} C$ de $\mathscr{C}$, se cumple la siguiente equivalencia:

$$F(g \circ f) = F(g) \circ F(f)$$

En términos gráficos, podemos decir que un funtor $F : \mathscr{C} \to \mathscr{D}$ hace conmutar al siguiente diagrama:



Un ejemplo trivial de funtor es el del funtor identidad, que mapea cada objeto y cada morfismo de la categoría, a sí mismo. Es decir, ambos mapeos resultan ser las identidades sobre objetos y morfismos. Trivialmente cumple con las leyes de preservación. Por otro lado, dados dos funtores $F : \mathscr{A} \to \mathscr{B}$ y $G : \mathscr{B} \to \mathscr{C}$, es posible definir la composición de funtores $G \circ F : \mathscr{A} \to \mathscr{C}$, componiendo individualmente cada mapeo. Podemos intuir de estas conclusiones que las categorías y los funtores forman una nueva categoría. El problema que tenemos aquí es análogo a la famosa paradoja de Russell, ¿esta nueva categoría se tiene por objeto a sí misma? ¿podríamos entonces construir la categoría de las categorías que no se contienen a sí mismas?. Para evitar este problema, seguimos el mismo criterio que se ha tomado en teoría de conjuntos:

Definición 2.10. Una categoría $\mathscr{C}$ es *pequeña* cuando la colección $|\mathscr{C}|$ de objetos es un conjunto.

Ejemplo 2.11. La categoría **Cat** tiene por objetos a las categorías pequeñas y por morfismos a los funtores.

Ejemplo 2.12. Dado un objeto A en una categoría $\mathscr{C}$, el funtor $Hom_{\mathscr{C}}(A, _)$ desde la categoría $\mathscr{C}$ hacia la categoría **Set**:

- mapea cada objeto B de $\mathscr{C}$ hacia el conjunto de morfismos $Hom_{\mathscr{C}}(A, B)$
- mapea cada morfismo $B \xrightarrow{f} B'$ hacia la función de pre-composición

$$f \circ _ : Hom_{\mathscr{C}}(A, B) \to Hom_{\mathscr{C}}(A, B')$$

Definición 2.13. Un funtor F se dice *contravariante* si revierte la dirección de los morfismos, es decir, mapea cada $A \xrightarrow{f} B$ a un morfismo $F(B) \xrightarrow{F(f)} F(A)$.

En oposición, un funtor que respeta el sentido de los morfismos se denomina *covariante*.

En lugar de presentar a un funtor contravariante $F : \mathscr{C} \to \mathscr{D}$, consideraremos que los funtores son siempre covariantes, y diremos $F : \mathscr{C}^{op} \to \mathscr{D}$

Ejemplo 2.14. Similarmente al ejemplo 2.12, dado un objeto B en una categoría $\mathscr{C}$, el funtor $Hom_{\mathscr{C}}(_, B) : \mathscr{C}^{op} \to$ **Set**

- mapea cada objeto A de $\mathscr{C}$ hacia el conjunto de morfismos $Hom_{\mathscr{C}}(A, B)$
- mapea cada morfismo $A \xrightarrow{f} A'$ de la categoría dual hacia la función de post-composición

$$_ \circ f : Hom_{\mathscr{C}}(A, B) \to Hom_{\mathscr{C}}(A', B)$$

2.3.1. Formalización

En el código 2.15 podemos observar la implementación en Agda del concepto de funtor. Los campos FObj y FHom son los respectivos mapeos entre objetos y morfismos. Los otros campos expresan las leyes de preservación de las identidades y composiciones.

Código 2.15. *Formalización del concepto de funtor en Agda*

```
record Fun (𝒞 : Category)(𝒟 : Category) : Set where
  field
```

```
FObj    :  Obj 𝒞 → Obj 𝒟
FHom    :  ∀{A B}
        →  Hom 𝒞 A B → Hom 𝒟 (FObj A) (FObj B)
idenPr  :  ∀{A}
        →  FHom (iden 𝒞 {A}) ≅ iden 𝒟 {FObj A}
compPr  :  ∀{A B C}{f : Hom 𝒞 A B}{g : Hom 𝒞 B C}
        →  FHom (comp 𝒞 g f) ≅ comp 𝒟 (FHom g) (FHom f)
```

Código 2.16. Formalizamos el funtor expuesto en el ejemplo 2.14 en Agda. Lo llamamos Hom_1.

```
Hom₁ :  ∀{𝒞 : Category }
     →  (B : Obj 𝒞)
     →  Fun (𝒞 ᵒᵖ) Set
Hom₁ {𝒞} B = record
  { FObj    = λ A → Hom 𝒞 A B
  ; FHom    = λ f g → comp 𝒞 g f
  ; idenPr  = ext (λ _ → idl (𝒞 ᵒᵖ))
  ; compPr  = ext (λ _ → sym (ass (𝒞 ᵒᵖ)))}
```

Dada una categoría $\mathscr{C}$, llamamos *endofuntores* en $\mathscr{C}$ a los funtores $F : \mathscr{C} \to \mathscr{C}$. En **Set** son endofuntores ciertos constructores de tipos, como por ejemplo, el constructor de listas, expuesto en el siguiente ejemplo. Algunos ejemplos más de constructores de tipos son los árboles, los streams, las matrices. Estos constructores tienen una característica común que los convierte en funtores y es que son factibles de ser *mapeados*. Es decir, dada una función f de un tipo A en un tipo B, si tengo por ejemplo, un árbol con elementos de tipo A, puedo obtener fácilmente un árbol de elementos de tipo B simplemente aplicando la función f a cada elemento almacenado. Veremos en el siguiente ejemplo cómo esta función de mapeo se constituye en la componente de mapeo de morfismos del funtor, mientras que el constructor de datos en sí mismo será la componente del funtor de mapeo de objetos.

Ejemplo 2.17. El constructor de tipos List se define en Agda como:

```
data List (A : Set) : Set where
  nil : List A
  cons : A → List A → List A
```

Siempre que contemos con una lista de un tipo A cualquiera, y una función f que vaya de A en B, podemos obtener una lista de elementos de B simplemente mapeando f sobre los elementos de la lista. A continuación se expone la implementación de dicha función de mapeo.

```
map : ∀{A B} → (A → B) → List A → List B
map f nil = nil
map f (cons x l) = cons (f x) (map f l)
```

Nos encontramos frente a un funtor, el siguiente código muestra algunos detalles de su construcción. Las pruebas formales del cumplimiento de las leyes de funtor no se exponen pero se pueden encontrar en el código que acompaña a este trabajo.

```
ListFun : Fun Set Set
ListFun = record
  { FObj    = List
```

```
; FHom    = map
; idenPr  = ext idenPrList
; compPr  = ext compPrList }
```

2.4. Transformación natural

Definición 2.18. Sean $\mathscr{C}$ y $\mathscr{D}$ dos categorías y sean los funtores $F, G : \mathscr{C} \to \mathscr{D}$. Una *transformación natural* η de F hacia G, que notamos $\eta : F \Rightarrow G$ es una familia $(\eta_A : F(A) \to G(A))_{A \in |\mathscr{C}|}$ de morfismos de $\mathscr{D}$ indexada por objetos de $\mathscr{C}$ tal que la siguiente ley se satisface:

- ⋆ **Ley de naturalidad:** Por cada morfismo $A \xrightarrow{f} B$ de $\mathscr{C}$

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

Es decir, el siguiente diagrama conmuta:

$$\begin{array}{ccc} F(A) & \xrightarrow{\eta_A} & G(A) \\ \downarrow{\scriptstyle F(f)} & & \downarrow{\scriptstyle G(f)} \\ F(B) & \xrightarrow[\eta_B]{} & G(B) \end{array}$$

Llamamos $Nat(F, G)$ al conjunto de las transformaciones naturales entre dos funtores F y G.

2.4.1. Formalización

Código 2.19. *Formalización del concepto de transformación natural en Agda*

```
record NatT {𝒞 𝒟}(F G : Fun 𝒞 𝒟) : Set where
  constructor _,_
  field
    η   :  ∀{A} → Hom 𝒟 (FObj F A) (FObj G A)
    nat :  ∀{A B}{f : Hom 𝒞 A B}
        →  comp 𝒟 (FHom G f) η ≅ comp 𝒟 η (FHom F f)
```

Definición 2.20. Sean los funtores $F, G : \mathscr{C} \to \mathscr{D}$. Decimos que una transformación natural $\eta : F \Rightarrow G$ es un *isomorfismo natural* cuando cada una de las componentes η_X de la transformación es un isomorfismo en la categoría $\mathscr{D}$.

Código 2.21. *Isomorfismo natural*

```
record NaturalIsomorphism {𝒞 𝒟}{F G : Fun 𝒞 𝒟} : Set where
  field
    n   : NatT F G
    n⁻¹ : ∀{A} → IsIsomorphism {𝒟} (η n {A})
```

2.5. Último ejemplo y motivación de Containers

Para concluir la sección dedicada a teoría de categorías, mostraremos formalmente cómo es posible formar una categoría que llamaremos **Fun**, cuyos objetos son funtores y sus morfismos, transformaciones naturales.

Ejemplo 2.22. **Fun** : Category → Category → Category

```
Fun 𝒞 𝒟 = record
  { Obj   = Fun 𝒞 𝒟
  ; Hom   = NatT
  ; iden  = natIden
  ; comp  = natComp
  ; idl   = NatT≅ (idl 𝒟)
  ; idr   = NatT≅ (idr 𝒟)
  ; ass   = NatT≅ (ass 𝒟) }
```

Dadas dos categorías, podemos formar la categoría **Fun**, cuyos objetos son los funtores entre ellas y cuyos morfismos, las transformaciones naturales. Para demostrar que **Fun** es una categoría, debe existir para cada objeto, un morfismo identidad. En otras palabras, para cada funtor debe existir una transformación natural identidad. Además, siempre deberá existir la composición de transformaciones naturales aptas de ser compuestas. El morfismo identidad está dado por la construcción natIden, expuesta en el código 2.23. La composición de transformaciones naturales también será siempre posible, hecho reflejado en la construcción natComp, que no se expone, pero que puede encontrarse en el código asociado a esta tesina.

Código 2.23. *Transformación natural identidad*

```
natIden : ∀{𝒞 𝒟}{F : Fun 𝒞 𝒟} → NatT {𝒞 = 𝒞} {𝒟} F F
natIden {𝒞 = 𝒞} {𝒟} {F} = record
  { η   = iden 𝒟
  ; nat = trans (idr 𝒟) (sym $ idl 𝒟)}
```

La prueba de las identidades y asociatividad para la categoría **Fun** recae en las identidades y asociatividad de la categoría de destino. Además, se hace uso de un lema que permite probar igualdad de dos transformaciones naturales a partir de la igualdad de la componente funcional de la transformación, puesto que la igualdad de pruebas es irrelevante. Los detalles del lema tampoco se exponen, pero podemos vislumbrar su utilidad a partir de observar su tipo, expuesto a continuación:

```
NatT≅ :   ∀{𝒞 𝒟}{F G : Fun 𝒞 𝒟}{α β : NatT F G}
       →  (∀{X} →  η α {X} ≅ η β {X}) → α ≅ β
```

El objetivo primordial de presentar este ejemplo es dar pie a la construcción del universo de containers, que se analizará en la parte II. Los containers son una categoría en sí misma, pero representa –funtor de extensión mediante– a una subcategoría de esta categoría de funtores.

Parte II

Formalización de Containers

Capítulo 3

Containers

> La intención de la neolengua no era solamente proveer un medio de expresión a la cosmovisión y hábitos mentales propios de los devotos del Ingsoc, sino también imposibilitar otras formas de pensamiento. Lo que se pretendía era que una vez la neolengua fuera adoptada de una vez por todas y la vieja lengua olvidada, cualquier pensamiento herético, es decir, un pensamiento divergente de los principios del Ingsoc, fuera literalmente impensable, o por lo menos en tanto que el pensamiento depende de las palabras.
>
> 1984
> George Orwell

Los containers [AAG03, AAG05], así como muchas otras construcciones matemáticas abstractas, tienen la particularidad de poder ser pensados desde diferentes puntos de vista. Podemos ponernos nuestros *lentes de teoría de tipos* o nuestros *lentes de teoría de categorías* y visualizar cada construcción en marcos diferentes. En cada caso, lo que cambiará será fundamentalmente el lenguaje en el que se presentan los conceptos y necesariamente, la relación con el mundo teórico que los rodea, los precede y explica.

En este capítulo trataremos de presentar a los containers como una forma particular de representar ciertos tipos paramétricos. Para comenzar, utilizaremos nuestros *lentes Agda* de programador funcional-dependiente. La elección de este recurso implica optar por un punto de vista *más concreto* (a.k.a. menos abstracto que la visión categórica) y por lo tanto a priori más didáctico, pero sin perder el nivel de correctitud. Correctitud obtenida gracias a las garantías que nos provee un lenguaje-de-programación/demostrador-de-teoremas como lo es Agda.

El universo de containers resulta muy apropiado a la hora de programar genéricamente [AMM07] y cobra relevancia al momento de poder representar constructores de tipos funtoriales, en particular, los generadores de los tipos estrictamente positivos. El universo de morfismos de containers, por su parte, tiene el poder expresivo para representar un subconjunto muy interesante de funciones: las que definen transformaciones naturales.

Comenzaremos la exposición, sin embargo, con un ejemplo en lenguaje coloquial, con la intención de presentar la idea de la forma más intuitiva posible. Para continuar, definiremos formalmente los conceptos de container y de extensión y ejemplificaremos su potencialidad, a partir de expresar en estos nuevos términos algunos exponentes ilustres de tipos paramétricos. Luego analizaremos la manera de expresar funciones entre tipos paramétricos, definiendo los morfismos de containers.

Finalmente, cambiaremos el punto de vista, y expresaremos la forma en la que los containers constituyen una categoría, formalizando en Agda la categoría **Cont**.

3.1. Motivación

Los tipos de datos que nos interesa modelar son aquellos que contienen valores de otros tipos de datos. Como por ejemplo, las listas, los árboles, los streams. Decimos que son tipos de datos parametrizados por otro tipo de datos, el tipo de los valores que almacenan. Les llamaremos también *constructores de tipos* porque construyen un nuevo tipo a partir de otro dado. No obstante el tipo de los valores que se almacenan es tomado como parámetro por el constructor, la estructura de este último no depende del primero.

Enfoquémonos por ejemplo en el tipo de las listas, definido en Agda de forma inductiva:

Código 3.1. Listas

```
data List (X : Set) : Set where
  nil :  List X
  cons : X → List X  → List X
```

El elemento nil es un habitante de List X, para todo X dado y representa la lista vacía. De forma similar, cons construye una lista agregando un elemento de X a una lista ya dada de elementos de X. A pesar de no conocer nada sobre el tipo X, podemos definir List X, indicio de que la esencia de List es independiente de cualquier particularidad de X. Vemos aquí presente una potencial capacidad de segregar estructura de contenido. Es este potencial el que explotaremos para presentar una forma alternativa de representación.

Es posible pensar a ciertos tipos de datos como un esqueleto, un contenedor, a llenar con datos. Las listas como esqueletos se pueden expresar visualmente de la siguiente forma:

$$[] \quad [\bigcirc] \quad [\bigcirc, \bigcirc] \quad [\bigcirc, \bigcirc, \bigcirc] \quad [\bigcirc, \bigcirc, \bigcirc, \bigcirc] \quad \text{etc.}$$

Figura 3.1: Containers de lista

Llamaremos *containers* a estos esqueletos. Para dar una lista como un container, proveemos dos coordenadas: por un lado la *forma* que tiene el container, y para cada forma, un conjunto de *posiciones*.

Si observamos la figura 3.1, advertimos que el conjunto de los números naturales representan de forma unívoca a cada forma de lista: la forma de una lista es su longitud. Fijada una forma n, cada $\bigcirc$ representa un *agujero* destinado a ser llenado con datos. Es decir, una posición. El conjunto de posiciones posibles será el de los números naturales entre 0 y $n - 1$. Por ejemplo, para el caso de una lista de longitud tres tenemos:

$$[\bigcirc_0, \bigcirc_1, \bigcirc_2]$$

donde los subíndices identifican las posiciones en la lista.

Al conjunto de naturales menores a n lo definimos como un nuevo tipo de datos Fin, indexado en n.

```
data Fin : ℕ → Set where
  zero :  ∀{n} → Fin (n + 1)
  suc  :  ∀{n} → Fin n → Fin (n + 1)
```

Notar que Fin 0 es el tipo vacío, ya que no tenemos ningún constructor que lo habite. Por otra parte, Fin $(n + 1)$ tiene tantos elementos como Fin n, gracias al constructor suc y uno más, el elemento zero. En conclusión, Fin n tiene n elementos. Coloquialmente decimos: Fin $n = \{0, \ldots, n - 1\}$.

Finalmente, nos queda pendiente proveer un mecanismo para "llenar" un esqueleto de lista. Para ello, fijada una forma de lista, nos basta con una función de posiciones en valores. Por ejemplo, para la lista $[x_1, x_2, x_3]$ de tipo X, necesitamos una función que asigne a cada posición, los valores correspondientes de X, como muestra la figura 3.2:

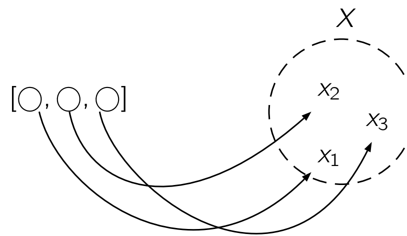


Figura 3.2: Llenando un container de lista

Generalizando, llenar un container con valores de un tipo X, es proveer una función que asigne valores de X a cada posición, para cada forma. Llamaremos a esta función, la *extensión* del container.

3.2. Definición y extensión

A continuación expondremos formalmente esta nueva forma de representación. Para dar una instancia de un nuevo constructor de tipo proveeremos por un lado su estructura en la forma de un *container*. Luego, para permitir "llenar" al container con datos de tipo X, calcularemos su *extensión*.

3.2.1. Definición

Definición 3.2. Un *container* es un par (Sh, Pos) que notamos $Sh \triangleleft Pos$ donde

- Sh es un conjunto de *formas* (o *shapes* en inglés).
- Pos es es un conjunto de posiciones, para cada forma s en Sh. Es decir, es una familia de posiciones indexada por las formas.

En Agda definimos:

Código 3.3. *Containers*

```
record Cont : Set₁ where
  constructor _◁_
  field
```

```
Sh  : Set
Pos : Sh → Set
```

Recordemos que en Agda un record es una tupla donde se le asigna nombre a cada componente y, opcionalmente, un constructor de instancias. Los nombres asignados a cada componente hacen a su vez las veces de destructores del record. En el caso de Cont, construimos sus instancias llamando a _◃_ con un S : Set y un P : S → Set. Notar que el tipo de los componentes de un record puede depender de componentes anteriores.

Los nombres de campos Sh y Pos del record Cont definen funciones de destrucción, con tipos:

```
Sh : Cont → Set

Pos : (C : Cont) → Sh C → Set
```

de forma tal que para todo C : Cont siempre valga la siguiente igualdad definicional: Sh C ◃ Pos C = C .

Ejemplo 3.4. Volviendo al ejemplo de la sección 3.1, el container que representa a las listas está dado por el elemento cList, definido como:

```
cList : Cont
cList = ℕ ◃ Fin
```

Aludiendo a lo expuesto anteriormente, las listas son aquél tipo cuya forma está dada por su longitud. Es decir, el conjunto de todas las formas posibles será el conjunto de los números naturales y el cero. Fijada una longitud n, el conjunto de posiciones será el de todos los enteros entre 0 y n - 1. Es decir, es el n-ésimo conjunto finitario.

3.2.2. Extensión

Hasta ahora hemos expuesto la forma en que el universo sintáctico de containers representa a ciertos constructores de tipos. Algunas preguntas que caben hacerse son: ¿Qué constructores son posibles de ser representados como containers? En otras palabras, ¿a qué constructores representan los containers? ¿Cómo estar seguros, más allá de la apelación a la intuición, que un container representa a un constructor determinado?

Antes de contestar esas, entre otras cuestiones, trataremos de dar respuesta a una pregunta más elemental y por lo tanto de más sutil respuesta: ¿Qué es un constructor de tipos? Una idea intuitiva fue ofrecida al comenzar el capítulo: un constructor de tipos construye un tipo a partir de otro dado genéricamente, en forma de parámetro. Para ejemplificar dicha aseveración, dimos el elemento List, en la definición 3.1. Si observamos dicha definición podemos apreciar que el constructor de tipos List tiene el siguiente tipo:

```
List : Set → Set
```

Otro constructor de tipos posible podría ser, por ejemplo, el que construye pares de elementos de dos tipos distintos:

```
data Pair (X : Set) (Y : Set) : Set where
  _,_ : X → Y → Pair X Y
```

Un ejemplo más, es el de las funciones de un tipo arbitrario en sí mismo. Es decir, el tipo de datos Endo X, expuesto en el siguiente código, engloba a todas las funciones

con tipo $X \to X$.

```
data Endo (X : Set) : Set where
  fun : (X → X) → Endo X
```

Tanto el constructor de tipos Pair, como el constructor Endo no serán posibles de ser representados como containers. Esto se debe a que restringimos el análisis a *ciertos* constructores de tipo. Por simplicidad, el tipo de datos Pair no será contemplado, puesto que nos limitamos a constructores de un sólo argumento. Sin embargo, toda la teoría de containers se extiende fácilmente a constructores de más argumentos. En tal caso sí incluiríamos al tipo de los pares. Por otro lado, tampoco podremos representar al tipo Endo puesto que nos conciernen aquellos constructores que cumplen ciertas propiedades y a los que llamaremos *funtores*.

Definición 3.5. Un constructor de tipos $\mathsf{F} : \mathsf{Set} \to \mathsf{Set}$ es un *funtor* [RP90] si existe una función de mapeo

$$\mathsf{map} : \forall \{A\ B\} \to (A \to B) \to \mathsf{F}\ A \to \mathsf{F}\ B$$

tal que

$$\begin{cases} \mathsf{map}\ \mathsf{id} = \mathsf{id} \\ \mathsf{map}\ f \circ \mathsf{map}\ g = \mathsf{map}\ (f \circ g) \qquad \forall\, f : Z \to B,\ g : A \to Z \end{cases}$$

Es decir, un funtor es un constructor F que cumple con ciertas propiedades extra. Siempre que exista una función entre los tipos A y B habrá una función entre $\mathsf{F}\ A$ y $\mathsf{F}\ B$ de forma tal que las identidades y composiciones se preserven.

El constructor de pares Pair no será funtor sino hasta que fijemos el tipo de uno de los dos elementos del par. Por ejemplo, fijando el primer tipo al de los naturales se obtiene:

```
PairℕX : Set → Set
PairℕX = Pair ℕ
```

En el caso de Endo no es posible definir una función map que cumpla con los requisitos de la definición.

Retomando nuestra exposición, presentaremos a continuación la noción de *extensión* de un container, que hará posible volver a interpretarlos como constructores de tipos en su forma funtorial.

Si cF : Cont es un container, entonces su extensión, que notamos $[\![\ cF\ ]\!]$, será un funtor:

$$[\![\ cF\ ]\!] : \mathsf{Set} \to \mathsf{Set}$$

Intuitivamente, la extensión permitirá llenar un container con valores del tipo argumento. En caso en que necesitemos precisión en la exposición, llamaremos *funtor container* a un funtor obtenido de calcular la extensión a un container.

Definición 3.6. La *extensión* del container $S \triangleleft P$ sobre el conjunto X es el conjunto de pares (s, f) donde $s \in S$ y $f : Ps \to X$. Lo notamos $[\![\ S \triangleleft P\ ]\!]$.

Dado un caso particular de forma, tenemos un conjunto de posiciones. La extensión requiere de una función de posiciones en valores.

Código 3.7. *Extensión de containers*

```
⟦_⟧ : Cont → Set → Set
⟦ S ◃ P ⟧ X = Σ[ s ∈ S ] (P s → X)
```

Dicho de otra forma, ⟦ $S \triangleleft P$ ⟧ define familias de funciones $Ps \to X$ de posiciones en valores por cada forma s del conjunto S.

Ejemplo 3.8. Entonces, para completar nuestro ejemplo de las listas, podemos finalmente calcular su extensión y así volver al «mundo real».

```
List : Set → Set
List = ⟦ cList ⟧
```

Incluso es posible definir los constructores originales del tipo List dado inductivamente:

```
nil : ∀{X} → List X
nil = 0 , (λ ())

cons : ∀{X} → X → List X → List X
cons x (s , f) = suc s , (λ { zero → x ; (suc i) → f i })
```

Dado que la forma de una lista es su longitud, el valor nil construye un elemento con forma igual a 0. La función que asigna valores a posiciones es la función vacía, representada en Agda por (λ ()), ya que la forma nula no tiene posiciones. Por su parte, la función cons x construye una lista cuya forma es el sucesor de la lista argumento, puesto que cons aumenta su longitud en uno. Además, ahora la posición zero mapea al nuevo elemento x, desplazando al resto.

A pesar de haber sostenido que los containers nos proveen una forma de construir funtores recurriendo al cálculo de su extensión, nos queda pendiente evidenciar que esto es cierto. Intuitivamente, si un container es un esqueleto donde se guardarán datos, resulta lógico que los datos almacenados puedan ser modificados sin tocar ese esqueleto que los contiene. A continuación definimos la función map de containers. Queda pendiente demostrar que dicha función cumple con los requerimientos de la definición 3.5

Definición 3.9. Sea f una función de X en Y y C un container. Se define el mapeo de f sobre el funtor container ⟦ C ⟧ como una nueva función de ⟦ C ⟧ X en ⟦ C ⟧ Y que preserva las formas y aplica f a los elementos almacenados, como se indica a continuación:

```
map : ∀{X Y C} → (f : X → Y) → ⟦ C ⟧ X → ⟦ C ⟧ Y
map f (s , p) = s , (f ∘Set p)
```

Así como vemos que la extensión de un container resulta ser un endofuntor en Set, y que lógicamente dichos funtores pueden componerse y obtener así otro funtor, también podemos componer containers antes de calcular su extensión. Es decir, los containers son cerrados bajo composición, siendo la extensión de la composición equivalente a la composición de las extensiones.

Definición 3.10. Definimos la composición de los containers C y D como el container que tiene por conjunto de formas posibles a ⟦ C ⟧ (Sh D). Es decir que por cada posición en una forma c de C, tenemos una forma de D.

Dada una forma (c, d), siendo c forma de C y d una función de posiciones en C hacia formas en D, el conjunto de posiciones es el conjunto de pares dependientes (p, q), donde p es una posición en C en la forma c y q una posición de D en la forma $(d\,p)$.

```
Compose : Cont → Cont → Cont
Compose C D = ⟦ C ⟧ (Sh D) ◃ (λ { (c , d) →
                                    Σ[ p ∈ Pos C c ] Pos D (d p)})
```

Componer constructores de tipos da como resultado constructores en cierta forma *anidados*. Con la composición podemos formar, por ejemplo, las listas de árboles o los árboles de listas. En el siguiente código vemos la construcción del tipo maybeSthg C, que no es más que el constructor C equipado de un valor extra.

Ejemplo 3.11. *Composición del constructor de tipos Maybe con cualquier otro constructor C*

```
maybeSthg : Cont → Cont
maybeSthg C = Compose cMaybe C
```

En conclusión, hemos expuesto una sintaxis para representar ciertos constructores de tipo funtoriales F : Set → Set como un container cF : Cont. La función de extensión se constituye como un mecanismo para obtener nuevamente un funtor, tal que ⟦ cF ⟧ ≅ F.

3.2.3. Ejemplos

Ejemplo 3.12. Comencemos con uno de los ejemplos más sencillos que podamos construir: el container que define al tipo identidad. Recordemos la definición canónica de Id, expresada en Agda:

```
data Id (X : Set) : Set where
  identity : X → Id X
```

En este caso tenemos una única forma posible, con una única posición.

identity◯

El elemento que simbolizará al tipo Id en el universo de containers será el siguiente, siendo ⊤ el conjunto que tiene a tt como único el elemento.

```
cId : Cont
cId = ⊤ ◃ (λ _ → ⊤ )
```

Al extender dicho container, obtenemos un constructor de tipos Id. Podemos definir también el constructor de datos identity.

```
Id : Set → Set
Id = ⟦ cId ⟧

identity : ∀{X} → X → Id X
identity x = tt , (λ _ → x)
```

Ejemplo 3.13. Otro ejemplo trivial es el del container que construye el funtor constante. Es decir, aquél que ignora totalmente el tipo argumento y retorna un tipo constante fijado previamente.

```
cK : Set → Cont
cK A = A ◃ (λ _ → ⊥)
```

Las formas posibles serán exactamente los elementos del tipo constante, careciendo todas ellas de posiciones, puesto que pretendemos ignorar al tipo argumento. El funtor container asociado resulta:

```
K : Set → Set → Set
K A = ⟦ cK A ⟧
```

Por ejemplo, el tipo K ℕ ⊥ construye los naturales ignorando al tipo ⊥. Un habitante posible será:

```
dos : K ℕ ⊥
dos = 2 , (λ { () })
```

Ejemplo 3.14. El constructor de datos Maybe es aquél que agrega un elemento al tipo argumento. Un valor de tipo Maybe X contiene o bien un valor x de tipo X (representado por el elemento just x) o bien nada (representado por el valor nothing). La utilización de Maybe es una estrategia para lidiar con errores o casos excepcionales y resulta fundamental a la hora de escribir programas totales. Se define inductivamente como:

```
data Maybe (X : Set) : Set where
  nothing : Maybe X
  just    : X → Maybe X
```

Queremos construir una porción de sintaxis que represente a los esqueletos de Maybe, i.e., queremos construir un container. Las formas posibles que tendrán los datos son dos:

nothing   just○

La primera forma carece de posiciones y la segunda tiene una única posición. Definimos entonces el container cMaybe como:

```
cMaybe : Cont
cMaybe = Bool ◃ (λ { false → ⊥ ; true → ⊤ })
```

Al igual que hicimos con el tipo Id, calculamos su extensión y definimos los constructores de datos.

```
Maybe : Set → Set
Maybe = ⟦ cMaybe ⟧

nothing : ∀{X} → Maybe X
nothing = false , (λ { () })

just : ∀{X} → X → Maybe X
just x = true , (λ { tt → x })
```

Ejemplo 3.15. Un ejemplo muy interesante de analizar es el del tipo Stream. Un stream es una secuencia infinita numerable de valores. Una posible definición de este tipo de datos es:

```
data Stream (A : Set) : Set where
  _::_ :  A → Stream A → Stream A
```

Resulta importante evidenciar que es imposible construir un elemento de este tipo en Agda, dado que el chequeo de terminación nos lo impide. Por ejemplo, si quisiéramos construir una secuencia infinita trivial de unos:

```
s : Stream ℕ
s = 1 :: s
```

El recuadro naranja es la simbología que usa Agda para indicar que no puede construir una función debido a que no puede garantizar su terminación. Por lo tanto, dicho tipo de datos se define de forma coinductiva:

```
data Stream (A : Set) : Set where
  _::_ : (x : A) (xs : ∞ (Stream A)) → Stream A
```

Ahora sí podemos definir el stream de unos:

```
s : Stream ℕ
s = 1 :: ♯ s
```

Para pensar en una representación con containers de los streams, tenemos que dar, por un lado, un conjunto de formas. A diferencia de las listas, que podían ser de longitudes arbitrariamente grandes pero siempre finitas, los streams tienen todos una única longitud, y por lo tanto una única forma. Las posiciones en esa única forma son tantas como naturales haya. Se define entonces el container que representa al constructor de streams como:

```
cStream : Cont
cStream = ⊤ ◃ (λ {tt → ℕ})
```

El container funtor Stream y la secuencia infinita de unos que llamamos s, están dados por:

```
Stream : Set → Set
Stream = ⟦ cStream ⟧

s : Stream ℕ
s = tt , (λ _ → 1)
```

Advertir una ventaja que nos proveen los containers: no necesitamos apelar a toda la construcción coinductiva, de la misma forma que evitamos el uso de la inducción en los ejemplos previos.

Ejemplo 3.16. Para representar computaciones que dependen de valores en un entorno compartido e inmutable, la programación funcional modela la dependencia como una función. El tipo de datos Reader E encapsula este tipo de valores. Notar que no contamos con valores de tipo X. No hasta que se provea un entorno E.

```
data Reader (E : Set) (X : Set) : Set where
  reader : (E → X) → Reader E X
```

Asumiendo que ya contamos con un elemento E, las formas posibles para los datos se limita a una sola. Las posiciones sí son más. ¿Cuántas?. Pues si estamos modelando funciones de $E{\to}X$, necesitamos guardar tantos X como elementos haya en E.

```
cReader : Set → Cont
cReader E = ⊤ ◃ (λ {tt → E})
```

Finalmente, podemos construir toda la infraestructura auxiliar, una vez que hayamos vuelto al «mundo real».

```
Reader : Set → Set → Set
Reader E = ⟦ cReader E ⟧

reader : ∀{E X} → (E → X) → Reader E X
reader x = tt , x
```

Observemos que el container cReader resulta ser una generalización del container cStream presentado en el ejemplo 3.15, siendo este último equivalente a cReader ℕ.

Ejemplo 3.17. Veamos cómo el constructor de los árboles binarios con la información en las hojas puede ser representado como container. El tipo de datos dado inductivamente es:

```
data Tree (X : Set) : Set where
  leaf : X → Tree X
  node : Tree X → Tree X → Tree X
```

Un container de árbol será simplemente un árbol donde borramos la información que se almacena. En la figura 3.3 se pueden apreciar algunas esqueletos posibles de árboles.

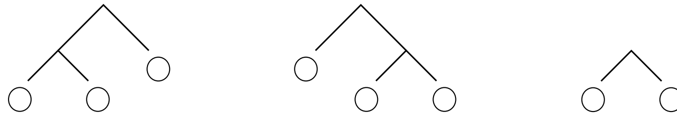


Figura 3.3: Containers de árbol

La definición del conjunto de formas no difiere en mucho de la definición inductiva del tipo Tree.[1]

```
data TreeSh : Set where
  leafSh : TreeSh
  nodeSh : TreeSh → TreeSh → TreeSh
```

Las posiciones posibles dependerán de las formas y serán tantas como hojas haya. Por cada hoja hay una posición y para una forma nodeSh l r, hay posiciones a la izquierda y a la derecha.

```
data TreePos : TreeSh → Set where
  leafPos : TreePos leafSh
  nodePosL : ∀{l r} → TreePos l → TreePos (nodeSh l r)
  nodePosR : ∀{l r} → TreePos r → TreePos (nodeSh l r)
```

[1] Notar que lo mismo sucede con las listas. El conjunto de formas resulta ser el de los naturales, cuya definición tiene la misma estructura que la de las listas, sólo que ignorando los datos que se almacenan

Finalmente, el container de los árboles binarios con datos en las hojas queda dado por:

```
cTree : Cont
cTree = TreeSh ◁ TreePos
```

Observar que si quisiéramos representar árboles con la información almacenada en los nodos, sólo cambiaríamos el conjunto de las posiciones, que dependerá de cuántos nodos tenga cada forma.

Otra cuestión interesante de destacar es el nivel de complejidad de notación que se alcanza al momento de representar árboles. En el capítulo 4 veremos que contamos con ciertos mecanismos de construcción de containers que proveen una suerte de modularidad, pudiendo componer containers de diversas maneras. Estudiaremos allí un álgebra de containers.

3.3. Morfismos de containers

En esta sección presentaremos una forma de escribir funciones polimórficas entre tipos funtoriales. La estrategia a seguir es similar a la tomada en la sección anterior. Proveeremos una forma de definir conjuntos de códigos sintácticos, que habremos de llamar *morfismos de containers* y una función de extensión que los reinterprete.

Sean los containers cF, cG : Cont que representan a los constructores de tipos F, G : Set → Set respectivamente. Es decir, ⟦ cF ⟧ ≅ F y ⟦ cG ⟧ ≅ G.

El objetivo es presentar una forma de representar ciertas funciones f : ∀{X}→ F X → G X como un morfismo de containers cf : cF ⇒ cG. En particular, nos interesan las f que preserven la estructura funtorial, es decir, que resulten ser *transformaciones naturales*.

Definición 3.18. Sean los funtores F, G : Set → Set. Una función f : ∀{X}→ F X → G X es una *transformación natural* si cumple, para toda $h : X \to Y$ y para alguna noción de igualdad.

map $h \circ f$ = f ∘ map h

Notar que de cada lado de la igualdad, f y map se instancian de forma diferente. Es decir, si agregamos los elementos implícitos como subíndices la ecuación de arriba nos queda:

$\mathsf{map}_G\ h \circ f_X = f_Y \circ \mathsf{map}_F\ h$

Para ello, definiremos un mecanismo de extensión que notaremos ⟦ _ ⟧$_m$ tal que ⟦ cf ⟧$_m$: ∀{X}→ ⟦ cF ⟧ X → ⟦ cG ⟧ X y así obtener ⟦ cf ⟧$_m$ ≅ f. La función ⟦ cf ⟧$_m$ obtenida de extender el morfismo cf será una transformación natural entre los funtores container ⟦ cF ⟧ y ⟦ cG ⟧.

Para comenzar la exposición analizaremos un ejemplo.

Ejemplo 3.19. Echemos un vistazo a la siguiente función, que retorna el primer elemento de una lista, si lo hay.

```
head : ∀{X} → List X → Maybe X
head nil = nothing
head (cons x l) = just x
```

Intuitivamente, un morfismo entre dos containers proveerá una forma de *reubicar* los elementos del primer container dentro del segundo. ¿Qué necesitamos proveer para crear una función chead que transforme un elemento de cList en uno de cMaybe de la forma esperada? Por una lado necesitamos una función de Sh cList en Sh cMaybe para poder indicar que la lista vacía se transforma en el elemento nothing y que para cualquier otra longitud de lista obtendremos la forma just ◯. Además necesitamos saber desde qué posiciones del container cList provienen los datos a guardar en el container imagen y así *reubicar* los valores almacenados. Para ello basta con dar una función que dada una posición de cMaybe nos indique una posición de cList, para cada forma. En el caso de head habrá que indicar que el dato a guardar en la única posición de la forma just ◯ proviene de la primera posición de una lista de al menos un elemento, como muestra la figura 3.4.

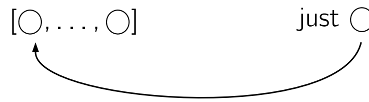


Figura 3.4: reubicando las posiciones según el comportamiento de head

3.3.1. Definición y extensión de morfismos de container

Definición 3.20. Un *morfismo de containers* entre los containers C_1 y C_2 es un par de funciones (m_{Sh}, m_{Pos}), donde

- m_{Sh} es una función que mapea las formas de C_1 hacia las formas de C_2
- Por cada forma s de C_1, m_{Pos} es una función que mapea cada posición de C_2 en la forma $m_{Sh}\ s$ hacia las posiciones de C_1 en la forma s.

Su formalización en Agda se expresa en el siguiente código:

Código 3.21. *Formalización de los morfismos de containers*

```
record _⇒_ (C₁ C₂ : Cont) : Set where
  constructor _,_
  field
    mSh  : Sh C₁ → Sh C₂
    mPos : ∀{s : Sh C₁} → Pos C₂ (mSh s) → Pos C₁ s
```

La componente mSh del morfismo indica cómo transformar cada elemento del primer conjunto de formas en uno del segundo. La función mPos es la encargada de reubicar los elementos a almacenar, indicando para cada posición del container imagen, una posición en el container origen.

Definición 3.22. Sea f un morfismo entre los containers C y D. La *extensión* de f sobre un conjunto X, que notamos $[\![\ f\ ]\!]_m$, es una función entre los funtores containers $[\![\ C\ ]\!]$ y $[\![\ D\ ]\!]$ dada por el siguiente código:

Código 3.23. *Extensión de morfismos de containers*

```
⟦_⟧m : ∀{C D} → (C ⇒ D) → ∀{X} → ⟦ C ⟧ X → ⟦ D ⟧ X
⟦ f ⟧m ( c , fp ) = (mSh f c) , (fp ∘Set mPos f {c})
```

Al aplicar $[\![f]\!]_m$ al elemento dado por una forma c_s y una función c_p de posiciones en valores, obtenemos otro container. Su forma la obtenemos gracias a la aplicación de la función de formas mSh f. Dada una posición en esta nueva forma, mPos f indicará de dónde provienen los datos a guardar allí, información que completará la función c_p .

Un resultado muy interesante de esta construcción es que los morfismos de containers representan exactamente al conjunto de las transformaciones naturales entre los funtores $[\![C]\!]$ y $[\![D]\!]$. Utilizando el universo de containers y morfismos nos aseguramos que las funciones definidas entre dos funtores containers serán transformaciones naturales, siendo imposible definir algo que no lo sea. Más aún, toda transformación natural t entre $[\![C]\!]$ y $[\![D]\!]$ es representable en el universo de morfismos de containers por un elemento f, de forma tal que $[\![\mathrm{f}]\!]_m = t$.

Ejemplo 3.24. Para completar el ejemplo 3.19 expuesto en la introducción, veamos finalmente cómo queda definido el morfismo chead:

```
chead : cList ⇒ cMaybe
chead = (λ { 0 → false; (suc _) → true }),(λ { {0} (); {suc _} tt → zero})
```

La función entre formas mapea la longitud de lista cero al elemento false, representante del constructor nothing. Para cualquier otra longitud de lista la forma resultante es true, que indicaba la forma just.

La función entre posiciones es vacía para el caso de las listas vacías puesto que allí no existen posiciones. En cualquier otra situación se mapea la posición única tt a la primera de la lista.

```
head : ∀{X} → List X → Maybe X
head = ⟦ chead ⟧m
```

Definición 3.25. Se define la *composición* entre dos morfismos de containers componiendo respectivamente las funciones de formas y posiciones, según se muestra en el siguiente código:

```
_∘_ : ∀{A B C} → (B ⇒ C) → (A ⇒ B) → (A ⇒ C)
g ∘ f = (mSh g ∘Set mSh f) , (mPos f ∘Set mPos g)
```

Ejemplo 3.26. *Morfismos de containers identidad*

Dado cualquier container C, existe el morfismo identidad, que deja fija las formas y las posiciones.

```
id : ∀{C} → C ⇒ C
id = idSet , idSet
```

Notar que la extensión de dicho morfismo será la función identidad.

3.3.2. Equivalencia de morfismos

Un lema que nos resultará muy útil para demostrar equivalencias en esta sección es aquél que prueba la igualdad de morfismos de containers. Dos morfismos de containers f y g son equivalentes siempre que lo sean sus componentes, las funciones de formas y posiciones.

Código 3.27. *Equivalencia entre morfismos de containers*

```
mEq : ∀{A B}{f g : A ⇒ B}
   → mSh f ≅ mSh g
   → (λ {s} → mPos f {s}) ≅ (λ {s} → mPos g {s})
   →  f ≅ g
mEq refl refl = refl
```

Este es un claro ejemplo donde Agda no puede resolver trivialmente una equivalencia por no poder unificar los tipos de los argumentos. Agda sólo nos permite probar por refl una vez hecho pattern matching en los argumentos de mEq. Esto es así puesto que sólo una vez probada la equivalencia de las funciones de formas, las funciones de posiciones tienen exactamente el mismo tipo.

Entonces, yendo paso a paso, el único habitante de mSh f ≅ mSh g es refl. Indicado esto, Agda unifica esta equivalencia y es posible hacer pattern matching en el segundo argumento. Al unificar la prueba de mPos f ≅ mPos g la prueba final resulta trivial.

3.4. Categoría de containers

Dedicamos esta sección a presentar la formalización de los containers como una categoría. Los objetos de la categoría, que habremos de llamar **Cont** resultan ser los containers, definidos en 3.3. El conjunto de morfismos será el de los morfismos de containers, expuesto en 3.21. Como hemos mostrado en 3.25 y 3.26, los morfismos pueden componerse y siempre existe la identidad.

Código 3.28. *Categoría de containers*

```
Cont : Category
Cont = record
   { Obj   = Cont
   ; Hom   = _⇒_
   ; iden  = id
   ; comp  = _○_
   ; idl   = refl
   ; idr   = refl
   ; ass   = refl }
```

Capítulo 4

Construcciones con containers

> Se conservaron algunos adjetivos de hoy en día como bueno, fuerte, grande, negro, blando, pero en un número muy reducido. Por otra parte, su necesidad era mínima, ya que se llegaba a cualquier significado adjetival añadiendo *lleno* a un sustantivo-verbo. (...) Además, a cualquier palabra —y esto, como principio, se aplicaba a todas las palabras del idioma—, se le daba sentido de negación añadiendo el prefijo *in* o se le daba fuerza con el sufijo *plus*, o para aumentar el énfasis, *dobleplus*.
>
> 1984
> George Orwell

En este capítulo presentaremos una serie de construcciones que no sólo aportarán a la modularidad de uso del universo de containers, apuntando a la programación genérica, sino que también clasificarán a la categoría de containers dentro de ciertas colecciones de categorías con propiedades interesantes, las categorías cartesianas cerradas.

Hemos visto en el capítulo 3 que los containers resultan ser una codificación válida de ciertos constructores de tipos. Desde este punto de vista propondremos formas genéricas de construir, por ejemplo, a partir de un par de constructores, el constructor de pares. Pensando lo antedicho en términos algebraicos, podemos intuir que hablamos de un operador producto sobre containers y su clausura. Es decir, que los containers son cerrados bajo alguna definición de producto. Si bien en términos particulares eso es cierto, nos interesa asimismo tomar un punto de vista aún más general, retomando las nociones de teoría de categorías expuestas en las preliminares y continuadas en el capítulo 3. De esta forma, veremos lo significa que una categoría *cuente con productos*. Hecho esto, mostraremos que la categoría **Cont** de containers y sus morfismos, efectivamente cuenta con ellos. Además del producto, presentaremos las construcciones de coproducto, exponencial, objetos inicial y terminal y veremos brevemente porqué se las denomina *construcciones universales*.

Debido a que no asumimos conocimientos previos de teoría de categorías, para cada construcción abstracta comenzaremos introduciendo brevemente su dimensión categórica. Para ello, se impulsará al lector a pensar desde lo particular –la teoría de conjuntos– hacia lo general –la teoría de categorías– y así motivar la introducción de cada nuevo concepto categórico a partir de la abstracción de un caso particular, siguiendo un camino similar al transitado por algunos libros de referencia [Awo10, BW95]. Con esta base expondremos una formalización genérica en código Agda, extendiendo lo presentado en la sección 2. Finalmente, exhibiremos la forma que tienen dichas construcciones sobre containers. Cuando sea posible daremos también una interpretación menos abstracta, en teoría de la programación, siguiendo la línea de presentación del capítulo 3, con ejemplos y análisis particulares al caso.

4.1. Coproducto

4.1.1. Motivación: (De)construyendo el coproducto

Muchas de las construcciones de teoría de categorías pueden pensarse a partir de la teoría de conjuntos, con vistas a la generalización. En efecto, muchas de estas nociones categóricas han surgido de esta manera. Nuestra meta en esta sección de motivación es tratar de seguir ese proceso que va desde lo particular a lo general para así comprender más profundamente la construcción de *coproducto* que nos concierne analizar en esta parte del trabajo.

La unión disjunta de conjuntos, que notamos $\uplus$, es la modificación de la unión clásica de conjuntos donde se *etiquetan* los elementos según el conjunto desde el que provienen. Para mayor claridad, observemos el siguiente ejemplo, donde las *etiquetas* se muestran como subíndices:

$$\begin{array}{cccc} A & B & A \cup B & A \uplus B \\ \{x, y\} & \{x, z, w\} & \{x, y, z, w\} & \{x_A, y_A, x_B, z_B, w_B\} \end{array}$$

Notar que con esta modificación la operación deja de ser idempotente: unir disjuntamente un conjunto consigo mismo traerá como resultado la duplicación de sus elementos.

Un habitante de la unión disjunta de dos conjuntos será o bien un elemento del primer conjunto o bien uno del segundo. En el caso de los conjuntos, el coproducto nos permite codificar una opción, una disyunción. En programación suele utilizarse esta construcción para expresar que un elemento es *o bien* de tipo A *o bien* de tipo B. En Agda suele definirse como:

Código 4.1. *Unión disjunta de conjuntos*

```
data _⊎_ (A B : Set) : Set where
  inj₁ : A → A ⊎ B
  inj₂ : B → A ⊎ B
```

En resumidas cuentas, un elemento del conjunto $A \uplus B$ es *o bien* un valor de la forma $\mathrm{inj}_1\ a$ (con $a : A$) *o bien* un valor de la forma $\mathrm{inj}_2\ b$ (con $b : B$). A las funciones de etiquetado suele designárseles los nombres de *inyecciones*, el nombre inj proviene del vocablo inglés *injection*. Se las denomina de esa manera porque se considera que inyectan en el objeto coproducto la información de los factores.

Volviendo a teoría de categorías, hemos mencionado en la sección 2 que dicha teoría hace hincapié en las relaciones entre los objetos más que en los objetos en sí mismos. Dicho de otra forma, son los morfismos los que capturan la esencia de cada idea. Trataremos entonces de expresar cada concepto que nos interesa de la teoría de conjuntos centrándonos en las funciones que acompañan a cada construcción, es decir, en las relaciones entre los conjuntos involucrados. Sin embargo, la definición recientemente expuesta hace justamente lo que queremos evitar: referirse a los habitantes de los conjuntos. Para comenzar a independizarnos de ellos podemos preguntarnos cómo definir un objeto coproducto a partir de las relaciones que podemos percibir entre los objetos.

En términos particulares, dados dos conjuntos A y B, ¿qué relación tienen con su unión disjunta? ¿qué función involucra a A y a B con $A \uplus B$? Por empezar, siempre podremos construir funciones que vayan desde cada conjunto hacia su unión disjunta, las

funciones de inyección. Con esto en mente, podríamos pensar entonces al coproducto como un objeto equipado con dos morfismos desde cada objeto factor hacia él.

Pero no cualquier objeto de estas características es el objeto coproducto, ya que claramente no cualquier conjunto C y par de funciones que vayan de A y B hacia C resulta ser la unión disjunta. Entonces, para indicar porqué otro objeto *no* es el coproducto, habrá que involucrar también a las relaciones entre el verdadero objeto coproducto que estamos definiendo y *cualquier otro* objeto de la categoría que se postule como tal. Siendo C un conjunto cualquiera y $f : A \to C,\ g : B \to C$ funciones, ¿qué hace que C no sea un buen candidato a ser el objeto coproducto? ¿qué relación encontramos entre C y $A \uplus B$? Pues desde la unión disjunta *siempre* podremos construir una función hacia este objeto C simplemente eligiendo qué función aplicar, si f o g. Aún más, dicha función es la única entre $A \uplus B$ y C con estas características. La definimos como:

Código 4.2.

```
[_,_] : ∀{A B C : Set} → (A → C) → (B → C) → (A ⊎ B → C)
[ f , g ] (inj₁ x) = f x
[ f , g ] (inj₂ y) = g y
```

Luego de haber desmenuzado a la unión disjunta de conjuntos en aras de la generalidad categórica, continuaremos la exposición con la definición formal de coproductos en una categoría.

4.1.2. Definición y formalización

Definición 4.3. Sean A, B objetos de una categoría $\mathscr{C}$. El *coproducto* de A y B es un objeto $A + B$, junto con dos morfismos $A \xrightarrow{\iota_1} A + B$ y $B \xrightarrow{\iota_2} A + B$ tal que para todo otro objeto C y par de morfismos $A \xrightarrow{f} C,\ B \xrightarrow{g} C$ de la categoría, exista un único morfismo $[f, g]$ que haga conmutar al siguiente diagrama :

$$\begin{array}{ccccc} & & C & & \\ & \overset{f}{\nearrow} & \uparrow{\scriptstyle [f,g]} & \overset{g}{\nwarrow} & \\ A & \xrightarrow[\iota_1]{} & A+B & \xleftarrow[\iota_2]{} & B \end{array}$$

Los morfismos como $[f, g]$ suelen denominarse *mediales* por ser los que atraviesan el diagrama conmutativo. En el caso particular del coproducto, lo llamaremos morfismo de *coreunión*. La representación del morfismo como una flecha con línea de puntos en el diagrama conmutativo es un recurso para expresar unicidad. Es decir, para que el objeto $A + B$ cumpla con la definición, debe existir un único morfismo $A + B \to C$ que haga conmutar al diagrama.

Cuando para cada par de objetos de una categoría existe su objeto coproducto, decimos que la categoría *cuenta con coproductos*.

Habiendo definido al objeto coproducto podemos mencionar de qué se tratan las *construcciones universales*, al menos sin referirnos a los pormenores del asunto. En general, cuando hablamos del “mejor objeto” que cumpla con ciertas propiedades, como lo hicimos con el coproducto, nos estamos refiriendo a una construcción universal. Muchas veces con “mejor” nos referimos a menor, mayor, con menos restricciones, etc.

Las propiedades universales son ubicuas en la matemática. La teoría de categorías apunta a analizarlas desde un punto de vista abstracto y así comprenderlas en su totalidad, evitando repetir el estudio de cada instancia en particular. Por ejemplo, sabemos que las propiedades universales definen objetos que resultan ser únicos, salvo isomorfismos únicos. Particularmente, puede demostrarse que todo objeto coproducto de otros dos es único hasta isomorfismos. Esto significa que es posible que exista más de un objeto coproducto dados otros dos, pero todos ellos serán isomorfos.

Como nota de color resulta interesante mencionar que otra denominación para los coproductos que suele encontrarse en la bibliografía es la de *suma*, siendo un nombre más intuitivo, sobre todo cuando pensamos en ciertas categorías. El apelativo *coproducto* se origina en el hábito de denominar co-*concepto* al dual de un *concepto*. Recordemos que la categoría dual $\mathscr{C}^{op}$ de una categoría $\mathscr{C}$ es aquella que se obtiene al revertir el sentido de los morfismos. De este modo, un co-*concepto* en una categoría $\mathscr{C}$ es un *concepto* en $\mathscr{C}^{op}$. Veremos en la sección 4.2 que la idea de *producto* puede pensarse en los mismos términos que el coproducto, pero revirtiendo el sentido de las flechas del diagrama conmutativo. Dicho esto cabe preguntarse: ¿porqué entonces no se llama *cosuma* al producto? Pues porque en teoría de categorías, el producto fue concebido primero. En 1950, Saunders Mac Lane [Mac50] presenta lo que sería muy probablemente el ejemplo más remoto de uso de la teoría de categorías para definir una noción matemática fundamental[1].

A continuación formalizaremos en Agda lo presentado hasta aquí. Seguiremos los pasos de la sección 2 y extenderemos dicho modelo para permitir expresar cuándo una categoría tiene coproductos.

Como expusimos anteriormente, Agda nos permite definir estructuras algebraicas que incluyan no sólo al portador con sus operaciones sino también las propiedades que se deben cumplir. El código 4.4 formaliza lo presentado, definiendo una estructura de record indexado por una categoría $\mathscr{C}$. Dar una instancia de HasCoproducts $\mathscr{C}$ implica probar que la categoría $\mathscr{C}$ cuenta con coproductos.

Código 4.4. *Formalización de categoría con coproductos*

```
record HasCoproducts (𝒞 : Category) : Set where
  open Category
  field
    Coprod       :  Obj 𝒞 → Obj 𝒞 → Obj 𝒞
    inl          :  ∀{X Y} → Hom 𝒞 X (Coprod X Y)
    inr          :  ∀{X Y} → Hom 𝒞 Y (Coprod X Y)
    copair       :  ∀{X Y Z}(f : Hom 𝒞 X Z)(g : Hom 𝒞 Y Z)
                 →  Hom 𝒞 (Coprod X Y) Z
    copairIdl    :  ∀{X Y Z}(f : Hom 𝒞 X Z)(g : Hom 𝒞 Y Z)
                 →  comp 𝒞 (copair f g) inl ≅ f
    copairIdr    :  ∀{X Y Z}(f : Hom 𝒞 X Z)(g : Hom 𝒞 Y Z)
                 →  comp 𝒞 (copair f g) inr ≅ g
    copairUnique :  ∀{X Y Z}{f : Hom 𝒞 X Z}{g : Hom 𝒞 Y Z}
                    {h : Hom 𝒞 (Coprod X Y) Z}
                 →  (p₁ : comp 𝒞 h inl ≅ f)
                 →  (p₂ : comp 𝒞 h inr ≅ g)
```

[1] Ver [Awo10]

$\rightarrow \; h \cong$ copair $f\, g$

El elemento Coprod es el que construye cada objeto coproducto, dados otros dos. Los elementos inl e inr son morfismos en la categoría $\mathscr{C}$, definidos para todo par de objetos X, Y como las respectivas inyecciones. De forma similar se expresa la signatura del morfismo medial copair. Los campos dedicados a las pruebas garantizarán que las construcciones proporcionadas son las correctas, dado que hacen conmutar al diagrama de la definición 4.3.

Veamos a continuación cómo se manifiesta esta construcción en la categoría de containers.

4.1.3. Coproducto de Containers

Para formalizar el coproducto en la categoría de containers necesitamos contar con un elemento Coprod que arme un container a partir de otros dos. Será obligatorio definir también funciones inl, inr y copair que proyecten cada uno de los containers involucrados en un coproducto. Finalmente, habrá que suministrar las pruebas requeridas y así garantizar que estamos presentando la construcción correcta. En resumen, buscamos construir un elemento ContHasCoproducts, llenando los huecos del código 4.5 indicados con signos de interrogación y así formalizar la noción de que la categoría **Cont** presentada en la sección 3.4 tiene coproductos.

Código 4.5.

```
ContHasCoproducts : HasCoproducts Cont
ContHasCoproducts = record
  { Coprod       = ?
  ; inl          = ?
  ; inr          = ?
  ; copair       = ?
  ; copairIdl    = ?
  ; copairIdr    = ?
  ; copairUnique = ? }
```

A continuación se muestra el código del constructor Either que arma un coproducto a partir de dos containers C y D. Por un lado, las formas de un container coproducto serán o bien formas de C o bien formas de D. Por otro lado, hecha esa elección y determinada una forma s, las posiciones posibles dentro de ella dependerán del conjunto del que s proviene. Si s es una forma de C, entonces las posiciones posibles serán las posiciones de C en s y de forma similar en el caso en que s sea forma de D.

```
Either : Cont → Cont → Cont
Either C D = (Sh C ⊎ Sh D) ◁ [ Pos C , Pos D ]Set
```

Renombramos como $[_ , _]_{\mathsf{Set}}$ al operador $[_ , _]$ presentado en el código 4.2 para dejar en claro que se trata del correspondiente a la categoría **Set** y no al que presentaremos próximamente, perteneciente a la categoría **Cont**.

Lógicamente, las inyecciones serán morfismos en la categoría de containers. La primera componente del morfismo es la función sobre formas, que simplemente inyecta una

forma de C en el conjunto Sh $C \uplus$ Sh D. Dada una forma c de C, las posiciones en Either C D se reducen a posiciones de C en c. La función de formas será simplemente la función identidad.

```
ι₁ : ∀{C D} → C ⇒ Either C D
ι₁ = inj₁ , id_Set
```

La segunda inyección es análoga.

El morfismo de coreunión construye un morfismo a partir de otros dos, eligiendo aplicar uno u otro dependiendo del caso. La función de formas elegirá cuál de las respectivas funciones de formas aplicar. La nueva función de posiciones hará lo mismo con las respectivas funciones de posiciones originales.

```
[_,_] :  ∀ {A B C} → (A ⇒ C) → (B ⇒ C) → (Either A B ⇒ C)
[ f , g ] = [ mSh f , mSh g ]_Set  ,  (λ {  {inj₁ x}  →  mPos f {x}
                                     ;      {inj₂ y}  →  mPos g {y} })
```

Para terminar, se requiere dar pruebas de haber construido el coproducto correctamente. Probaremos por un lado que inyectar elementos al lado izquierdo de un coproducto para luego aplicar la coreunión es igual a sólo hacer lo que decide la coreunión sobre la parte izquierda. Igualmente sobre la parte derecha. Es decir, queremos construir elementos de $[\ f, g\ ] \circ \iota_1 \cong f$ y de $[\ f, g\ ] \circ \iota_2 \cong g$ dados los morfismos f y g. A continuación se expone la definición para el primer caso, la prueba para la segunda proyección es análoga.

```
[_,_]∘ι₁≅f  : ∀ {A B C}
            → (f : A ⇒ C)
            → (g : B ⇒ C)
            → [ f , g ] ∘ ι₁ ≅ f
[ f , g ]∘ι₁≅f  = refl
```

Recordemos que el único habitante expreso de la equivalencia proposicional es refl. En el caso de esta primera prueba, Agda computa cada lado de la equivalencia y obtiene el mismo resultado, pudiendo entonces probar la regla trivialmente.

Pero hay casos donde de cada lado de la igualdad hay elementos de distinto tipo. Habrá que proveer primero una prueba de la equivalencia de esos tipos para poder continuar. Este es un caso muy común cuando tratamos con tipos indexados, como lo son el tipo de los containers y de los morfismos de containers. Para sortear este obstáculo se definen lemas que ayuden con la prueba, como ser el lema de equivalencia de morfismos expuesto en el código 3.27. Otro recurso al que habrá que apelar es al axioma de extensionalidad definido en 1.26, en la situación en que queramos probar la equivalencia de funciones.

Apelando a los recursos recientemente mencionados, podremos demostrar que la coreunión es única. Homólogamente, probaremos que cualquier otro morfismo h entre los elementos indicados que haga conmutar al diagrama de la definición 4.3 es equivalente a él. Es decir, dadas pruebas de que h *se comporta* como una función de coreunión, veremos que en efecto lo es. Construiremos un elemento de $h \cong [\ f\ ,\ g\ ]$ a partir de elementos de $h \circ \iota_1 \cong f$ y de $h \circ \iota_2 \cong g$.

```
copairUnique_Cont  : {A B C : Cont}
                   → {f : A ⇒ C}
                   → {g : B ⇒ C}
                   → {h : Either A B ⇒ C}
```

```
                    → h ∘ ι₁  ≅ f
                    → h ∘ ι₂  ≅ g
                    → h ≅ [ f , g ]
copairUnique_Cont refl refl =
  mEq  (ext   ( λ  {  (inj₁ _) → refl
                   ;  (inj₂ _) → refl }))
       (iext  ( λ  {  {inj₁ _} → ext (λ _ → refl)
                   ;  {inj₂ _} → refl }))
```

Finalmente, habiendo presentado cada construcción y probado cada lema, podemos completar los huecos del código 4.5, simbolizados con signos interrogatorios:

Código 4.6. *Formalización* **Cont** *como categoría con coproductos*

```
ContHasCoproducts : HasCoproducts Cont
ContHasCoproducts = record
  { Coprod        = Either
  ; inl           = ι₁
  ; inr           = ι₂
  ; copair        = [_,_]
  ; copairIdl     = λ f g → [ f , g ]∘ι₁≅f
  ; copairIdr     = λ f g → [ f , g ]∘ι₂≅g
  ; copairUnique  = copairUnique_Cont }
```

Ejemplo con coproductos de containers

Como ya hemos mencionado, el tipo de datos de coproducto se utiliza en programación para expresar que un elemento puede ser, o bien de un tipo, o bien de otro. En particular, suele utilizarse en los casos donde una función retorna o bien un valor, el valor esperado, o bien un mensaje de error en el caso en que un error haya ocurrido. Desde esta perspectiva, el constructor Either resulta una buena generalización de Maybe, donde en lugar de retornar un valor único nothing en el caso de error, se retorna una descripción más útil de lo ocurrido.

En el código a continuación vemos un ejemplo sencillo de cómo con containers podemos crear un mecanismo para equipar a un elemento de tipo Maybe C en un valor de la forma Either C Str.

```
errorMes :  ∀{C : Cont}{Str : Set}
            → Str
            → Compose cMaybe C ⇒ Either C (cK Str)
errorMes s =  (λ  {  (false  , _)      → inj₂ s
                  ;  (true   , c)      → inj₁ (c tt) }),
              (λ  {  {false  , _} ()
                  ;  {true   , _} x → tt , x })
```

La función errorMes toma como argumentos un container C, un tipo Str y un valor de dicho tipo que será el elemento que se retorne en caso de error. El tipo Str también será determinado por el usuario. A pesar de que el nombre elegido evoca al tipo *String* de cadenas de caracteres, podría ser cualquier tipo que se considere apto para representar el error. Con esa información se construye un morfismo de containers que tiene por origen al container Compose cMaybe C.

Recordemos que el container cMaybe tenía por conjunto de formas a los booleanos, siendo false el representante del elemento canónico nothing y el valor true el representante de constructor just. De esta forma, el morfismo transforma el elemento false en el nuevo valor s, dejándolo del lado derecho de la unión disjunta con el constructor inj_2; del elemento true se extrae la información de tipo C almacenada y se la ubica del lado izquierdo de la unión disjunta.

4.2. Producto

4.2.1. Motivación: (De)construyendo el producto

Para continuar con la línea seguida en la sección 4.1 de coproductos, trataremos de definir la construcción de producto en una categoría con la intención de abstraernos del caso particular de la ya conocida categoría de conjuntos.

Recordemos que el producto cartesiano de dos conjuntos X, Y se define como el conjunto de los pares (x, y) donde $x \in X$, $y \in Y$. Es posible definir dos funciones que dado un par, proyecten cada uno de los elementos. El producto en Agda está dado por un record equipado con dos funciones (ver código 1.12, pág 14)

Es nuestro objetivo pensar en las relaciones en lugar de pensar en los elementos, para así abstraernos y concluir en la definición categórica, determinando restricciones sobre los morfismos. Con esto en mente, podríamos decir que un objeto producto es uno para el cual existan morfismos de proyección que vayan del objeto en cuestión hacia cada uno de los factores.

De la misma forma que concluimos en la sección de coproductos, no cualquier objeto que cuente con dicho par de morfismos resulta ser el objeto producto. Esto resulta aún más evidente cuando pensamos en los conjuntos, ya que no cualquier conjunto que cuente con un par de funciones hacia los factores X e Y resulta ser el producto cartesiano $X \times Y$. Es necesario entonces establecer relaciones entre el objeto producto y cualquier otro objeto de la categoría. Si pensamos en cualquier otro conjunto C que se postule como producto cartesiano, con un par de funciones f, g hacia X e Y respectivamente, ¿qué relación se puede establecer con $X \times Y$? Notemos que una característica del producto cartesiano es que mantiene la información de los objetos factores *a la vez*, siendo posible volver a cada uno de ellos a través de las proyecciones. Pero otra cualidad del producto cartesiano es que no contiene más ni menos información que esa. En este sentido, el objeto producto es el mínimo objeto que contiene la información de los factores.
Cualquier otro conjunto C tendrá o más o menos información de la necesaria, pero siempre existirá una única función hacia $X \times Y$ simplemente aplicando f y g y reuniendo el resultado en un par, como se define en el siguiente código:

Código 4.7. *Función de reunión de conjuntos*

```
<_,_> : ∀{X Y C : Set} → (C → X) → (C → Y) → (C → X × Y)
< f , g > = λ x → ( f x , g x )
```

Cabe aclarar que a pesar que en el caso de la categoría **Set** estos elementos existan siempre, esto no tiene que impulsarnos a pensar que será así en cualquier categoría. Definiremos a continuación cuándo una categoría cuenta con productos.

4.2.2. Definición y formalización

Definición 4.8. Sean A, B objetos de una categoría $\mathscr{C}$. Decimos que el objeto $A \times B$ es el *producto* de A y B si existen los morfismos $A \times B \xrightarrow{\pi_1} A$ y $A \times B \xrightarrow{\pi_2} B$ tal que para todo otro objeto C y par de morfismos $C \xrightarrow{f} A$, $C \xrightarrow{g} B$ de la categoría, exista un único morfismo $\langle f, g \rangle$ que haga conmutar al siguiente diagrama:

$$\begin{array}{ccccc} & & C & & \\ & \swarrow^{f} & \downarrow{\scriptstyle \langle f,g \rangle} & \searrow^{g} & \\ A & \xleftarrow[\pi_1]{} & A \times B & \xrightarrow[\pi_2]{} & B \end{array}$$

En otras palabras, se requiere que el siguiente conjunto de ecuaciones se cumpla, para todas f y g:

$$\begin{cases} \pi_1 \circ \langle f, g \rangle & \cong \quad f \\ \pi_2 \circ \langle f, g \rangle & \cong \quad g \\ \langle f, g \rangle & \cong \quad h \qquad \text{para toda } h \text{ tal que} \quad \pi_1 \circ h \cong f \quad \text{y} \quad \pi_2 \circ h \cong g \end{cases}$$

De la misma forma que con los coproductos, cuando para todo par de objetos de una categoría existe su producto, decimos que la misma *cuenta con productos*.

Para completar la abstracción definiremos un morfismo auxiliar que mapea otros dos sobre un producto. Notar que este morfismo no forma parte de la estructura algebraica de los productos, sino que se define derivada de ella:

Definición 4.9. Sean A, B, C y D objetos de una categoría $\mathscr{C}$ con productos y sea el par de morfismos $A \xrightarrow{f} B$ y $C \xrightarrow{g} D$. Se define el morfismo producto de f y g como

$$f \times g = \langle f \cdot \pi_1, g \cdot \pi_2 \rangle$$

Notar que decimos que el morfismo $f \times g$ *mapea* f y g sobre $A \times C$ puesto que: $A \times C \xrightarrow{f \times g} B \times D$

Para formalizar esta construcción en Agda, diremos que una categoría tiene productos cuando sea posible proveer un elemento Prod que construya un objeto a partir de otros dos, junto con los morfismos de proyección que llamaremos projl y projr. El morfismo medial de reunión pair puede pensarse como un constructor de un morfismo a partir de otros dos.

A continuación se provee una función extra, debido a su amplio uso: la función pmap, ya expuesta en la definición 4.9. Su implementación se provee dentro del record, haciendo uso de los campos declarados.

Finalmente, el record incluye tres ecuaciones, pairIdl, pairIdr y pairUnique, cuyas pruebas también deberán proveerse a la hora de definirse instancias. Dichas ecuaciones se corresponden con las presentadas en la definición 4.8. Presentamos esta formalización en el código 4.10.

Código 4.10. *Formalización de categoría con productos*

```
record HasProducts (𝒞 : Category ) : Set where
  field
    Prod  : Obj 𝒞 → Obj 𝒞 → Obj 𝒞
    projl : ∀{X Y} → Hom 𝒞 (Prod X Y) X
    projr : ∀{X Y} → Hom 𝒞 (Prod X Y) Y
    pair  : ∀{X Y Z}(f : Hom 𝒞 Z X)(g : Hom 𝒞 Z Y)
          → Hom 𝒞 Z (Prod X Y)

  pmap  : ∀{X X' Y Y'}
        → (f : Hom 𝒞 X X')(g : Hom 𝒞 Y Y')
        → Hom 𝒞 (Prod X Y) (Prod X' Y')
  pmap f g = pair (comp 𝒞 f projl) (comp 𝒞 g projr)

  field
    pairIdl    : ∀{X Y Z}(f : Hom 𝒞 Z X)(g : Hom 𝒞 Z Y)
               → comp 𝒞 projl (pair f g) ≅ f
    pairIdr    : ∀{X Y Z}(f : Hom 𝒞 Z X)(g : Hom 𝒞 Z Y)
               → comp 𝒞 projr (pair f g) ≅ g
    pairUnique : ∀{X Y Z}{f : Hom 𝒞 X Y}{g : Hom 𝒞 X Z}
               → {h : Hom 𝒞 X (Prod Y Z)}
               → (p₁ : comp 𝒞 projl h ≅ f)
               → (p₂ : comp 𝒞 projr h ≅ g)
               →  h ≅ (pair f g)
```

Dada un categoría cualquiera $\mathscr{C}$, proveer un elemento de tipo HasProducts $\mathscr{C}$ garantizará que dicha categoría cuente con productos, asegurando por un lado una forma de construirlos y por otro las pruebas requeridas por la definición.

Como hemos visto en la sección de 4.2.1, la categoría **Set** cuenta con productos. A modo de ejemplo, vemos formalizada esta aseveración en el código 4.11. La prueba de la unicidad de la función de reunión no se expone para no ahondar en detalles que no aportan, pero resulta trivial una vez unificadas las pruebas que toma como supuesto.

Código 4.11.

```
SetHasProducts : HasProducts Set
SetHasProducts = record
  { Prod       = _×_
  ; projl      = proj₁
  ; projr      = proj₂
  ; pair       = <_,_>
  ; pairIdl    = λ f g → refl
  ; pairIdr    = λ f g → refl
  ; pairUnique = pairUnique_Set
  }
```

4.2.3. Producto de Containers

De la misma forma que hemos procedido con los coproductos de containers, tenemos el objetivo de proveer un habitante del record HasProducts **Cont**.

Retomando el análisis general de la construcción de productos en una categoría, hemos observado que un objeto producto incluye las informaciones de los objetos factores al mismo tiempo, siendo posible proyectar cada una de ellas.

Dados dos containers, ¿qué container podemos construir para poder guardar los datos que guardan ambos, a la vez? Denominaremos Both C D al container producto de C y D. Both C D tendrá por conjunto de formas a los pares (c, d), siendo c forma de C y d, de D. Es decir, el conjunto de formas de Both C D no es más ni menos que el producto cartesiano entre Sh C y Sh D. Las posiciones posibles dentro de una forma (c, d) serán, o bien posiciones en c, o bien posiciones en d. Es decir, las posiciones serán el resultado de la unión disjunta (definida en 4.1, pág 22) de las respectivas posiciones originales.

Queda entonces definido el constructor de productos Both como:

```
Both : Cont → Cont → Cont
Both C D = (Sh C × Sh D) ◁ (λ { (c , d) → Pos C c ⊎ Pos D d })
```

Para proyectar la información contenida en un producto de containers, construiremos dos morfismos de containers Π_1 y Π_2 que para todo par de containers vaya desde su producto hasta cada uno de ellos. Sus tipos serán entonces los siguientes:

```
Π₁ : ∀{C D} → (Both C D) ⇒ C
Π₂ : ∀{C D} → (Both C D) ⇒ D
```

Para el caso del primer morfismo proyección, la función de formas retornará el primer valor del par. Es decir, mSh Π_1 será el morfismo proj_1 de conjuntos. La función de posiciones indicará de qué posición en una forma dada (c, d) proviene el dato a almacenar en una posición p de c. En este caso, indicaremos que queremos buscar dentro de las posiciones del primer conjunto. Esto es, mPos (Both C $D \Rightarrow C$) $\{(c, d)\}$ p será inj_1 p.

```
Π₁ = proj₁ , inj₁
```

Análogamente podemos concluir que el segundo morfismo proyección es:

```
Π₂ = proj₂ , inj₂
```

El morfismo de reunión construye un producto a partir de dos morfismos de mismo origen y tendrá la siguiente signatura:

```
⟨_,_⟩ : {A B C : Cont} → (C ⇒ A) → (C ⇒ B) → (C ⇒ Both A B)
```

En cuanto a su funcionamiento, por un lado, la función de formas aplicará las respectivas funciones de formas de los morfismos f y g, armando un par con los resultados. La función de posiciones decide si aplicar mPos f o mPos g, según a qué posición en Both C D se aplique. Esto lo logramos con la función de coreunión de conjuntos [_, _] dada en el código 4.2. Finalmente,

```
⟨ f , g ⟩ = < mSh f , mSh g > , [ mPos f , mPos g ]
```

La prueba de la conmutatividad de cada triángulo del diagrama de la definición la obtenemos trivialmente. El siguiente código muestra la demostración de la conmutatividad izquierda. La derecha no se expone puesto que resulta análoga.

```
Π₁∘⟨_,_⟩≅f   :   {A B C : Cont}(f : C ⇒ A)(g : C ⇒ B)
              →  Π₁ ∘ ⟨ f , g ⟩ ≅ f
Π₁∘⟨ f , g ⟩≅f  = refl
```

Para proveer la prueba de la unicidad del morfismo de reunión tendremos que apelar al lema de equivalencia de morfismos de containers expresado en el código 3.27 (pág, 49) y a los postulados de extensionalidad provistos en 1.26 (pág. 21).

```
pairUnique_Cont  :   {A B C : Cont}
                 →  {f : C ⇒ A}
                 →  {g : C ⇒ B}
                 →  {h : C ⇒ Both A B}
                 →  Π₁ ∘ h ≅ f
                 →  Π₂ ∘ h ≅ g
                 →  h ≅ ⟨ f , g ⟩
pairUnique_Cont refl refl  =
   mEq  refl
        (iext (ext (λ  {         (inj₁ _) → refl
                       ;         (inj₂ _) → refl })))
```

Finalmente, podemos proveer un habitante de HasProducts **Cont**:

Código 4.12. *Formalización de* **Cont** *como categoría con productos*

```
ContHasProducts : HasProducts Cont
ContHasProducts = record
   { Prod        = Both
   ; projl       = Π₁
   ; projr       = Π₂
   ; pair        = ⟨_,_⟩
   ; pairIdl     = λ f g → Π₁∘⟨ f , g ⟩≅f
   ; pairIdr     = λ f g → Π₂∘⟨ f , g ⟩≅g
   ; pairUnique  = pairUnique_Cont }
```

Ejemplos con productos de containers

Ejemplo 4.13. *Par de listas*

Veamos primero un caso muy sencillo de producto de containers donde los dos containers originales son listas. Los pares de listas del mismo tipo se pueden constituir como un container Both cList cList. Se define a continuación un ejemplo de par de listas de naturales que llamamos ceroAdos,tresAsiete.

```
ceroAdos,tresAsiete : ⟦ Both cList cList ⟧ ℕ
ceroAdos,tresAsiete = (3 , 5) , (λ  {  (inj₁ x) → toℕ x
                                    ;  (inj₂ y) → ( 3 + toℕ y ) })
```

La primera lista del par tiene longitud 3 y la segunda, 5. En cuanto al contenido, la primera lista alberga los naturales asociados a cada posición en la lista. Es decir, es la lista del cero al dos. La segunda le suma 3 al índice en la lista, por lo que resulta ser la lista del tres al siete.

Ejemplo 4.14. *Concatenación de listas*

El producto de containers resultará útil para los casos donde queramos definir morfismos que tomen como argumento más de un container, siempre que los elementos almacenados en cada uno de ellos sea el mismo. Por ejemplo, la función de concate-

nación toma dos listas del mismo tipo y construye otra con los elementos de las dos primeras dispuestas de forma contigua. Podemos construir un morfismo:

append : Both cList cList ⇒ cList

La longitud de la nueva lista será la suma de las longitudes de las listas originales.

append = (uncurry _+_) , splitFin

Dadas dos listas de longitudes n_1 y n_2, un índice i en la nueva lista de longitud (n_1+n_2) se corresponderá con una posición de la primera si i es menor a su longitud n_1, en otro caso habrá que buscar el contenido en la segunda lista. Este trabajo lo realiza la función splitFin, que convierte un elemento de Fin $(n_1 + n_2)$ en, o bien uno de tipo Fin n_1, o bien uno de tipo Fin n_2.

splitFin : $\{n_1\ n_2 : \mathbb{N}\}(i : \text{Fin } (n_1 + n_2)) \rightarrow \text{Fin } n_1 \uplus \text{Fin } n_2$
splitFin {zero} i = inj$_2$ i
splitFin {suc n_1} zero = inj$_1$ zero
splitFin {suc n_1} (suc i) = map suc id$_{\text{Set}}$ (splitFin i)

La figura 4.14 expone cómo se realiza la concatenación de los containers de listas de longitud uno y tres. En la parte superior vemos los containers de lista originales y en la parte inferior, el container resultante.

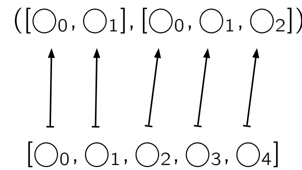


Figura 4.1: Concatenación de containers de listas

La reubicación de posiciones se representa en la figura con la flechas. De esta forma, para el caso particular del ejemplo, la función splitFin reubicará las posiciones de la siguiente manera:

$$\text{splitFin } 0 = \text{inj}_1\ 0$$
$$\text{splitFin } 1 = \text{inj}_1\ 1$$
$$\text{splitFin } 2 = \text{inj}_2\ 0$$
$$\text{splitFin } 3 = \text{inj}_2\ 1$$
$$\text{splitFin } 4 = \text{inj}_2\ 2$$

Si extendemos el morfismo append sobre los naturales, obtendremos una función de pares de listas de naturales en listas de naturales. Podemos aplicar esta función al par de listas del ejemplo 4.13 y obtener una lista del cero al siete.

ceroAsiete : $[\![$ cList $]\!]$ $\mathbb{N}$
ceroAsiete = $[\![$ append $]\!]_m$ ceroAdos,tresAsiete

Para convertir una lista de tipo List en una definida inductivamente para así poder observar y analizar los resultados más simplemente, contamos con la siguiente función:

```
ListTo[] : ∀{A} → List A → [ A ]
ListTo[] (zero , p) = []
ListTo[] (suc s , p) =  p zero :: (ListTo[] (s , (λ i → p (suc i))))
```

Al evaluar ListTo[] ceroAsiete obtenemos efectivamente una lista de naturales del cero al siete :

```
0 ::(1 ::(2 ::(3 ::(4 ::(5 ::(6 ::(7 ::[])))))))
```

4.3. Inicial y terminal

4.3.1. Motivación: (De)construyendo los objetos inicial y terminal

En las diversas secciones de motivación de cada construcción universal teníamos como objetivo el introducir la noción categórica a partir de pensar en teoría de conjuntos, yendo de lo particular a lo general.

¿Cómo hablar del conjunto vacío sin referirnos a sus elementos, o mejor dicho, a la ausencia de ellos? La estrategia es siempre pensar en los morfismos, o en este caso, en las funciones; podemos entonces reformular la pregunta: ¿Qué relaciones se pueden establecer entre el conjunto vacío y el resto de los conjuntos?, ¿qué funciones podemos construir desde o hacia el conjunto vacío?

Hacia el conjunto vacío resulta imposible construir cualquier función. Si pensamos por un momento e intentamos construir dicha función, nos resultará imposible, puesto que el conjunto vacío carece de elementos que oficien de recorrido de la función.

Por otro lado, es muy simple construir una función *desde* el conjunto vacío hacia cualquier otro objeto, simplemente por el hecho de no necesitar proveer ningún valor. La función vacía será la única función que podamos construir. Hemos encontrado una característica del conjunto vacío: siempre existe una única función desde él hacia cualquier otro conjunto. Es este atributo el que le da el nombre de *objeto inicial* a su generalización en teoría de categorías. En la definición 4.15 se expone formalmente.

Como ya hemos comentado en reiteradas ocasiones, si revertimos el sentido de los morfismos en una categoría $\mathscr{C}$ obtenemos una nueva categoría, la categoría dual $\mathscr{C}^{op}$. Allí encontramos los duales de cada construcción, en particular, de la construcción de objeto inicial: el objeto *terminal*. Si dijimos que un objeto es inicial cuando siempre exista un morfismo único hacia cualquier otro objeto, uno terminal será aquél con morfismos únicos *desde* cualquier otro objeto. En la categoría **Set** contamos con muchos objetos terminales. Cualquier conjunto con un único elemento posee la propiedad de aceptar una única función desde cualquier otro conjunto hacia él: la función constante.

4.3.2. Definición y formalización

Definición 4.15. Un objeto $\perp$ en una categoría $\mathscr{C}$ se dice *inicial* si por cada objeto A de $\mathscr{C}$ hay exactamente un morfismo $\perp \to A$ desde él.

$$\perp \overset{!}{\dashrightarrow} A$$

Definición 4.16. Un objeto $\top$ en una categoría $\mathscr{C}$ se dice *terminal* si por cada objeto A de $\mathscr{C}$ hay exactamente un morfismo $A \to \top$ hacia él.

$$A \overset{!}{\dashrightarrow} \top$$

Damos a continuación la implementación en Agda de estos conceptos. Una categoría $\mathscr{C}$: Category tendrá objeto inicial si podemos proveer una instancia del record definido en el código 4.17. Para ello habrá que individualizar un objeto Initial. Además, para poder proclamar la existencia de un morfismo único desde el objeto inicial hacia todo otro objeto, se requiere una forma de construir un habitante de Hom $\mathscr{C}$ Initial X, para todo X y una prueba de su unicidad.

Código 4.17. *Formalización de categoría con objeto inicial*

```
record HasInitial (𝒞 : Category) : Set where
  open Category
  field
    Initial              :  Obj 𝒞
    fromInitMor          :  ∀{X} → Hom 𝒞 Initial X
    isUniqueFromInitMor  :  ∀{X}{f : Hom 𝒞 Initial X}
                         →  fromInitMor {X} ≅ f
```

Asimismo, proveer una instancia de HasTerminal $\mathscr{C}$, record definido en el código 4.18, es prueba formal de la existencia de objeto terminal en la categoría $\mathscr{C}$.

Código 4.18. *Formalización de categoría con objeto terminal*

```
record HasTerminal (𝒞 : Category ) : Set where
  open Category
  field
    Terminal            :  Obj 𝒞
    toTermMor           :  ∀{X} → Hom 𝒞 X Terminal
    isUniqueToTermMor   :  ∀{X}{f : Hom 𝒞 X Terminal}
                        →  toTermMor {X} ≅ f
```

Para ver formalizado lo analizado al comenzar la sección, se expone a continuación los objetos inicial y terminal de la categoría de conjuntos.

Código 4.19. *Formalización de* **Set** *como categoría con objeto inicial*

```
SetHasInitial : HasInitial Set
SetHasInitial = record
  { Initial              = ⊥
  ; fromInitMor          = λ { () }
  ; isUniqueFromInitMor  = ext (λ { () })
  }
```

Código 4.20. *Formalización de* **Set** *como categoría con objeto terminal*

```
SetHasTerminal : HasTerminal Set
SetHasTerminal = record
  { Terminal             = ⊤
  ; toTermMor            = λ _ → tt
```

```
  ; isUniqueToTermMor  = refl
  }
```

Observemos que para el caso del objeto terminal, expuesto en el código 4.20, podríamos haber elegido cualquier conjunto de un elemento. La función toTermMor será simplemente la función que retorne el único elemento del conjunto. La garantía de que dicha función es única la provee el campo isUniqueToTermMor.

4.3.3. Containers vacío y unitario

En esta sección proveeremos instancias de las formulaciones recientemente expuestas, para el caso de la categoría **Cont** de containers.

Si pensamos en que tenemos como objetivo proveer un container inicial tal que siempre exista un morfismo único desde él hacia cualquier otro objeto, entonces podemos empezar pensando en los morfismos. Un morfismo de containers se compone, por un lado, de una función entre formas y por otro de una función en posiciones. Quisiéramos que la construcción a proveer posea inherentemente una única función en formas y, elegida una forma, una única en posiciones. De lo analizado en la sección 4.3.1 de motivación, podemos intuir que el conjunto de las formas deberá ser el vacío. Llamamos Zero al container inicial:

```
Zero : Cont
Zero = ⊥ ◃ (λ _ → ⊥)
```

Como hemos expresado, el morfismo único constará de dos funciones únicas, las funciones vacías. Lo llamamos Zero_m.

```
Zero_m : {C : Cont} → Zero ⇒ C
Zero_m = (λ ()) , (λ {})
```

Proveemos también la demostración de la unicidad del morfismo Zero_m, apelando al lema mEq y a los postulados de extensionalidad, utilizados de forma trivial con las funciones vacías.

```
Zero_mUnique  : {C : Cont}
              → {f : Zero ⇒ C}
              → Zero_m {C} ≅ f
Zero_mUnique  = mEq  (ext   (λ ()))
                     (iext  (λ {  }))
```

En el código 4.21 se presenta la instancia completa, garantizando que la categoría **Cont** cuenta con objeto inicial.

Código 4.21. *Formalización de* **Cont** *como categoría con objeto inicial*

```
ContHasInitial : HasInitial Cont
ContHasInitial = record
  { Initial              = Zero
  ; fromInitMor          = Zero_m
  ; isUniqueFromInitMor  = Zero_mUnique }
```

Notar que así como hemos pensado en construir el objeto inicial de la categoría de containers a partir de la definición e intuición expresadas previamente, podríamos también

haberlo motivado como la introducción de aquél container que representa al tipo vacío. Pensado de esa forma, observemos que ya contamos con una forma de construirlo, puesto que el container vacío no es más que el container constante presentado en el ejemplo 3.13, instanciado en el conjunto vacío:

```
Zero = cK ⊥
```

De la misma forma, el container unitario es aquél que construye el tipo que alberga a un único elemento. Podemos pensarlo como un container constante instanciado en el tipo ⊤ con un único elemento, tt.

```
One = cK ⊤
```

Sin embargo, es posible demostrar que cualquier conjunto con un único elemento funciona igual de bien, con lo que podemos concluir que el objeto terminal no es único en la categoría de containers, como no lo era en **Set**. En teoría de categorías, las restricciones sobre los objetos siempre se dan a través de restricciones sobre los morfismos. Esto es así porque el nivel de abstracción no nos permite analizar cada objeto en particular más que a partir de observar su relación con otros objetos. Esta es la razón por la que no hablaremos de igualdad entre los objetos. Serán los isomorfismos, i.e. ciertos morfismos particulares, los que otorgarán una forma de referirnos a objetos que comparten propiedades equivalentes. Es posible demostrar que el objeto terminal, tanto como el inicial, siempre que existan serán únicos salvo isomorfismos únicos.

Retomando la exposición en la categoría de containers, si desplegamos la definición de cK, vemos que One es el container con forma única, sin posiciones.

```
One : Cont
One = ⊤ ◃ (λ _ → ⊥)
```

El morfismo único hacia el container terminal resulta del morfismo único hacia el conjunto unitario por el lado de las formas y la función vacía por el lado de las posiciones.

```
Oneₘ : {A : Cont} → A ⇒ One
Oneₘ = (λ _ → tt) , (λ ())
```

A continuación, la prueba de la unicidad de esta construcción.

```
OneₘUnique  : {C : Cont}
            → {f : C ⇒ One}
            → Oneₘ {C} ≅ f
OneₘUnique = mEq refl
                 (iext (ext (λ ())))
```

Finalmente, en el código 4.22 proveemos una instancia de HasTerminal **Cont**:

Código 4.22. *Formalización de* **Cont** *como categoría con objeto inicial*

```
ContHasTerminal : HasTerminal Cont
ContHasTerminal = record
  { Terminal            = One
  ; toTermMor           = Oneₘ
  ; isUniqueToTermMor   = OneₘUnique }
```

4.4. Exponencial

4.4.1. Motivación: (De)construyendo el exponencial

Nos interesa comenzar esta sección evocando una problemática simple de la combinatoria de la matemática discreta y preguntarnos: ¿cuántas funciones existen entre dos conjuntos cualesquiera A y B?

En la figura 4.4.1 vemos un ejemplo muy sencillo de dos conjuntos A y B y tres funciones posibles de un conjunto al otro. Se muestran las posibilidades dada una elección determinada de asignación al elemento a_1. Tenemos tantas opciones de mapeo para a_2 como elementos hay en B. La misma cantidad de alternativas existen para asignar a a_1.

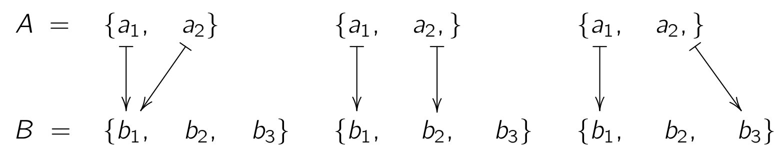


Figura 4.2: Algunas funciones entre los conjuntos A y B

Observemos que si los conjuntos A y B tienen respectivamente a y b cantidad de elementos, tenemos, por cada elemento de A, b opciones de mapeos; entonces existen

$$\underbrace{b \times b \times \cdots b}_{a \text{ veces}} = b^a$$

funciones posibles. Por dicha razón, al conjunto de las funciones de A hacia B le asignamos el nombre de exponencial de B a la A, y lo simbolizamos B^A.

En resumen, en la categoría **Set** de conjuntos y funciones, el objeto exponencial entre dos objetos A y B que denominamos B^A, es el conjunto de las funciones de A en B.

Para empezar a pensar en las funciones hacia cada conjunto exponencial, consideremos las funciones $f(_, _) : A \times B \to C$. Al fijar un elemento $a \in A$ obtenemos una nueva función $f(a, _) : B \to C$; en otras palabras, $f(a, _) \in C^B$. Si hacemos nuevamente variar a entre los elementos de A, obtenemos una nueva función que llamamos $\lfloor f \rfloor$ tal que

$$\lfloor f \rfloor : A \to C^B$$
$$\lfloor f \rfloor(a) = f(a, _)$$

La función $\lfloor f \rfloor$ suele llamarse *currificación* de f en honor al matemático Haskell B. Curry. Definimos la currificación de una función f como $\lfloor f \rfloor(a)(b) = f(a, b)$.

En resumidas cuentas, cualquier función que reciba dos argumentos puede ser currificada y obtener así una nueva función que recibe un solo argumento, retornando una función. Sin embargo, seguir el camino inverso también es posible; para toda función $g : A \to B \to C$ existe una única $\lceil g \rceil : A \times B \to C$ tal que $g = \lfloor \lceil g \rceil \rfloor$. Podemos definirla como:

$$\lceil g \rceil(a, b) = g(a)(b)$$

Es simple de probar que nos encontramos –para cada conjunto C^B– frente a una serie de isomorfismos en **Set**, uno por cada conjunto A. Si tenemos las funciones $f : A \times B \to C$

y $g : A \to C^B$ entonces vemos que se cumple que las dos posibles composiciones de las funciones $\lfloor _ \rfloor$ y $\lceil _ \rceil$ dan como resultado las respectivas identidades.

$\lceil \lfloor f \rfloor \rceil (a, b) = \lfloor f \rfloor (a)(b) = f(a, b)$

$\lfloor \lceil g \rceil \rfloor (a)(b) = \lceil g \rceil (a, b) = g(a)(b)$

4.4.2. Definición y formalización

Antes de continuar con la presentación categórica del objeto exponencial, veamos otro ejemplo de funtor que puede encontrarse en las categorías que cuentan con productos, y que nos será necesario para continuar con la exposición.

Definición 4.23. Sea $\mathscr{C}$ una categoría con productos. Dados dos objetos B y C de $\mathscr{C}$, el funtor $Hom_{\mathscr{C}}(_ \times B, C)$ desde la categoría $\mathscr{C}^{op}$ hacia la categoría **Set** mapea:

- cada objeto A de $\mathscr{C}$ hacia el conjunto de morfismos $Hom_{\mathscr{C}}(A \times B, C)$
- cada morfismo $A' \xrightarrow{f} A$ hacia la función de post-composición

$$_ \circ (f \times id_B) : Hom_{\mathscr{C}}(A \times B, C) \to Hom_{\mathscr{C}}(A' \times B, C)$$

El siguiente diagrama muestra cómo transformamos un morfismo del primer conjunto en uno del segundo a partir de la post-composición, vemos cómo en la categoría $\mathscr{C}$, de componer un morfismo $A \times B \xrightarrow{h} C$ con el morfismo $A' \times B \xrightarrow{f \times id_B} A \times B$ obtenemos un morfismo de $A' \times B$ hacia C.

$$\begin{array}{ccc} A \times B & \xrightarrow{h} & C \\ \uparrow{\scriptstyle f \times id_B} & \nearrow{\scriptstyle h \circ (f \times id_B)} & \\ A' \times B & & \end{array}$$

En el código asociado a la tesina se puede encontrar la formalización y prueba de este funtor, que habremos de llamar HomProd.

Definición 4.24. Sea $\mathscr{C}$ una categoría que cuenta con productos y sean B y C dos de sus objetos. Un objeto C^B es el *exponencial* de C a la B siempre que exista un isomorfismo natural entre los funtores $Hom(_ \times B, C)$ y $Hom(_, C^B)$.

Al igual que con el coproducto y el producto, diremos que una categoría *cuenta con exponenciales* cuando para todo par de objetos existe el exponencial entre ellos.

Del análisis de la definición observamos que para que un objeto C^B sea exponencial deberá existir una transformación natural $\lfloor _ \rfloor : Hom(_ \times B, C) \Rightarrow Hom(_, C^B)$ tal que para cada A, cada una de sus componentes $\lfloor _ \rfloor_A$ es un isomorfismo. Llamaremos $\lceil _ \rceil_A$ a cada una de las inversas. El siguiente diagrama lo expresa de una forma alternativa. La doble línea indica la presencia de un isomorfismo.

$$\lceil_\rceil_A \left(\frac{A \times B \to C}{\overline{A \to C^B}} \right) \lfloor_\rfloor_A$$

Para apreciar mejor las consecuencias de la ley de naturalidad que obtenemos de la transformación natural $\lfloor_\rfloor$, observemos el siguiente diagrama conmutativo. Para todo par de objetos A y A' y morfismo $f : A' \to A$ de la categoría $\mathscr{C}^{op}$, el siguiente diagrama deberá conmutar.

$$\begin{array}{ccc} Hom(A \times B, C) & \xrightarrow{_\circ(f \times id_B)} & Hom(A' \times B, C) \\ \lfloor_\rfloor_A \downarrow \uparrow \lceil_\rceil_A & & \lfloor_\rfloor_{A'} \downarrow \uparrow \lceil_\rceil_{A'} \\ Hom(A, C^B) & \xrightarrow[_\circ f]{} & Hom(A', C^B) \end{array}$$

Además de los isomorfismos ya mencionados, se deberá pedir que se cumpla la ley de naturalidad dada por:

$$\lfloor g \circ (f \times id_B) \rfloor_{A'} \cong \lfloor g \rfloor_A \circ f \qquad \text{para todo } g : A \times B \to C$$

Finalmente formalizamos el hecho de que una categoría $\mathscr{C}$ cuente con exponenciales con la estructura presentada en el código 4.25. El campo Exp construye el objeto exponencial a partir de otros dos cualesquiera. El elemento floor es la transformación natural entre los funtores mencionados anteriormente, siendo el campo nat la formalización de la ley de naturalidad. Finalmente, el campo ceil construye cada uno de los isomorfismos requeridos, y la garantía de serlo está dada por los elementos iso_1 e iso_2.

Código 4.25. *Formalización de categoría con exponenciales*

```
record HasExponentials (𝒞 : Category) (pr : HasProducts 𝒞) : Set where
  field
    Exp   :  Obj 𝒞 → Obj 𝒞 → Obj 𝒞
    floor :  ∀{X Y Z}
          →  Hom 𝒞 (Prod pr X Y) Z
          →  Hom 𝒞 X (Exp Z Y)
    ceil  :  ∀{X Y Z}
          →  Hom 𝒞 X (Exp Z Y)
          →  Hom 𝒞 (Prod pr  X Y) Z
    iso₁  :  ∀{X Y Z}{f : Hom 𝒞 (Prod pr X Y) Z}
          →  ceil (floor f) ≅ f
    iso₂  :  ∀{X Y Z}{f : Hom 𝒞 X (Exp Z Y)}
          →  floor (ceil f) ≅ f
    nat   :  ∀{X X' Y Z : Obj 𝒞}
          →  (g : Hom 𝒞 (Prod pr X Y) Z)
          →  (f : Hom 𝒞 X' X)
          →  floor (comp 𝒞 g (pmap pr f (iden 𝒞))) ≅ comp 𝒞 (floor g) f
```

Nota. Teniendo formalizados el concepto de isomorfismo natural (código 2.21) y los funtores Hom_1 y HomProd (código 2.16 y definición 4.23), podríamos haber formalizado

HasExponentials como un record que contuviera simplemente al constructor Exp y un isomorfismo natural entre los funtores mencionados. Para mayor claridad, evitamos en este caso la construcción de records anidados.

4.4.3. Exponencial de Containers

En esta sección veremos que la categoría **Cont** de containers cuenta con exponenciales [ALS10] y expondremos su formalización.

Con el objetivo presentar el exponencial de containers de la forma más comprensible posible, lo introduciremos a partir del análisis de un ejemplo, para luego presentarlo de forma general.

Ejemplo 4.26. *Currificando* append

Para comenzar, recordemos que tenemos como objetivo construir, dados dos containers B y C, su container exponencial, al que llamaremos $C \hat{} B$. Dicho container deberá satisfacer las reglas de la definición, esto es, deberá existir un isomorfismo natural entre los siguientes morfismos de containers:

$$\frac{\text{Both } A\ B \Rightarrow C}{A \Rightarrow C \hat{} B}$$

El morfismo append es un caso particular de morfismo de la parte superior del isomorfismo, instanciando A, B y C con el container cList.

```
append : Both cList cList ⇒ cList
```

Buscamos construir un morfismo curryappend de tipo cList ⇒ cList ˆ cList isomorfo a append, es decir, sin pérdida de información, manteniendo la posibilidad de reconstruir append a partir de curryappend. Para ello, recordemos (ver ejemplo 4.14, pág. 62) que el morfismo de concatenación arma un container de lista de longitud igual a la suma de las longitudes de las listas argumento; por otro lado, reubica las posiciones según la función splitFin, como se puede apreciar en la figura 4.3, donde la reubicación se indica con flechas.

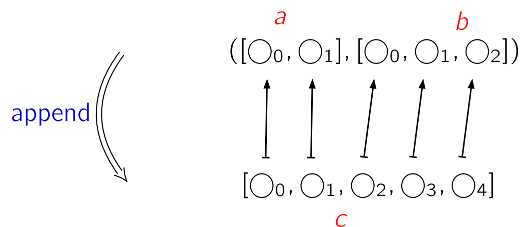


Figura 4.3: Concatenación de containers de listas

Podemos apreciar en la figura 4.4 un panorama de general de comportamiento de la función curryappend y un esquema del objeto exponencial. Para referirnos con más exactitud a cada una de los containers de lista, los llamaremos a, b y c, de longitudes 2, 3 y 5 respectivamente. Así como en la figura 4.3 representábamos al container producto (a, b) entre paréntesis, ahora representamos al exponencial dentro del óvalo de la figura 4.4.

Las formas posibles del exponencial de dos listas incluirán no sólo una forma de construir la longitud de c a partir de las longitudes de las listas a y b, sino también la información de la reubicación de las posiciones de b en las posiciones de c. Más aún, se marcarán con un elemento distinguible tt de tipo $\top$ a aquellas posiciones que se solían asignar a posiciones en a. Esto se ve representado en la figura 4.4 con flechas rectas.

De esta forma, de las únicas posiciones que queda por determinar la procedencia son de aquellas marcadas con el valor tt. Esas posiciones conformarán entonces el conjunto de posiciones del container exponencial. En el ejemplo de la figura 4.4 encontramos dos de estas posiciones, representadas como aquellas de donde parten las flechas curvas.

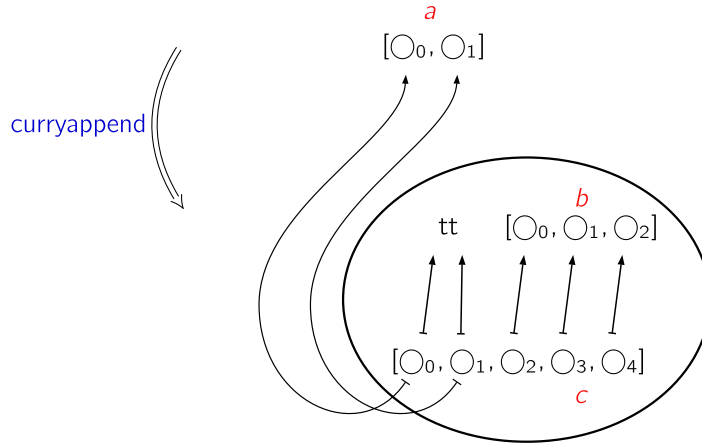


Figura 4.4: Currificación de la función de concatenación

Continuaremos con la presentación de este morfismo una vez presentada la construcción del exponencial de containers.

Definición 4.27. Definimos al constructor de cada objeto exponencial, el elemento _ ^ _, como una función que toma dos containers C y B y retorna otro container.

$$
\begin{aligned}
\hat{} &: \mathsf{Cont} \rightarrow \mathsf{Cont} \rightarrow \mathsf{Cont} \\
C \,\hat{}\, B &= ((b : \mathsf{Sh}\ B) \rightarrow \Sigma[\ c \in \mathsf{Sh}\ C\ ]\ (\mathsf{Pos}\ C\ c \rightarrow \top \uplus \mathsf{Pos}\ B\ b)) \\
&\quad \triangleleft \\
&\quad (\lambda\ f \rightarrow\ \Sigma[\ b \in \mathsf{Sh}\ B\ ]\ (\Sigma[\ q \in \mathsf{Pos}\ C\ (\mathsf{proj}_1\ (f\ b))] \\
&\qquad (\mathsf{proj}_2\ (f\ b)\ q \equiv \mathsf{inj}_1\ \mathsf{tt})))
\end{aligned}
$$

Recapitulando, una forma de un container exponencial es, por un lado, una función desde las formas de B hacia las formas de C. Además, determinada una forma de C, una función que reubique posiciones, mapeando hacia un elemento distintivo tt las posiciones que solían mapearse hacia A. Por otra parte, habrá tantas posiciones en un container exponencial como elementos marcados con el valor tt.

Ejemplo 4.26 (Continuación)**.** Para retomar con el ejemplo introductorio, veamos cómo se implementa la función curryappend ahora que contamos con la definición del exponencial de containers.

$$
\begin{aligned}
\mathsf{curryappend}\ &: \mathsf{cList} \Rightarrow \mathsf{cList}\ \hat{}\ \mathsf{cList} \\
\mathsf{curryappend}\ &= (\lambda\ n_1 \rightarrow (\lambda\ n_2 \rightarrow n_1 + n_2\ ,\ \mathsf{eraseLeft} \circ_{\mathsf{Set}} \mathsf{splitFin}\ \{n_1\})\ ) \\
&\ ,\ \ (\lambda\ \{\ \{n_1\}\ (n_2\ ,\ q\ ,\ p) \rightarrow \mathsf{fromInj}_1\ (\mathsf{splitFin}\ q)\ p\ \})
\end{aligned}
$$

Por un lado, la función de mapeo de formas, de tipo Sh cList → (Sh cList ^ cList), asigna a cada forma n_1 de la primera lista una forma del exponencial de las otras dos listas. Esta forma asignada no es ni más ni menos que la función suma, igual que para el caso de append. Además, incluye una función de posiciones, la misma función splitFin utilizada en append, sólo que con los elementos izquierdos del coproducto *borrados*, i.e. reemplazados por el elemento tt.

La función eraseLeft transforma un elemento de tipo $A \uplus B$ en uno de tipo $\top \uplus B$, convirtiendo todo elemento de la forma inj_1 a en inj_1 tt, dejando igual al resto.

```
eraseLeft : {A B : Set} → A ⊎ B → ⊤ ⊎ B
eraseLeft (inj₁ x) = inj₁ tt
eraseLeft (inj₂ y) = inj₂ y
```

En cuanto al mapeo de posiciones, obtenemos una posición del primer container a partir de una posición del exponencial haciendo uso de la función fromInj_1, a la que se la provee de una posición q y una prueba de que q era de la forma inj_1 tt.

La función fromInj_1 toma un valor de tipo $A \uplus B$ y extrae el valor de tipo A, provista prueba de que realmente nos encontramos en dicho caso. Cuando no sea así, será absurda la existencia de un habitante de la prueba.

```
fromInj₁ : {A B : Set} → (b : A ⊎ B) → (eraseLeft b ≡ inj₁ tt) →  A
fromInj₁ (inj₁ x) pr = x
fromInj₁ (inj₂ y) ()
```

De observar las funciones curryappend y append, presentadas en los ejemplos 4.26 y 4.14 podemos vislumbrar la forma de construir una función ⌊_⌋, que crea un morfismo de tipo $A \Rightarrow C$ ^ B a partir de un morfismo f : Both A $B \Rightarrow C$. El siguiente código muestra su implementación, que hace uso de las funciones auxiliares eraseLeft y fromInj_1.

```
⌊_⌋ : {A B C : Cont} → (Both A B ⇒ C) → (A ⇒ C ^ B)
⌊ f ⌋ = (λ a b → mSh f (a , b) , eraseLeft ∘Set (mPos f) ),
   (λ { {a} (b , c , r) → fromInj₁ (mPos f {a , b} c) r })
```

La inversa de la función ⌊_⌋ es una función ⌈_⌉ dada por:

```
⌈_⌉ : {A B C : Cont} → (A ⇒ C ^ B) → (Both A B ⇒ C)
⌈ f ⌉ = proj₁ ∘Set uncurry (mSh f) ,
   (λ { {a , b} c → insLeft  (proj₂ (mSh f a b) c)
                            (λ pr → (mPos f {a} (b , c , pr))) })
```

donde la función insLeft se comporta de forma contraria a eraseLeft, convirtiendo todo elemento de la forma inj_1 tt en inj_1 x, y dejando sin modificar los elementos de la forma inj_2 y.

```
insLeft : {A B : Set} → (b : ⊤ ⊎ B) → (b ≡ inj₁ tt → A) → A ⊎ B
insLeft (inj₁ _)  pr = inj₁ (pr refl)
insLeft (inj₂ y)  pr = inj₂ y
```

Los elementos restantes que componen el record ContHasExponentials son las pruebas de la naturalidad e isomorfismos. Antes veremos algunas pruebas auxiliares. La siguiente función prueba que insertar la componente izquierda de un valor, dentro del mismo valor luego de haber borrado a la izquierda, es igual a la identidad:

```
lema₁ :  {A B : Set}
```

```
    →   {a : A ⊎ B}
    →   insLeft (eraseLeft a) (fromInj₁ a) ≅ a
lema₁ {a = inj₁ x} = refl
lema₁ {a = inj₂ y} = refl
```

La función lema₂ afirma que borrar a izquierda luego de insertar a izquierda es lo mismo que no hacer nada:

```
lema₂  :  {A B : Set}
    →  {a : ⊤ ⊎ B}
    →  {f : a ≡ inj₁ tt → A}
    →  eraseLeft (insLeft a f) ≅ a
lema₂ {a = inj₁ x} = refl
lema₂ {a = inj₂ y} = refl
```

Extraer el valor izquierdo luego de haberlo insertado retorna como resultado el mismo valor:

```
lema₃  :  {A B : Set}
                          →  {b : ⊤ ⊎ B}
                          →  {f : b ≡ inj₁ tt → A }
                          → fromInj₁ (insLeft b f) ≅ f
lema₃ {b = inj₁ tt}  = dext (λ  { {a = refl} {refl} _
                            → refl })
lema₃ {b = inj₂ _}  = ext (λ { () })
```

Hasta ahora hemos expuesto funciones auxiliares que usarán en las demostraciones de los respectivos isomorfismos, hecho que resulta evidente al observar que en todos los casos probamos que la composición de dos funciones nos da como resultado la identidad. A continuación probaremos dos lemas necesarios para la prueba de la naturalidad. Esta situación también puede vislumbrarse si se observa con atención.

Borrar a izquierda un coproducto al que se le modificó su componente izquierda es equivalente a borrar el coproducto original:

```
lema₄  :    {A B C : Set}{f : A → C}{a : A ⊎ B}
        →  eraseLeft (map f idSet a) ≅ eraseLeft a
lema₄ {a = inj₁ x} = refl
lema₄ {a = inj₂ y} = refl
```

Similarmente, extraer de la izquierda luego de haber modificado dicha componente de un coproducto es lo mismo que extraer y luego aplicar la función.

```
lema₅ :
       ∀{A B C : Set}{f : A → C}{a : A ⊎ B}{p2}
    →  {p1 : eraseLeft (map f idSet a) ≡ inj₁ tt}
    →  (fromInj₁ ∘Set map f idSet) a p1 ≅ f (fromInj₁ a p2)
lema₅ {a = inj₁ x} {p} = refl
lema₅ {a = inj₂ y} {()}
```

En los casos donde tengamos que probar la equivalencia de funciones que toman como argumento pares dependientes, será más cómodo probar la equivalencia de las versiones currificadas de dichas funciones. Para ello, utilizamos el siguiente lema.

```
uncurryEq  :    ∀{A : Set} {B B' : A → Set} {C : Set }
            →  {f : (p : Σ A B) → C } {g : (p : Σ A B') → C}
```

```
         → B ≅ B' → curry f ≅ curry g → f ≅ g
uncurryEq refl p = dcong uncurry p refl refl
```

Finalmente, las pruebas del isomorfismo y la naturalidad se exponen en los códigos 4.28, 4.29 y 4.30. Se utilizan los lemas dcong (código 1.24, pág 20), dext (código 1.28, pág. 22) y dSumEq (código 1.25, pág. 20)

Código 4.28. *Prueba de un lado del isomorfismo entre ⌊_⌋ y ⌈_⌉*

```
iso₁  :   {A B C : Cont}
        → {f : Both A B ⇒ C}
        → ⌈ ⌊ f ⌋ ⌉ ≅ f
iso₁  =   mEq  (ext (λ { (_ , _) → refl }))
               (iext (λ { {_ , _} →
                  ext (λ _ → lema₁)}))
```

Código 4.29. *Prueba del otro lado del isomorfismo entre ⌊_⌋ y ⌈_⌉*

```
iso₂  :   {A B C : Cont}
        → {g : A ⇒ (C ^ B)}
        → ⌊ ⌈ g ⌉ ⌋ ≅ g
iso₂ {A} {B} {C} {g} =
   mEq  (ext  (λ _ → ext (λ _ → dSumEq  refl refl
                                        (ext (λ _ → lema₂)))))
        (iext  (λ {a} →
               uncurryEq  (ext  (λ b →
                                dcong  (Σ (Pos C (mSh ⌈ g ⌉ (a , b))))
                                       (ext (λ _ → cong  (λ x → x ≡ inj₁ tt)
                                                         lema₂))
                                       refl refl))
                          (ext  (λ _ →
                                uncurryEq  (ext (λ _ → cong  (λ x → x ≡ inj₁ tt)
                                                             lema₂))
                                           (ext (λ _ → lema₃))))))
```

Código 4.30. *Prueba de la naturalidad de ⌊_⌋*

```
natural  :   {A A' B C  : Cont}
           → (g : (Both A B) ⇒ C)
           → (f : A' ⇒ A)
           → ⌊ g ∘ (f ×ₘ id) ⌋ ≅ ⌊ g ⌋ ∘ f
natural {A} {A'} {B} {C} g f =
   mEq  (ext (λ _ →
            ext (λ _ → dSumEq  refl refl
                               (ext  (λ c →
                                     lema₄ {a = mPos g c}) ))))
        (iext (λ {a'} →
           uncurryEq  (ext  (λ b →
                            dcong      (Σ (Pos C (mSh (g ∘ (f ×ₘ id)) (a' , b))))
                                       (ext  (λ c →
                                             cong  (λ x → x ≡ inj₁ tt)
                                                   (lema₄ {a = mPos g c})))
                                       refl refl))
```

```
(ext (λ _ →
     uncurryEq (ext (λ c →
                      cong (λ x → x ≡ inj₁ tt)
                           (lema₄ {a = mPos g c})))
               (ext (λ c →
                      dext (λ _ →
                             lema₅ {f = mPos f}
                                   {mPos g c}))))))
```

Presentadas ya cada una de las funciones necesarias para la construcción de una instancia de HasExponential para el caso de la categoría de containers, el siguiente código las reúne, quedando formalmente demostrado que la categoría **Cont** cuenta con exponenciales.

Código 4.31. *Formalización de* **Cont** *como categoría con exponenciales*

```
ContHasExponentials : HasExponentials Cont ContHasProducts
ContHasExponentials = record
  { Exp   = _^_
  ; floor = ⌊_⌋
  ; ceil  = ⌈_⌉
  ; iso₁  = iso₁
  ; iso₂  = iso₂
  ; nat   = natural }
```

4.4.4. Categorías Cartesianas Cerradas

Un resultado interesante de haber formalizado estas construcciones es el haber demostrado que la categoría **Cont** pertenece a un subconjunto de categorías denominadas *cartesianas cerradas*.

Definición 4.32. Una categoría $\mathscr{C}$ se denomina *cartesiana cerrada* si cuenta con objeto terminal, productos y exponenciales.

La formalización de dicho concepto en Agda se presenta en el código 4.33 y la instancia para la categoría **Cont**, en el código 4.34.

Código 4.33. *Formalización de categoría cartesiana cerrada*

```
record IsCartesianClosed (𝒞 : Category) : Set where
  field
    hasTerminal     : HasTerminal 𝒞
    hasProducts     : HasProducts 𝒞
    hasExponentials : HasExponentials 𝒞 hasProducts
```

Código 4.34. *Formalización de* **Cont** *como categoría cartesiana cerrada*

```
ContIsCartesianClosed : IsCartesianClosed Cont
ContIsCartesianClosed = record
  { hasTerminal     = ContHasTerminal
  ; hasProducts     = ContHasProducts
  ; hasExponentials = ContHasExponentials }
```

Como hemos mencionado brevemente en la introducción, la principal relevancia de las categorías cartesianas cerradas se resumen en el hecho de poder ponerlas en correspondencia con el lambda cálculo con tipos simples y la lógica proposicional intuicionista.

Conclusiones

La programación genérica tiene como objetivo primordial el reuso de código. Para lograrlo, apela a la construcción de programas cuyos argumentos puedan pertenecer no sólo a un tipo de datos particular, sino a un conjunto más amplio. Una de las problemáticas principales a la que la programación genérica debe enfrentarse es la de poder definir adecuadamente su ámbito de abstracción. Es decir, para cumplir la meta de construir programas cuyo alcance sea mayor a un único tipo de dato y, en su lugar, funcione para un subconjunto de todos los tipos posibles, es imperativo poder definir con precisión este subconjunto e incluso poder inspeccionar cada habitante en forma individual.

En los lenguajes de tipos dependientes podemos obtener ese subconjunto de forma precisa a partir de definir universos. Estos consisten simplemente en códigos sintácticos, uno por cada tipo a incluir, equipados con una función de extensión que dado un código asigna el tipo de datos que éste representa.

En este trabajo se ha analizado un universo de constructores de tipos en particular, el universo de containers. Se comenzó con un análisis más bien intuitivo, introduciendo múltiples ejemplos. Se expuso la forma de reinterpretar cada container –a partir del cálculo de su extensión– como un constructor de tipos funtorial, es decir, como un endofuntor en **Set**.

Se ha mostrado la forma en que los morfismos de containers se erigen como un universo para representar a las funciones paramétricas entre los tipos de datos generados por containers. Estos resultan ser ni más ni menos que las transformaciones naturales entre los mencionados endofuntores que los containers representan. Se probó que es posible componerlos y siempre existe el morfismo identidad.

Debido a la fuerte relación existente entre la teoría de tipos, la lógica y la programación, dar un salto de abstracción y tomar un punto de vista más general, dejando de lado los detalles de la construcción, otorga sin lugar a dudas una garantía de legitimidad siempre apreciada. Este salto se realiza en el momento en que pasamos a analizar las construcciones en el marco de la teoría de categorías.

La generalización es la esencia de las matemáticas. Muchas veces existen similitudes entre objetos de estudio a priori muy distantes, donde las analogías no se encuentran a simple vista. Abstraer, exponer las definiciones en términos categóricos puede implicar también que demostrar teoremas o encontrar propiedades sobre el objeto de estudio resulte más sencillo. En efecto, de no haber sido tomado este punto de vista categórico, no se hubiera llegado al resultado de que los containers cuentan con exponenciales y son aptos para modelar sistemas de alto orden.

Consecuentemente, se ha expuesto al universo de containers como una categoría denominada **Cont**, cuyos objetos son containers y sus morfismos, los morfismos de containers. Además, se ha provisto una formalización completa en Agda de esta construcción.

Como punto central del trabajo, se han presentado pruebas formales de que la categoría **Cont** cuenta con coproductos, productos, exponenciales, objetos inicial y terminal. Particularmente, es una categoría cartesiana cerrada, hecho que la pone en correspondencia con sistemas como el lambda cálculo simplemente tipado o la lógica proposicional.

Como se ha expuesto en las preliminares, gracias al isomorfismo de Curry-Howard y al requerimiento de que los programas sean totales, programar en Agda y demostrar teoremas son dos tareas en consonancia. Por lo tanto, haber provisto las pruebas formales de que cada uno de los objetos (producto, coproducto, etc) realmente lo son, según parámetros categóricos, implica haber extendido las librerías de containers existentes al día de hoy.

Trabajos futuros

En esta tesina se ha presentado el universo de containers *de único argumento*. Es decir que se ha trabajado a partir de una definición de container donde, al momento de calcular su extensión, se obtienen endofuntores en **Set** de solamente un argumento. Un evidente posible camino a seguir es extender la formalización para incluir containers a más de un argumento.

Una vez realizado este avance, será posible también formalizar el hecho de que la categoría de containers es cerrada bajo álgebras iniciales y coálgebras finales.

Otras extensiones que son posibles de realizarse a partir de las contribuciones de este trabajo son las referidas a la función de extensión, que lleva a cualquier habitante del universo de containers hacia un endofuntor en la categoría de conjuntos. Es posible generalizar dicha función para que resulte en un endofuntor en cualquier categoría extensa y localmente cartesiana cerrada. Además, propiedades como que dicho funtor es totalmente fiel pueden ser formalizadas y así garantizar que los morfismos de containers resulten ser, en efecto, todas y sólo todas las transformaciones naturales entre los definidos endofuntores.

En lo que se refiere al uso de las contribuciones realizadas, se espera que la biblioteca provista sirva como universo para programar genéricamente sobre un subconjunto acotado de tipos de datos, provistas las garantías de que dicho universo es cerrado bajo una serie interesante de construcciones. Por otro lado, se espera que sea un buen aporte como punto de partida para futuras formalizaciones.

Apéndice A

Hoja de ruta del código fuente

La contribución principal del presente trabajo es una librería de containers extendida con las formalizaciones de la clausura de la categoría con respecto al producto, coproducto, exponencial y la existencia de objetos terminal e inicial. Actualmente está alojada en un repositorio *Git* situado en el siguiente enlace: `http://www.github.com/eugeniasimich/containers`. Allí se puede encontrar todo el código Agda analizado en este trabajo, así como también porciones de código menos relevantes que no fueron expuestas, e incluso este documento. A continuación se expondrá un pantallazo general del contenido del mencionado repositorio. Para visualizar más claramente su estructura, observar la figura A.1 (pág. 82).

En la carpeta `doc` se puede encontrar el texto completo de esta tesina, cuyo archivo LaTeX principal es `main.tex`, compilable realizando `make`. En la subcarpeta `code` se encuentra el código simplificado para su exposición y anotado para la compilación del documento.

El resto de los archivos corresponden a la formalización en sí. Los archivos relacionados a la formalización de la teoría de categorías se encuentran en el archivo `Category.agda` y en la carpeta `Category`. El archivo `Extras.agda` contiene definiciones auxiliares utilizadas en muchas de las demostraciones. La formalización de containers se encuentra distribuida en los archivos almacenados en la carpeta `Container` y los archivos `Container.agda`. Cada una de las construcciones sobre containers se encuentra en un archivo distinto.

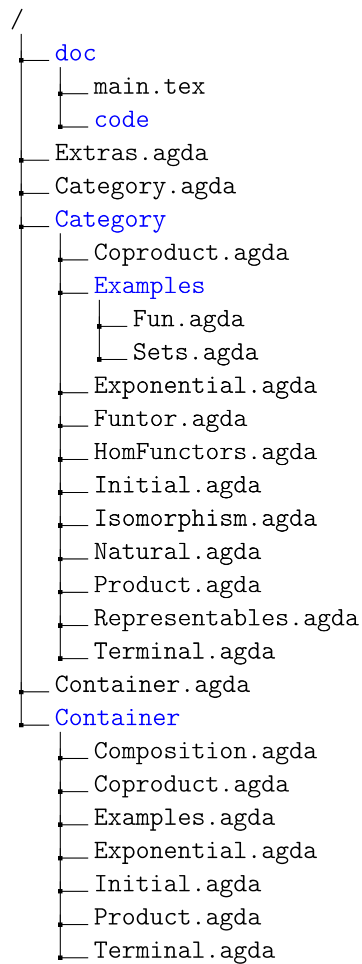


Figura A.1: Organización de los directorios del proyecto

Bibliografía

Referencias históricas

[CF58] Curry, Haskell y Robert Feys: *Combinatory Logic I*. Studies in Logic and the Foundations of Mathematics. North-Holland, 1958. Volume II, with Jonathan Seldin, 1972.

[Chu36] Church, Alonzo: *An unsolvable problem of elementary number theory*. American journal of mathematics, 58(2):345–363, 1936.

[Chu40] Church, Alonzo: *A Formulation of the Simple Theory of Types*. Jurnal of Symbolic Logic, 5(2):56–68, June 1940.

[EML45] Eilenberg, Samuel y Saunders Mac Lane: *General theory of natural equivalences*. Trans. Am. Math. Soc., 58:231–294, 1945.

[Fre79] Frege, Gottlob: *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinenDenkens*. Halle, 1879. Traducción al inglés en [vH67].

[Gen35] Gentzen, Gerhard: *Untersuchungen über das Logische Schliessen*. Mathematische Zeitschrift, 39:176–210 and 405–431, 1935. English translation in [Gen69], pages 68–131.

[Gen69] Gentzen, Gerhard: *The Collected Papers of Gerhard Gentzen*. Studies in Logic and the Foundations of Mathematics. North-Holland, 1969. Edited by M. E. Szabo.

[Gir72] Girard, Jean Yves: *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Tesis de Doctorado, Université Paris VII, 1972.

[God31] Godel, Kurt: *On formally undecidable propositions of Principia Mathematica and related systems*, 1931.

[HA28] Hilbert, David y Wilhelm Ackermann: *Grundzüge der theoretischen Logik*. Springer-Verlag, 1928.

[How80] Howard, William: *The formulae-as-types notion of construction*. En Curry, Haskell, Jonathan Seldin y Roger Hindley (editores): *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, páginas 479–490. Academic Press, 1980, ISBN 0123490502.

[KR35] Kleene, S. C. y J. B. Rosser: *The inconsistency of certain formal logics*. The Annals of Mathematics, Second Series, 36:630–636, 1935. http://www.jstor.org/stable/1968646.

[Lam72] Lambek, Joachim: *Deductive systems and categories III. Cartesian closed categories, intuitionist propositional calculus, and combinatory logic*, páginas 57–82. Springer Berlin Heidelberg, 1972, ISBN 978-3-540-37609-5. http://dx.doi.org/10.1007/BFb0073965.

[Tur36] Turing, Alan M.: *On Computable Numbers, with an Application to the Entscheidungsproblem*. Proceedings of the London Mathematical Society, 2(42):230–265, 1936.

[vH67] Heijenoort, J. van: *From Frege to Gödel*. Harvard University Press, 1967.

[Wad15] Wadler, Philip: *Propositions As Types*. Commun. ACM, 58(12):75–84, Noviembre 2015, ISSN 0001-0782. http://doi.acm.org/10.1145/2699407.

[WR13] Whitehead, Alfred North y Bertrand Arthur William Russell: *Principia Mathematica*. Cambridge University Press, first edición, 1910–1913. Three volumes.

Bibliografía general

[AAG03] Abbott, Michael, Thorsten Altenkirch y Neil Ghani: *Categories of Containers*. En *Proceedings of Foundations of Software Science and Computation Structures*, páginas 23–38, Berlin, Heidelberg, 2003. Springer-Verlag, ISBN 3-540-00897-7. http://dl.acm.org/citation.cfm?id=1754809.1754813.

[AAG05] Abott, Michael, Thorsten Altenkirch y Neil Ghani: *Containers - Constructing Strictly Positive Types*. Theoretical Computer Science, 342:3–27, September 2005. Applied Semantics: Selected Topics.

[Abb03] Abbott, Michael: *Categories of Containers*. Tesis de Doctorado, University of Leicester, October 2003.

[ACU14] Altenkirch, Thorsten, James Chapman y Tarmo Uustalu: *Relative Monads Formalised*. Journal of Formalized Reasoning, 7(1):1–43, 2014, ISSN 1972-5787. http://jfr.unibo.it/article/view/4389.

[AGH+15] Altenkirch, Thorsten, Neil Ghani, Peter Hancock, Conor McBride y Peter Morris: *Indexed containers*. Journal of Functional Programming, 25:e5, 2015.

[ALS10] Altenkirch, Thorsten, Paul Levy y Sam Staton: *Higher Order Containers*. Computability in Europe, 2010.

[AMM07] Altenkirch, Thorsten, Conor Mcbride y Peter Morris: *Generic Programming with Dependent Types*. En Backhouse, Roland, Jeremy Gibbons, Ralf Hinze y Johan Jeuring (editores): *Spring School on Datatype-Generic Programming*, volumen 4719 de *LNCS*. Springer-Verlag, 2007.

[Awo10] Awodey, Steve: *Category Theory*. Oxford University Press, 2010.

[BGHJ07] Backhouse, Roland, Jeremy Gibbons, Ralf Hinze y Johan Jeuring (editores): *Datatype-Generic Programming: International Spring School, SSDGP 2006, April 24-27, 2006, Revised Lectures*. Springer Berlin Heidelberg, Nottingham, UK, 2007, ISBN 978-3-540-76786-2.

[Bra] Brady, Edwin: *Idris*. http://www.idris-lang.org/.

[Bra13] Brady, Edwin: *Idris: General Purpose Programming with Dependent Types*. En *Proceedings of the 7th Workshop on Programming Languages Meets Program Verification*, PLPV '13, páginas 1–2, New York, NY, USA, 2013. ACM, ISBN 978-1-4503-1860-0. http://doi.acm.org/10.1145/2428116.2428118.

[BW95] Barr, Michael y Charles Wells: *Category Theory for Computing Science, 2Nd Ed.* Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1995, ISBN 0-13-323809-1.

[Coq04] Coq development team: *The Coq proof assistant reference manual*. LogiCal Project, 2004. http://coq.inria.fr, Version 8.0.

[GK10] Gambino, Nicola y Joachim Kock: *Polynomial functors and polynomial monads*. 2010.

[Jay95] Jay, C. Barry: *A Semantics for Shape*. Science of Computer Programming, 25:25–251, 1995.

[JC94] Jay, C. Barry y J. R. B. Cockett: *Shapely types and shape polymorphism*, páginas 302–316. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994, ISBN 978-3-540-48376-2. http://dx.doi.org/10.1007/3-540-57880-3_20.

[Joy81] Joyal, André: *Une théorie combinatoire des séries formelles*. Advances in Mathematics, 42(1):1 – 82, 1981.

[Mac50] MacLane, Saunders: *Duality for groups*. Bull. Amer. Math. Soc., 56(6):485–516, Noviembre 1950. http://projecteuclid.org/euclid.bams/1183515045.

[McB02] McBride, Conor: *Elimination with a Motive*. En Callaghan, Paul, Zhaohui Luo, James McKinna y Robert Pollack (editores): *Types for Proofs and Programs (Proceedings of the International Workshop, TYPES'00)*, volumen 2277 de *LNCS*. Springer-Verlag, 2002.

[McB04] McBride, Conor: *Epigram*, 2004. http://www.dur.ac.uk/CARG/epigram.

[ML75] Martin-Löf, Per: *An intuitionistic theory of types: predicative part*. En Rose, H.E. y J.C. Shepherdson (editores): *Logic Colloquium '73, Proceedings of the Logic Colloquium*, volumen 80 de *Studies in Logic and the Foundations of Mathematics*, páginas 73–118. North-Holland, 1975.

[ML82] Martin-Löf, Per: *Constructive mathematics and computer programming*. En Cohen, L. Jonathan, Jerzy Łoś, Helmut Pfeiffer y Klaus Peter Podewski (editores): *Logic, Methodology and Philosophy of Science VI, Proceedings of the Sixth International Congress of Logic, Methodology and Philosophy of Science, Hannover 1979*, volumen 104 de *Studies in Logic and the Foundations of Mathematics*, páginas 153–175. North-Holland, 1982. http://dx.doi.org/10.1016/S0049-237X(09)70189-2.

[ML84] Martin-Löf, Per: *Intuitionistic type theory*, volumen 1 de *Studies in Proof Theory*. Bibliopolis, 1984, ISBN 88-7088-105-9.

[ML98] Martin-Löf, Per: *An intuitionistic theory of types*. En Sambin, Giovanni y Jan M. Smith (editores): *Twenty-five years of constructive type theory (Venice, 1995)*, volumen 36 de *Oxford Logic Guides*, páginas 127–172. Oxford University Press, 1998.

[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.

[RP90] Reynolds, John C. y Gordon D. Plotkin: *On functors expressible in the polymorphic typed lambda calculus*. En *Logical Foundations of Functional Programming*, páginas 127–152. Addison-Wesley, 1990.