FUNDAMENTOS DEL DEMOSTRADOR LEAN

Rodrigo Pallares Palencia

FUNDAMENTOS DEL DEMOSTRADOR LEAN

Rodrigo Pallares Palencia

Trabajo de Fin Máster presentado como requisito para la obtención del título del Máster en Lógica, Computación e Inteligencia Artificial por la Universidad de Sevilla.

Tutelado por

Francisco Jesús Martín Mateos

Índice general

English Abstract

Lean is a theorem prover based on dependent type theory that aims to bridge the gap between interactive and automated theorem proving. The main goal of this project is to understand the axiomatic foundations that make up Lean's underlying logic. For that purpose we first explore the type dependent theories which Lean is based on – namely, Coquand and Huet's *Calculus of Constructions* [1] and Dybjer's *Inductive Families* [2]. We also take a look at Martin-Löf's *Intuitionistic Type Theory* [3], which serves as a starting point for both aforementioned theories. Secondly, we turn the attention to the Lean Tutorial [4] in order to study how the Calculus of Constructions and the Inductive Families *actually* make up Lean's axiomatic framework, along with some extensions and variations. The study of Lean's framework will be accompanied by actual examples written in Lean code. To wrap up the project, we put into practice everything learned about Lean by formalizing some basic concepts about Set Theory and Topology. We do so inspired by *mathlib* [5], a user-maintained library of mathematics formalized in Lean.

Introducción

La forma estándar de respaldar una afirmación matemática es proporcionar una prueba. En términos generales, hay dos formas en las que un ordenador puede asistir en la demostración de una afirmación: puede ayudar a encontrar una prueba, y puede ayudar a verificar si una prueba dada es correcta. La primera posibilidad suele denominarse prueba **automatizada** de teoremas, mientras que nos referimos a la segunda como prueba **interactiva** de teoremas.

Por lo general, la prioridad de la automatización de pruebas es la potencia y eficiencia de búsqueda, a veces a costa de la solidez del sistema. Por otro lado, las pruebas interactivas suelen permitir la construcción de pruebas como objetos del sistema, exigiendo que toda afirmación esté estrictamente respaldada por una prueba válida a partir de ciertos axiomas y haciendo que las demostraciones requieran mucha interacción por parte del usuario.

El demostrador Lean[1] busca salvar la distancia entre las pruebas automatizadas y las interactivas. Desarrollado por *Microsoft Research* y lanzado originalmente en 2013, Lean consta de una infraestructura fundacional que hace compatible la interacción del usuario para la construcción de pruebas axiomáticas con el uso de herramientas y métodos automatizados.

A continuación se resume brevemente el contenido de cada capítulo del presente trabajo.

En el primer capítulo recogemos las teorías fundacionales en las que se sustenta la infraestructura axiomática de Lean. Más concretamente nos centramos en el Cálculo de Construcciones de Coquand y Huet [1] y en las Familias Inductivas de Dybjer [2], pasando por la Teoría Intuicionista de Tipos de Martin-Löf [3], que sirve como punto de partida para ambos.

[1] https://leanprover.github.io

En el segundo capítulo seguimos parte del tutorial de Lean [4] para estudiar cómo la infraestructura de Lean adapta las teorías vistas en el primer capítulo, introduciendo variaciones y extensiones. Paralelamente vemos cómo la teoría fundacional de Lean se corresponde con el propio lenguaje de programación del sistema.

En el tercer capítulo vemos cómo Lean puede usarse para formalizar teorías matemáticas. Para ello tomamos como referencia la librería *mathlib*[2] [5], un repositorio de matemáticas formalizadas en Lean mantenido por las contribuciones de la comunidad. Concretamente formalizamos algunos conceptos de Topología, pasando por Teoría de Conjuntos.

En el cuarto y último capítulo repasamos todo lo estudiado a lo largo del trabajo. También se proponen posibles futuras líneas de trabajo relativas al estudio de Lean y a su utilización para la formalización de matemáticas.

[2] https://leanprover-community.github.io/mathlib_docs/

1 Teoría de tipos dependientes

La teoría de tipos es un paradigma fundacional de las matemáticas alternativo a la teoría de conjuntos. En realidad no debemos hablar de *la* teoría de tipos, sino de *una* teoría de tipos, pues existen diversas variaciones. Entre ellas se encuentran el λ-cálculo (o teoría de tipos simples) de Alonzo Church, la teoría de tipos intuicionista de Martin-Löf o la más reciente teoría de tipos homotópica.

En este capítulo veremos en primer lugar las características comunes entre las diferentes teorías de tipos, prestando especial atención a la categoría de las teorías con tipos dependientes. Posteriormente estudiaremos el caso particular del cálculo de construcciones y su extensión con familias inductivas, que son la principal fuente de inspiración para la teoría de tipos implementada en el sistema Lean.

1.1 Teoría de tipos

Si descendemos al nivel más básico, hacer matemáticas se reduce esencialmente a manipular símbolos de acuerdo con una serie de reglas. En general, las reglas son muy complicadas, por eso el objetivo de establecer un sistema fundacional es precisamente poder derivar las reglas más complejas a partir de una lista (idealmente) pequeña de reglas básicas. En una teoría de tipos estas reglas básicas son un cálculo de términos, y cada término es de un tipo. Así, las reglas definen términos y sus tipos, y el principal juicio que puede emitirse es de la forma "t es un término bien construido de tipo A".

1.1.1 Términos y tipos

Generalmente, en teoría de tipos hay una distinción entre términos y tipos, junto con una relación $t : A$ que significa "el término t es de tipo A". Suele haber uno o varios tipos básicos a partir de los cuales pueden definirse otros tipos. Los tipos adicionales se definen en el marco de la deducción natural, es decir, a partir de las siguientes reglas:

- **Regla de formación**: describe cómo construir el nuevo tipo a partir de otros.
- **Reglas de introducción**: determinan cómo construir términos del tipo que se está definiendo.
- **Regla de eliminación**: describe cómo construir términos de otros tipos a partir de términos del nuevo tipo definido. Dicho de otra forma, la regla de eliminación de un tipo permite definir una función cuyo dominio es ese tipo.
- **Reglas de igualdad**: relacionan las reglas de introducción y la regla de eliminación, estableciendo cómo las funciones definidas mediante la regla de eliminación actúan sobre elementos construidos mediante la regla de introducción.

La mayoría de teorías de tipos tienen un enfoque constructivista, donde todos los términos tienen un comportamiento computacional incorporado, de hecho, a veces nos referiremos a los términos como *programas*. En el caso de los términos cuyo tipo está definido por las reglas anteriores, las reglas de formación e introducción describen la forma que tienen los tipos y sus términos, mientras que las reglas de eliminación e igualdad describen el contenido computacional de esos términos.

1.1.2 Juicios, contextos, consecuentes y reglas

En teoría de tipos la distinción entre proposiciones y juicios es fundamental. Por un lado, las proposiciones pueden o no ser ciertas, y las podemos combinar por medio de operaciones lógicas. Por otro, cuando afirmamos que una proposición es cierta, estamos emitiendo un juicio. Los juicios más comunes en teoría de tipos son declaraciones de tipado ($t : A$) e igualdad (o equivalencia) entre términos de un mismo tipo ($t = t' : A$).

Un contexto es una lista de declaraciones de tipado, donde cada término es una variable que no ha aparecido antes en la lista. Cada teoría define de forma precisa

cómo construir contextos válidos, pero la idea en general es que cada tipo A que aparece en un contexto válido Γ debe tener sentido en el contexto declarado hasta la aparición de A.

Los consecuentes en teoría de tipos son de la forma $\Gamma \vdash J$, donde Γ es un contexto válido y J es un juicio. Las reglas de inferencia nos permiten derivar consecuentes a partir de otros. La forma general de una regla es

$$\frac{\Gamma_1 \vdash t_1 : A_1 \qquad \cdots \qquad \Gamma_n \vdash t_n : A_n}{\Delta \vdash s : B}, \tag{1.1}$$

que significa que podemos derivar $\Delta \vdash s : B$ si los consecuentes $\Gamma_1 \vdash t_1 : A_1, \ldots, \Gamma_n \vdash t_n : A_n$ son válidos.

Las reglas de una teoría de tipos pueden clasificarse atendiendo a la función que cumplen en la teoría. Algunas categorías comunes son:

- **Reglas de conversión**: determinan cuándo ciertos términos pueden reescribirse manteniendo su comportamiento computacional. Por ejemplo, la α-conversión permite renombrar las variables ligadas de un término.
- **Reglas de sustitución**: permiten sustituir iguales por iguales en cualquier expresión.
- **Reglas de definición de tipos**: son las reglas de formación, introducción, eliminación e igualdad descritas previamente.
- **Reglas de universos**: son las reglas de determinan el comportamiento de los universos presentes en la teoría (ver siguiente apartado).

En muchos casos las reglas de conversión y de sustitución se dan por hecho y se omiten a la hora de definir la teoría.

1.1.3 Universos

En general las teorías de tipos necesitan alguna forma de declarar que algo es un tipo. Algunas lo hacen por medio de un juicio de la forma "A type"; otras, en cambio, lo resuelven tratando los tipos como términos de un tipo *universal* que suele llamarse Type, Sort, $*$ o similar.

Tiene sentido plantearse entonces si tal tipo universal Type debe ser también tratado como un término y si por tanto debe ser de un tipo. Algunas teorías dictaminan que el universo Type es un tipo especial que no tiene tipo. Otras deciden que Type, igual que el resto de tipos, debe ser de un cierto tipo. En tal caso, si queremos una teoría consistente, imponer Type : Type no es una opción, así que se opta por introducir una *jerarquía de universos* de forma que $\text{Type} = \text{Type}_0$ es de tipo Type_1, que a su vez es de tipo Type_2...

1.1.4 Proposiciones como tipos: la correspondencia de Curry-Howard

La correspondencia (o isomorfismo) de Curry-Howard establece una relación biunívoca entre los sistemas de demostración en lógica y los sistemas de tipos en ciencias de la computación. Más concretamente se trata de una correspondencia a dos niveles: las proposiciones se corresponden con tipos, y sus pruebas se corresponden con términos de esos tipos.

La teoría de tipos aprovecha la correspondencia de Curry-Howard para incorporar las proposiciones sin necesidad de definir un lenguaje adicional paralelo al de la propia teoría. Una proposición P puede verse como el tipo de todas sus pruebas y un término t de tipo P es por tanto una prueba de P. Así, el juicio $t : P$ significa "t es una prueba de P".

Por ejemplo, si P y Q son proposiciones, podemos ver el tipo $P \to Q$ como la proposición "P implica Q" y, si dado un término $t : P$, la expresión $e(t)$ es una prueba de Q, entonces $(\lambda x : P)e(x)$ (la función que para cada x de tipo P devuelve $e(x)$) es una prueba de $P \to Q$. Más aún, un término (una prueba) $f : P \to Q$ es una función, así que dada una prueba $t : P$, tenemos que $f\,t$ (f aplicado a t) es una prueba de Q. Es decir, la aplicación de funciones se corresponde precisamente con el *modus ponens*. Esta analogía entre las reglas de la implicación en un sistema de deducción lógica y las reglas de λ-abstracción y aplicación para funciones es una instancia de la correspondencia de Curry-Howard. Veremos más ejemplos a lo largo del trabajo.

Desde el punto de vista constructivista, el paradigma de las proposiciones como tipos es fiel a lo que significa ser una proposición: una proposición es una especificación del tipo de datos (o de los programas) que constituyen una prueba de la misma. Así, una prueba de P no es más que un término $t : P$ del tipo correcto. Quienes no estén alineados con la perspectiva constructivista pueden pensar en este enfoque co-

mo un truco ingenioso: a cada proposición le asociamos un tipo vacío si es falsa o un tipo *habitado*[1] si es cierta.

En muchos casos las teorías de tipos implementan lo que se conoce como *irrelevancia de las pruebas*: dada una proposición P, cualesquiera dos términos $a : P$ y $b : P$ son tratados como equivalentes en el sistema. Así, aunque podemos tratar las pruebas $t : P$ como términos en el lenguaje de la teoría de tipo, estos no contienen información más allá de que P es cierta. Bajo la interpretación descrita en el párrafo anterior, la irrelevancia de las pruebas se corresponde con asociar una proposición que es cierta con un tipo formado por un único término.

1.2 Tipos dependientes. Teoría intuicionista de tipos

La teoría intuicionista de tipos (ITT) es una teoría de tipos desarrollada por Per Martin-Löf [3] como soporte para las matemáticas constructivistas. La característica principal que diferencia a la teoría intuicionista de la teoría de tipos simples de Church es la inclusión de los tipos dependientes, es decir, se permite la construcción de familias arbitrarias de tipos.

1.2.1 Juicios

El objetivo principal de la ITT es la construcción de un sistema de reglas formales capaz de representar de la mejor forma posible el razonamiento matemático informal. Es por eso que durante la descripción de la teoría se intenta no separar la sintaxis de la semántica; cada vez que se presenta un juicio o una regla de inferencia, se proporciona inmediatamente una justificación semántica en concordancia.

En la teoría intuicionista de tipos hay cuatro tipos de juicio:

(1) A es un tipo (A type)

(2) A y B son tipos iguales ($A = B$)

(3) a es un término de tipo A ($a : A$)

(4) a y b son términos iguales de tipo A ($a = b : A$)

[1] Un tipo A se dice habitado si existe al menos un término de tipo A

Cada fila de la siguiente tabla muestra una interpretación distinta de los juicios:

A type	$a : A$	
A es un tipo	a es un término de tipo A	A está habitado
A es una proposición	a es una prueba (construcción) de la proposición A	A es cierta
A es un objetivo	a es un método para alcanzar el objetivo A	A es alcanzable
A es un problema	a es un método/programa para resolver el problema A	A tiene solución

La analogía entre la primera y la segunda interpretación se hace explícita en la correspondencia de Curry-Howard. La última columna describe la lectura de cada interpretación en el paradigma de la irrelevancia de las pruebas.

Una regla de inferencia se justifica explicando la conclusión bajo la asunción de que las premisas son ciertas. Por lo tanto, para poder justificar las reglas de inferencia de la teoría, primero es necesario entender qué necesitamos saber para tener el derecho de emitir un juicio. A continuación se da una explicación semántica de cada juicio.

(1) A es un tipo

Un tipo A se define dando una receta de cómo formar un término canónico de A y de cómo formar dos términos canónicos iguales en A.

Por ejemplo, el tipo $\mathbb{N}$ de los números naturales queda definido mediante las siguientes reglas:

$$0 : \mathbb{N} \qquad \frac{a : \mathbb{N}}{s(a) : \mathbb{N}} \qquad 0 = 0 : \mathbb{N} \qquad \frac{a = a' : \mathbb{N}}{s(a) = s(a') : \mathbb{N}} \tag{1.2}$$

Los términos canónicos de un tipo son por tanto aquellos cuya forma es exactamente como dictan las reglas que describen al tipo.

La única restricción a la hora de definir un tipo es que la igualdad entre términos canónicos debe ser reflexiva, simétrica y transitiva.

(2) A y B son tipos iguales

Al emitir un juicio de la forma $A = B$ estamos suponiendo que A y B son tipos, es decir, que sabemos cómo formar términos canónicos (iguales) de A y de B. Así, dos tipos A y B son iguales si tienen los mismos términos canónicos y si la igualdad entre estos se da simultáneamente en ambos tipos, es decir, si para elementos a y b cualesquiera se tienen

$$\frac{a : A}{a : B} \text{ (es decir, } \frac{a : A}{a : B} \text{ y } \frac{a : B}{a : A}\text{)} \tag{1.3}$$

y

$$\frac{a = b : A}{a = b : B}. \tag{1.4}$$

(3) El término a es de tipo A

Al afirmar $a : A$ estamos suponiendo que A es un tipo y, en particular, que sabemos cómo se forman los términos canónicos de A. Así, un término cualquiera de tipo A es un método o programa que al ejecutarse devuelve como resultado un término canónico de A.

La computación de un término a de A se considera terminada cuando se alcanza un valor cuya forma es la de un término canónico de A.

(4) Los términos a y b de tipo A son iguales

Dos términos arbitrarios a y b del tipo A son iguales si al ejecutarse (de acuerdo con la definición anterior) dan como resultado términos canónicos iguales en A.

1.2.2 Proposiciones

Desde el punto de vista intuicionista, una proposición se define decribiendo cómo deben ser sus pruebas. La siguiente tabla muestra cómo las operaciones lógicas encajan en la interpretación intuicionista de las proposiciones:

Una prueba de la proposición	tiene la forma
$\bot$	–
$A \wedge B$	(a, b), con a una prueba de A y b una prueba de B
$A \vee B$	$i(a)$, donde a es una prueba de A, o $j(b)$, donde b es una prueba de B
$A \to B$	$(\lambda x)b(x)$, donde $b(a)$ es una prueba de B si a es una prueba de A
$(\forall x)B(x)$	$(\lambda x)b(x)$, donde $b(a)$ es una prueba de $B(a)$ para cada elemento a
$(\exists x)B(x)$	(a, b), donde a es un elemento y b es una prueba de $B(a)$

La tabla anterior muestra únicamente las pruebas *canónicas* de cada proposición. Una prueba arbitraria de una proposición es un método que produce una prueba canónica de la misma.

En definitiva, una proposición se define estableciendo cómo se construyen sus pruebas canónicas, de forma completamente análoga a lo que sucede con los tipos. En vez de duplicar las nociones, la ITT opta por identificar tipos y proposiciones. A veces incluso escribiremos A prop. en lugar de A type para enfatizar que el tipo A está siendo interpretado como una proposición.

Veremos posteriormente que cuando interpretamos los tipos como proposiciones generalmente se omiten las pruebas, pues lo relevante es si existe o no una prueba. Así, escribiremos A true en lugar de $a : A$ cuando A sea una proposición.

1.2.3 Tipos dependientes y juicios hipotéticos

En ITT se permite la construcción de expresiones cuyo tipo depende de otro término. Dado un tipo A, podemos tener una expresión $b(x)$ de forma que $b(a)$ es un término de tipo $B(a)$ para cada $a : A$. Se dice que el tipo $B(x)$ de la expresión $b(x)$ es dependiente (de A). Podemos ver $B(x)$ simplemente como una familia de tipos indexada en A.

La noción de tipo dependiente nos permite introducir hipótesis en los cuatro tipos de juicios vistos en la sección 1.2.1:

(1) $B(x)$ type $(x : A)$

Este juicio significa que dado cualquier término a de A, $B(a)$ es un tipo y que si a y a' son términos iguales de A entonces $B(a)$ y $B(a')$ son tipos iguales. Esto justifica las siguientes reglas de sustitución:

$$\frac{a : A \qquad B(x) \text{ type}\ (x : A)}{B(a) \text{ type}} \qquad \frac{a = a' : A \qquad B(x) \text{ type}\ (x : A)}{B(a) = B(a')} \tag{1.5}$$

El resto de juicios tienen significados y reglas de sustitución análogos.

(2) $B_1(x) = B_2(x)$ $(x : A)$

(3) $b(x) : B(x)$ $(x : A)$

(4) $b_1(x) = b_2(x) : B(x)$ $(x : A)$

Los juicios hipotéticos pueden generalizarse a un número arbitrario (pero finito) de hipótesis. Veremos esta idea con más detalle por medio de la definición de *contexto* presentada en el Cálculo de Construcciones (ver Sección 1.3).

1.2.4 Definición de tipos básicos en ITT

A la hora de definir tipos, Martin-Löf sigue la estructura de reglas descrita en la sección 1.1.1: cada tipo tiene una regla de formación, reglas de introducción (posiblemente ninguna), una regla de eliminación y reglas de igualdad. De acuerdo con la semántica asociada a los términos y tipos, las reglas de introducción determinan cómo

construir *términos canónicos* del tipo que se está definiendo y las reglas de igualdad describen cómo las reglas de eliminación actúan sobre los términos canónicos.

A continuación se presentan las reglas de definición de algunos de los tipos básicos en ITT: los productos dependientes, los Σ-tipos y la igualdad proposicional.

Producto dependiente (Π)

Si entendemos los tipos como conjuntos, los productos dependientes se corresponden con los productos cartesianos de familias de tipos indexadas en otro tipo. Las reglas de Π-formación son las siguientes:

$$\frac{\begin{array}{cc}(x : A)\\ A \text{ type} \qquad B(x) \text{ type}\end{array}}{(\Pi x : A)B(x) \text{ type}} \qquad \frac{\begin{array}{cc}(x : A)\\ A = C \qquad B(x) = D(x)\end{array}}{(\Pi x : A)B(x) = (\Pi x : C)D(x)} \tag{1.6}$$

Todas las reglas de tipos incluyen una regla análoga a la segunda regla de 1.6, que permite extender la igualdad de tipos y términos; este prototipo de reglas se omitirá a partir de ahora. La conclusión de la primera regla de 1.6 afirma que algo es un tipo. La información sobre *qué tipo es* viene "codificada" en las reglas de introducción:

$$\frac{\begin{array}{c}(x : A)\\ b(x) : B(x)\end{array}}{(\lambda x)b(x) : (\Pi x : A)B(x)} \tag{1.7}$$

Es decir, los términos canónicos de $(\Pi x : A)B(x)$ son de la forma $(\lambda x)b(x)$, donde $b(x) : B(x)$ $(x : A)$ es una expresión de tipo dependiente. Si entendemos la expresión $b(x)$ como una función, entonces la λ-abstracción $(\lambda x)b(x)$ puede interpretarse como un nombre para esa función. Al fin y al cabo, el producto dependiente $(\Pi x : A)B(x)$ puede verse como el tipo de las funciones con dominio en A y cuyo codominio $B(x)$ depende del valor $x : A$ de entrada[2].

La regla de Π-eliminación codifica la interpretación computacional de los términos del producto dependiente:

$$\frac{c : (\Pi x : A)B(x) \qquad a : A}{\mathrm{Ap}(c, a) : B(a)} \tag{1.8}$$

Hemos visto que los elementos de $(\Pi x : A)B(x)$ pueden verse como (nombres de) funciones. Así, dado el nombre $c : (\Pi x : A)B(x)$ de una función, $\mathrm{Ap}(c, a)$ calcula el

[2]De hecho, los productos dependientes también se denominan tipos de *funciones dependientes*.

valor de esa función en $a : A$. En términos de la interpretación descrita en la sección 1.2.1, $\mathrm{Ap}(c, a)$ es un método para obtener un término canónico de $B(a)$ a partir de c. A continuación se explica cómo ejecutar el método. Sabemos que $c : (\Pi x : A)B(x)$, es decir, que c es un método que devuelve un término canónico $(\lambda x)b(x)$. Ahora tomamos $a : A$ y sustituimos x por a en $b(x)$, obteniendo $b(a) : B(a)$. Ejecutando el método $b(a)$ obtenemos un término canónico de $B(a)$, como buscábamos.

Por supuesto, la explicación anterior es puramente semántica; en realidad no se lleva a cabo ningún tipo de computación.

Por último, las reglas de Π-igualdad describen cómo interaccionan los términos canónicos con la regla de eliminación:

$$\frac{\begin{array}{c}(x : A)\\ a : A \qquad b(x) : B(x)\end{array}}{\mathrm{Ap}((\lambda x)b(x), a) = b(a) : B(a)} \qquad \frac{c : (\Pi x : A)B(x)}{c = (\lambda x)\mathrm{Ap}(c, x) : (\Pi x : A)B(x)} \tag{1.9}$$

La primera regla se conoce comúnmente como regla de β-conversión; y la segunda, como regla de η-conversión.

Cuantificador universal (∀)

Sean A un tipo y $B(x)$ una familia de tipos indexada en A. Si interpretamos $B(x)$ como una función proposicional (es decir, $B(a)$ es una proposición para cada $a : A$), entonces podemos reinterpretar $(\Pi x : A)B(x)$ como una cuantificación universal sobre los elementos de A, es decir, podemos definir:

$$(\forall x : A)B(x) \equiv (\Pi x : A)B(x)$$

Podemos ahora reinterpretar las reglas correspondientes a la operación Π. Por ejemplo, la regla de ∀-introducción sería

$$\frac{\begin{array}{c}(x : A)\\ B(x)\ \mathrm{true}\end{array}}{(\forall x : A)B(x)\ \mathrm{true}} \tag{1.10}$$

De acuerdo con el paradigma de la irrelevancia de las pruebas, esta regla se obtiene a partir de la regla de Π-introducción (1.7) eliminando las pruebas $b(x)$ y $(\lambda x : A)b(x)$. La reinterpretación del resto de reglas es análoga.

Implicación y tipos de funciones (→)

Supongamos ahora que el tipo $B(x) = B$ no depende de x. Entonces podemos definir el tipo de las funciones de A en B como

$$A \to B \equiv (\Pi x : A)B. \tag{1.11}$$

Así, la expresión $\mathrm{Ap}(c, a)$ se corresponde con la aplicación de funciones tal y como la conocemos. Es más, de acuerdo con la correspondencia de Curry-Howard, si vemos los tipos A y B como proposiciones, entonces el tipo $A \to B$ puede interpretarse como la proposición "A implica B" y la aplicación de funciones $\mathrm{Ap}(c, a)$ puede verse como un *modus ponens*.

Suma dependiente (Σ)

Las sumas dependientes, también conocidas como Σ-tipos, se corresponden con la unión "disjunta" de una familia de tipos. La regla de Σ-formación es análoga a la de Π-formación:

$$\frac{A \text{ type} \qquad \begin{matrix}(x : A)\\ B(x) \text{ type}\end{matrix}}{(\Sigma x : A)B(x) \text{ type}} \tag{1.12}$$

Un Σ-tipo puede verse como una generalización del producto cartesiano en el que, dado un par (a, b), el tipo de b puede depender de a. Esto es precisamente lo que refleja la regla de Σ-introducción:

$$\frac{a : A \qquad b : B(a)}{(a, b) : (\Sigma x : A)B(x)} \tag{1.13}$$

Dada una familia de tipos $C(z)$ con índice $z : (\Sigma x : A)B(x)$, la regla de Σ-eliminación nos permite definir una función que para cada término $c : (\Sigma x : A)B(x)$ devuelve un término de tipo $C(c)$:

$$\frac{c : (\Sigma x : A)B(x) \qquad \begin{matrix}(x : A, y : B(x))\\ d(x, y) : C((x, y))\end{matrix}}{\mathrm{E}(c, (x, y)d(x, y)) : C(c)} \tag{1.14}$$

La notación $(x, y)d(x, y)$ quiere decir que las variables x e y son libres en $d(x, y)$. A continuación se describe cómo ejecutar $E(c, (x, y)d(x, y))$. Primero ejecutamos c, dando lugar a un término canónico (a, b) de $(\Sigma x : A)B(x)$, es decir, con $a : A$ y

$b : B(a)$. Ahora sustituimos x e y por a y b en la premisa de la derecha, obteniendo $d(a,b) : C((a,b))$, es decir, ejecutando $d(a,b)$ conseguimos un término canónico e de $C((a,b))$. Como tenemos $c = (a,b) : (\Sigma x : A)B(x)$, concluimos que $C((a,b)) = C(c)$ y por tanto $e : C(c)$, como buscábamos.

De forma análoga a lo que sucede con el producto dependiente y la cuantificación universal, si interpretamos $B(x)$ como una función proposicional con dominio en A, entonces el Σ-tipo $(\Sigma x : A)B(x)$ se corresponde con la **cuantificación existencial** $(\exists x : A)B(x)$. Recordemos que, de acuerdo a la regla de Σ-introducción (1.13), las pruebas canónicas de $(\exists x : A)B(x)$ serían términos de la forma (a,b), donde a es un término de tipo A y b es una prueba de $B(a)$, es decir, una prueba de que $B(x)$ es cierta para a.

Por otro lado, si $B(x) = B$ es un tipo que no depende de x, entonces $(\Sigma x : A)B$ es precisamente el **producto cartesiano** $A \times B$, y si además interpretamos A y B como proposiciones, entonces estamos ante la **conjunción** $A \wedge B$.

La igualdad proposicional

Hasta ahora, la única igualdad entre términos de la que disponemos en la teoría es la correspondiente al juicio $a = b : A$. Sin embargo, al no tratarse de una proposición (un tipo) no podemos manipularla con operadores lógicos. Necesitamos por tanto una proposición $I(A,a,b)$ cuyo significado sea equivalente al del juicio $a = b : A$.

$$\frac{A \text{ type} \qquad a : A \qquad b : A}{I(A,a,b) \text{ type}} \tag{1.15}$$

La forma canónica de obtener un término de $I(A,a,b)$ es teniendo $a = b : A$, como se refleja en la regla de I-introducción:

$$\frac{a = b : A}{\mathrm{r} : I(A,a,b)} \tag{1.16}$$

Observemos que el término canónico r de $I(A,a,b)$ no depende de A, a ni b. Esto se debe a que no importa cuál es el término canónico de $I(A,a,b)$ cuando se tiene $a = b : A$; lo que importa es que tiene uno.

Para las reglas de eliminación e igualdad podríamos seguir el mismo patrón que con el producto y la suma dependientes, adoptando un operador eliminatorio. De esta forma podríamos derivar las siguientes reglas, así que las tomamos directamente

como primitivas. La primera puede considerarse una regla de I-eliminación y la segunda, de I-igualdad:

$$\frac{c \,:\, I(A,a,b)}{a = b \,:\, A} \qquad \frac{c \,:\, I(A,a,b)}{c = r \,:\, I(A,a,b)} \tag{1.17}$$

Otros tipos

Aunque no entraremos en detalles, Martin-Löf incluyó más tipos y constructores en su teoría intuicionista de tipos. Entre ellos están la suma + de dos tipos (o la disyunción de proposiciones), los tipos finitos $\mathbb{N}_n$ de los n primeros naturales, los números naturales, las listas y los conjuntos bien ordenados.

El tipo $\mathbb{N}_0$ es el tipo vacío; no tiene reglas de introducción. Interpretado como una proposición, $\mathbb{N}_0$ es la proposición $\bot$ (falso). Por el contrario, el tipo $\mathbb{N}_1$ tiene un único término canónico ($0 \,:\, \mathbb{N}_1$), y visto como proposición se interpreta como $\top$ (verdadero). Dada una proposición P, se define la negación de P como $\neg P \equiv P \to \bot$.

Las reglas de eliminación de tipos definidos recursivamente (como los números naturales o las listas) se corresponden con los principios de inducción/recursión asociados a esos tipos. Por ejemplo, la regla de eliminación de los números naturales es

$$\frac{\begin{array}{ccc} & & (x \,:\, \mathbb{N}, y \,:\, C(x)) \\ c \,:\, \mathbb{N} \qquad d \,:\, C(0) & & e(x,y) \,:\, C(s(x)) \end{array}}{\mathrm{R}(c,d,(x,y)e(x,y)) \,:\, C(c)} \tag{1.18}$$

El término $\mathrm{R}(c,d,(x,y)e(x,y))$ nos permite definir una función que toma un término $c \,:\, \mathbb{N}$ y devuelve otro término en $C(c)$. Para ello es necesario proporcionar los términos d (la salida de la función en 0) y $(x,y)e(x,y)$ (la salida de la función en un natural $s(x)$ en términos de x y de la salida y de la función en x). Las reglas de $\mathbb{N}$-igualdad reflejan que en efecto este es el significado de d y $(x,y)e(x,y)$:

$$\frac{\begin{array}{cc} & (x \,:\, \mathbb{N}, y \,:\, C(x)) \\ d \,:\, C(0) & e(x,y) \,:\, C(s(x)) \end{array}}{\mathrm{R}(0,d,(x,y)e(x,y)) = d \,:\, C(c)} \tag{1.19}$$

$$\frac{\begin{array}{ccc} & & (x \,:\, \mathbb{N}, y \,:\, C(x)) \\ c \,:\, \mathbb{N} \qquad d \,:\, C(0) & & e(x,y) \,:\, C(s(x)) \end{array}}{\mathrm{R}(s(c),d,(x,y)e(x,y)) = e(c,\mathrm{R}(c,d,(x,y)e(x,y)) \,:\, C(c)} \tag{1.20}$$

Veremos esta idea con más detalle en la sección de Familias Inductivas (ver Sección 1.4).

1.2.5 Construcción de pruebas

En ITT, dada una proposición P, podemos construir una prueba de P mediante la "concatenación" de reglas asociadas a los tipos y constructores que constituyen P. Para ilustrarlo damos a continuación una prueba de una de las implicaciones de la identidad de los indiscernibles de Leibniz: objetos que son iguales satisfacen las mismas propiedades.

Sean A un tipo y $B(x)$ prop. $(x : A)$ una función proposicional. El principio de Leibniz puede enunciarse en ITT como

$$(\forall x : A)\,(\forall y : A)\,I(A,x,y) \to (B(x) \to B(y))\ \text{true}.$$

Para probarlo, asumimos $x : A, y : A$ y $z : I(A,x,y)$. Entonces $x = y : A$ y por tanto $B(x) = B(y)$ por sustitución. Asumiendo ahora $w : B(x)$ obtenemos, por igualdad de tipos, $w : B(y)$. Finalmente, mediante λ-abstracción[3] en w, z, y y x obtenemos la prueba deseada. Expresada formalmente en forma de árbol, la prueba es la siguiente:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(z : I(A,x,y))}{x = y : A} \quad (x : A)}{B(x)\ \text{type}}}{(w : B(x)) \qquad B(x) = B(y)}}{w : B(y)}}{(\lambda w)\, w : B(x) \to B(y)}}{(\lambda z)\,(\lambda w)\, w : I(A,x,y) \to (B(x) \to B(y))}}{(\lambda x)\,(\lambda y)\,(\lambda z)\,(\lambda w)\, w : (\forall x : A)\,(\forall y : A)\,I(A,x,y) \to (B(x) \to B(y))}$$

1.2.6 Universos

Volvamos a la proposición $P \equiv (\forall x : A)\,(\forall y : A)\,I(A,x,y) \to (B(x) \to B(y))$ de la sección anterior. Observemos que P no dice que el principio de Leibniz sea cierto para toda función proposicional $B(x)$ prop. $(x : A)$. Más bien, por cada función proposicional tenemos una proposición $P_{B(x)}$, así que, técnicamente, cada vez que quisiéramos usar el principio de Leibniz tendríamos que probarlo independientemente para cada función proposicional, aunque la prueba sea la misma.

[3] Recordemos que la λ-abstracción es la regla de introducción del producto dependiente (ecuación 1.7) y que el operador $\to$ se define en 1.11 como un caso particular del producto dependiente.

Esto se solucionaría si pudiéramos generalizar el principio de Leibniz abstrayendo también sobre una variable que representase a cualquier función proposicional. Esta nueva proposición completamente general sería algo así:

$$(\forall A : \text{Type})\,(\forall B : A \to \text{Type})\,(\forall x : A)\,(\forall y : A)\,I(A,x,y) \to (B(x) \to B(y)), \quad (*)$$

Sin embargo, para enunciar esta versión general necesitaríamos un tipo universal Type de forma que el juicio $A : \text{Type}$ fuera equivalente a A type; algo similar a lo que sucede con el tipo $I(A,a,b)$ y el juicio $a = b : A$. Es decir, Type sería el tipo de todos los tipos (o de todas las proposiciones). Sin embargo, dentro de la semántica descrita hasta ahora en la ITT, esto es imposible. Recordemos que en ITT un tipo se define diciendo de forma explícita cómo son sus términos canónicos. En el caso de un hipotético tipo Type, sus términos canónicos serían los tipos, pero no es posible dar reglas para todas la formas posibles de construir tipos, pues siempre podremos describir nuevos tipos, por ejemplo, el tipo de todos los tipos.

En otras palabras, la teoría intuicionista de Martin-Löf es de carácter **predicativo**: no se permite la autorreferencia a la hora de enunciar tipos o proposiciones. Esto quiere decir que no está permitido hacer cuantificación universal sobre todos los tipos o proposiciones. Más generalmente, un universo se dice impredicativo (no predicativo) si cuando hacemos cuantificación universal sobre todos los tipos del universo obtenemos un tipo de ese universo. Veremos esta noción con más detalle a lo largo del trabajo, especialmente en el segundo capítulo. Nótese que la proposición, llamémosla T, definida en $(*)$ es impredicativa: si T es un tipo, es decir, $T : \text{Type}$, entonces podemos sustituir A por T en T, obteniendo una definición autorreferencial.

En definitiva, no podemos tener un "tipo de todos los tipos" en ITT. Sin embargo, en numerosas partes de las matemáticas, por ejemplo, en teoría de categorías, resulta muy útil e incluso indispensable poder trabajar con conjuntos de conjuntos, cuantificación sobre conjuntos... La idea para solventar el problema de la autorreferencia es definir un tipo U, llamado universo, como el menor tipo cerrado con respecto a las operaciones básicas de la teoría. Las siguientes reglas definen el universo U junto con una familia de tipos $T(x)\,(x : U)$.

U-formación

$$U \text{ type} \qquad \frac{a : U}{T(a) \text{ type}}$$

U-introducción

$$\frac{a : U \qquad b(x) : U}{\pi(a,(x)b(x)) : U}(x : T(a)) \qquad \frac{a : U \qquad b(x) : U}{T(\pi(a,(x)b(x))) = (\Pi x : T(a))T(b(x))}(x : T(a))$$

$$\frac{a : U \qquad b(x) : U}{\sigma(a,(x)b(x)) : U}(x : T(a)) \qquad \frac{a : U \qquad b(x) : U}{T(\sigma(a,(x)b(x))) = (\Sigma x : T(a))T(b(x))}(x : T(a))$$

$$\frac{a : U \qquad b : T(a) \qquad c : T(b)}{i(a,b,c) : U} \qquad \frac{a : U \qquad b : T(a) \qquad c : T(b)}{T(i(a,b,c)) = I(T(a),b,c)}$$

$$n_0 : U \quad n_1 : U \quad \ldots \qquad T(n_0) = \mathbb{N}_0 \quad T(n_1) = \mathbb{N}_1 \quad \ldots$$

$$n : U \qquad T(n) = \mathbb{N}$$

$$\vdots \qquad \vdots$$

Es decir, U es el tipo cuyos términos canónicos son los *nombres* de los tipos que se pueden formar usando las operaciones básicas y T es una función de "decodificación" que envía el nombre $a : U$ de un tipo al propio tipo $T(a)$.

La inclusión del universo U en la teoría puede verse como un truco para poder tratar a los tipos como términos. Gracias a esto, podemos enunciar el principio de Leibniz en toda su generalidad sin saltarnos las reglas de la teoría:

$$(\forall a : U)\,(\forall b : T(a) \to U)\,(\forall x : T(a))\,(\forall y : T(a))\,I(T(a),x,y) \to (T(b(x)) \to T(b(y)))$$

Para mostrar la utilidad del universo U construimos a continuación una prueba del cuarto axioma de Peano:

$$(\forall x : \mathbb{N})\,\neg I(\mathbb{N},0,s(x))\ \text{true}$$

Supongamos $x : \mathbb{N}$. Recordemos que, por definición $\neg I(\mathbb{N},0,s(x))$ es $I(\mathbb{N},0,s(x)) \to \mathbb{N}_0$, así que debemos suponer $y : I(\mathbb{N},0,s(x))$. Por I-eliminación (1.17) tenemos entonces que $0 = s(x) : \mathbb{N}$. Por U-introducción sabemos que $n_0 : U$ y $n_1 : U$. Usando $\mathbb{N}$-eliminación (1.18) podemos definir $f(a) \equiv \mathrm{R}(a, n_0, (x,y)n_1)$ para cada $a : \mathbb{N}$. Por las reglas de $\mathbb{N}$-igualdad (1.19 y 1.20) se tiene que $f(0) = n_0 : U$ y $f(s(a)) = n_1 : U$ para cualquier $a : \mathbb{N}$. Ahora bien, como $0 = s(x) : \mathbb{N}$, tenemos que $f(0) = f(s(x)) : U$, es

decir, $n_0 = n_1 : U$. Aplicando T obtenemos $T(n_0) = T(n_1)$, es decir, $\mathbb{N}_0 = \mathbb{N}_1$. Como $0 : \mathbb{N}_1$, entonces $0 : \mathbb{N}_0$. Abstrayendo las variables x e y, la prueba que buscamos es

$$(\lambda x)\,(\lambda y)\,0 : (\forall x : \mathbb{N})\,\neg I(\mathbb{N}, 0, s(x))$$

Nótese que, aunque es evidente que no puede probarse $0 = s(a) : \mathbb{N}$ en la teoría, probar $\neg I(\mathbb{N}, 0, s(a))$ no parece posible sin tratar a los tipos $\mathbb{N}_0$ y $\mathbb{N}_1$ como términos por medio del universo U y la función T.

1.3 El Cálculo de Construcciones

El Cálculo de Construcciones (CoC) es una teoría de tipos dependientes desarrollada por Thierry Coquand y Gérard Huet [1]. A diferencia de la ITT, el CoC se sustenta en un universo impredicativo.

Mientras que Martin-Löf se centra en dar una justificación semántica a cada término, juicio o regla que define, Coquand y Huet tienen como prioridad la potencia computacional de su teoría, por lo que hacen especial énfasis en el álgebra de términos sobre la que se construye.

1.3.1 La sintaxis abstracta de los términos

Los términos de la teoría son esencialmente λ-expresiones, donde las variables están tipadas y sus tipos son también términos de la misma naturaleza (a diferencia de lo que sucede en ITT, donde los tipos no son términos). La sintaxis cuenta con dos operadores de ligadura para formar términos: $(\lambda x : N)M$ para representar λ-abstracciones y $[x : A]M$ para representar productos. Escribimos (MN) para la aplicación del término M al término N.

Finalmente, el álgebra de términos cuenta con una constante $*$ que juega el papel del universo de todos los tipos. De acuerdo con la correspondencia de Curry-Howard, $*$ debe pensarse simultáneamente como el tipo de todos los tipos y el tipo de todas las proposiciones. Sin embargo, no hay circularidad: $*$ no es de tipo $*$.

Los términos que consisten únicamente en productos sobre $*$ se denominan *contextos*. Los contextos son los tipos formados por las proposiciones y las funciones

proposicionales. El resto de términos los llamamos *objetos*. Veremos que hay dos categorías de tipos: los tipos usuales (son objetos de tipo $*$) y los tipos de tipos (son contextos), cuyo papel principal es permitir polimorfismos[4].

A continuación se define formalmente el conjunto de términos del cálculo.

Definición 1.1. Se define el conjunto de términos como

$$\Lambda = \Lambda_1 \cup \Lambda_0,$$

donde Λ_1 es el conjunto de objetos

$$\Lambda_1 = \bigcup_{n \geq 0} \Lambda_1^n$$

y Λ_0 es el conjunto de contextos

$$\Lambda_0 = \bigcup_{n \geq 0} \Lambda_0^n,$$

donde Λ_0^n es el conjunto de contextos con n o menos variables libres, generado por las reglas inductivas

$$\begin{array}{rl} * \in \Lambda_0^n & \textit{universo} \\ [x : M]N \in \Lambda_0^n \quad \text{si} \quad M \in \Lambda^n, N \in \Lambda_0^{n+1} & \textit{cuantificación} \end{array}$$

y Λ_1^n es el conjunto de objetos con n o menos variables libres, generado por las reglas inductivas[5]

$$\begin{array}{rlll} k \in \Lambda_1^n & \text{si} & 1 \leq k \leq n & \textit{variables} \\ [x : M]N \in \Lambda_1^n & \text{si} & M \in \Lambda^n, N \in \Lambda_1^{n+1} & \textit{producto} \\ (\lambda x : M)N \in \Lambda_1^n & \text{si} & M \in \Lambda^n, N \in \Lambda_1^{n+1} & \textit{abstracción} \\ (M\,N) \in \Lambda_1^n & \text{si} & M, N \in \Lambda_1^n & \textit{aplicación} \end{array}$$

y

$$\Lambda^n = \Lambda_0^n \cup \Lambda_1^n.$$

Los términos *cerrados* son los elementos de Λ^0, legales en el contexto vacío $*$.

[4]Un polimorfismo es una aplicación que puede actuar sobre elementos que viven en dominios distintos. Por ejemplo, tiene sentido aplicar la función identidad a elementos de cualquier conjunto.

[5]La numeración de las variables sigue el convenio de los índices de De Bruijn.

Aunque para hacer más legible las expresiones usaremos letras para denotar a las variables ligadas que aparecen en los términos de la forma $[x : M]N$ y $(\lambda x : M)N$, en realidad en los términos del Cálculo de Construcciones se reemplazan estas variables por *índices de De Bruijn*.

Los índices de De Bruijn funcionan de la siguiente forma: el índice i en un cierto término hace referencia a la i-ésima variable (empezando a contar por la derecha) que se declara en el término antes de la aparición del índice. Por ejemplo, la expresión $[x : M][y : N]((x\,y)\,x)$ es en realidad una representación de $[M][N]((2\,1)\,2)$, y la expresión $(\lambda A : *)(\lambda x : A)x$ (la versión polimórfica de la función identidad) es una representación de $(\lambda *)(\lambda 1)1$. En [6] puede consultarse con más detalle el funcionamiento de los índices de De Bruijn, aunque no es necesario para entender lo sucesivo.

Por lo general se usarán las letras Γ, Δ para los contextos y las letras A, B, M, N, P, Q para términos cualesquiera (es decir, objetos y contextos).

Los objetos servirán para denotar proposiciones así como elementos y términos funcionales. Los elementos y términos funcionales pueden interpretarse como pruebas de sus tipos, de acuerdo con la correspondencia de Curry-Howard. En concreto, el término $[x : M]P$ representa tanto a la cuantificación universal $(\forall x : M)P$ como al producto dependiente $(\Pi x : M)P$[6]. Por un lado, podemos interpretar $(\lambda x : M)N$ como una prueba de $[x : M]P$ si N es una prueba de P bajo la hipótesis M, luego $[x : M]P$ puede verse como la cuantificación universal $(\forall x : M)P$. Por otro lado, también podemos ver $(\lambda x : M)N$ como una función que envía cada x de tipo M en N de tipo P (pudiendo P depender de x), es decir, como un elemento del producto dependiente $(\Pi x : M)P$. De igual forma, la aplicación $(M\,N)$ puede pensarse o bien como un *modus ponens*, o bien como la aplicación de una función a un elemento de su dominio.

Antes de describir las reglas de inferencia del sistema, vamos introducir un poco de notación.

Los contextos son de la forma $\Gamma = [x_n : M_n] \cdots [x_1 : M_1]*$, así que pueden ser tratados como listas cuyo constructor es la cuantificación, siendo $*$ la lista vacía. Así, la longitud $|\Gamma|$ de un contexto es la longitud de la lista correspondiente. De igual forma, dados dos contextos $\Gamma \in \Lambda_0^m$ y $\Delta \in \Lambda_0^n$ con $n \leq |\Gamma|$, podemos definir la concatenación de ambos, $\Gamma; \Delta$, como el contexto en Λ_0^m dado por la concatenación de las correspondien-

[6] En esta observación los tipos $(\forall x : M)P$ y $(\Pi x : M)P$ no hacen referencia a los tipos definidos en ITT, sino a las nociones subyacentes de cuantificación universal y producto dependiente.

tes listas. Observemos que el contexto resultante de concatenar Γ y Δ (en ese orden) tiene tantas variables libres como Γ, pues en la concatenación las variables libres de Δ pasan a hacer referencia a variables introducidas por Γ.

Si $\Gamma \in \Lambda_0^0$ es un contexto cerrado y $M \in \Lambda^n$ con $n \leq |\Gamma|$, usamos $\Gamma[x : M]$ para denotar el contexto cerrado $\Gamma; [x : M]*$. Si Γ es el contexto vacío escribiremos directamente $[x : M]$. Así, todo contexto cerrado Γ de longitud n puede ser escrito como

$$\Gamma = [x_n : \Gamma_n] \cdots [x_1 : \Gamma_1],$$

con $\Gamma_i \in \Lambda^{n-i}$ (la numeración de las variables coincide con el convenio de los índices de De Bruijn).

Dado un contexto cerrado $\Gamma = [x_n : \Gamma_n] \cdots [x_1 : \Gamma_1]$, observemos que cada término $\Gamma_k \in \Lambda^{n-k}$ es legal dentro del contexto cerrado $\Delta = [x_n : \Gamma_n] \cdots [x_{k+1} : \Gamma_{k+1}]$. Es decir, para toda variable libre i que aparece en Γ_k, se cumple $1 \leq i \leq n - k$. Necesitamos un término como Γ_k pero visto en el contexto total Γ en lugar de en el contexto parcial Δ. Para ello, definimos Γ/k como el término resultante de sumar k a todas las variables libres de Γ_k.

1.3.2 El sistema de reglas de inferencia

El Cálculo de Contrucciones cuenta, en su forma más básica, con dos tipos de juicio:

(1) $\Gamma \vdash \Delta$

(2) $\Gamma \vdash M : N$

Intuitivamente, el consecuente $\Gamma \vdash \Delta$ significa que el contexto Δ es válido en el contexto (válido) Γ, es decir, que las variables y tipos que introduce Δ tienen sentido en vista de Γ. Por otro lado, $\Gamma \vdash M : N$ significa que, en el contexto (válido) Γ, M es un término bien construido de tipo N. En CoC los contextos son términos que son tipos, pero que no son de ningún tipo.

El segundo juicio también tiene una interpretación bajo el paradigma de las proposiciones vistas como tipos. Si N es un objeto, entonces el segundo consecuente significa que, en el contexto Γ, M es una prueba bien construida de la proposición N. Si N es un contexto, entonces el consecuente significa que, en el contexto Γ, M es una función proposicional bien construida y con argumentos declarados en N.

En particular, $\Gamma \vdash *$ significa que el contexto Γ es válido. Abreviaremos $* \ \vdash \Delta$ y $* \ \vdash M \ : \ N$ como $\vdash \Delta$ y $\vdash M \ : \ N$, respectivamente.

Primero damos las reglas para la construcción de contextos válidos:

$$\vdash * \qquad \frac{\Gamma \vdash \Delta}{\Gamma[x \ : \ \Delta] \vdash *} \qquad \frac{\Gamma \vdash M \ : \ *}{\Gamma[x \ : \ M] \vdash *} \tag{1.21}$$

Las siguientes reglas corresponden a la formación de productos:

$$\frac{\Gamma[x \ : \ M] \vdash \Delta}{\Gamma \vdash [x \ : \ M]\Delta} \qquad \frac{\Gamma[x \ : \ M_1] \vdash M_2 \ : \ *}{\Gamma \vdash [x \ : \ M_1]M_2 \ : \ *} \tag{1.22}$$

En la segunda regla se encuentra oculta una de las principales diferencias de esta teoría con la teoría de tipos intuicionista de Martin-Löf: el Cálculo de Construcciones es una teoría impredicativa. Aquí, el tipo $*$ de todas las proposiciones es cerrado con respecto al producto. Por ejemplo, de acuerdo con la segunda regla, el término $[A \ : \ *]A$ es de tipo $*$, de hecho, este término es la definición del tipo vacío (o de la proposición $\bot$) en CoC.

Finalmente presentamos las reglas correspondientes a las variables, la λ-abstracción y la aplicación, respectivamente:

$$\frac{\dfrac{(k \leq |\Gamma|)}{\Gamma \vdash *}}{\Gamma \vdash k \ : \ \Gamma/k} \qquad \frac{\Gamma[x \ : \ M_1] \vdash M_2 \ : \ P}{\Gamma \vdash (\lambda x \ : \ M_1)M_2 \ : \ [x \ : \ M_1]P} \tag{1.23}$$

$$\frac{\Gamma \vdash M \ : \ [x \ : \ P]Q \qquad \Gamma \vdash N \ : \ P}{\Gamma \vdash (M \, N) \ : \ [N/x]Q} \tag{1.24}$$

En la regla 1.24 $[N/x]Q$ denota al término resultante de sustituir la variable x en Q por el término N.

Observemos que en el Cálculo de Construcciones, a diferencia de lo que sucede en la ITT, no definimos los tipos desde cero a partir de una serie de juicios y juicios hipotéticos, sino que partimos de un tipo universal $*$ junto con la λ-abstracción, la cuantificación universal y la aplicación. La idea es que a partir de estas operaciones, y gracias a que el tipo $*$ es impredicativo, podemos definir en CoC todos los tipos (y más) que se definen en la teoría de Martin-Löf. Por ejemplo, el conjunto de los números naturales puede definirse como

$$\mathbb{N} \equiv [A \ : \ *](A \to A) \to (A \to A),$$

donde $A \to A$ denota al término $[x : A]A$. Según la definición anterior, los números naturales pueden verse como los términos que, dado un tipo A cualquiera, una función s de A en A y un término z de A, devuelven un término de A. En efecto, podemos definir

$$\begin{aligned} 0 &\equiv (\lambda A : *)(\lambda s : A \to A)(\lambda z : A)\ z \\ 1 &\equiv (\lambda A : *)(\lambda s : A \to A)(\lambda z : A)\ s(z) \\ 2 &\equiv (\lambda A : *)(\lambda s : A \to A)(\lambda z : A)\ s(s(z)) \\ &\dots \end{aligned}$$

1.3.3 Las reglas de conversión

La λ-abstracción se puede usar para construir funciones proposicionales. Por ejemplo, si queremos usar $A \to B$ para denotar al término $[x : A]B$ de tipo $*$ (suponiendo $A : *$ y $B : *$), podríamos definir $(\lambda A : *)(\lambda B : *)[x : A]B$ de tipo $[A : *][B : *]*$. Ahora bien, con las reglas que tenemos hasta ahora, un término t de tipo $\to M\,N$ no es de tipo $[x : M]N$, sino de tipo $((\lambda A : *)(\lambda B : *)[x : A]B)M\,N$.

Necesitamos por tanto reglas que permitan reemplazar un término $((\lambda x : A)M\,N)$ por $[N/x]M$. Esto se conoce como β-conversión. Para ello introducimos un nuevo juicio $M \cong N$ entre términos.

Definición 1.2. La relación $\cong$ es la menor relación de equivalencia entre proposiciones y contextos generada por las siguientes reglas:

$$\frac{\Gamma \vdash P_1 \cong P_2 \qquad \Gamma[x : P_1] \vdash M_1 \cong M_2}{\Gamma \vdash [x : P_1]M_1 \cong [x : P_2]M_2} \tag{1.25}$$

$$\frac{\Gamma \vdash P_1 \cong P_2 \qquad \Gamma[x : P_1] \vdash M_1 \cong M_2 \qquad \Gamma[x : P_1] \vdash M_1 : N}{\Gamma \vdash (\lambda x : P_1)M_1 \cong (\lambda x : P_2)M_2} \tag{1.26}$$

$$\frac{\Gamma \vdash (MN) : P \qquad \Gamma \vdash M \cong M_1 \qquad \Gamma \vdash N \cong N_1}{\Gamma \vdash (M\,N) \cong (M_1 N_1)} \tag{1.27}$$

$$\frac{\Gamma[x : A] \vdash M : P \qquad \Gamma \vdash N : A}{((\lambda x : A)M\,N) \cong [N/x]M} \tag{1.28}$$

La última regla (1.28) es la que corresponde a la β-conversión.

Si solo permitimos conversión a nivel de tipos, entonces obtenemos el cálculo de construcciones restringido. Si por el contrario permitimos conversión sobre términos cualesquiera, obtenemos el cálculo de construcciones completo (o cálculo de construcciones, a secas).

1.3.4 Algunas propiedades del cálculo

En esta sección se usa la letra E para referirse a un juicio arbitrario (puede ser de la forma Δ, $M : P$, o $M \cong N$). A continuación se enuncian algunas propiedades del cálculo que se pueden probar por inducción en las reglas de derivación:

Lema 1.3. *Si* $\Gamma \vdash E$, *entonces* $\Gamma \vdash *$ *y, más concretamente, cada derivación de* $\Gamma \vdash E$ *contiene una subderivación de* $\Delta \vdash *$ *para todo prefijo* Δ *de* Γ.

Lema 1.4. *Si* $\Gamma[x : P]\Delta \vdash E$ *y* $\Gamma \vdash N : P$, *entonces* $\Gamma[N/x]\Delta \vdash [N/x]E$.

Lema 1.5. *Si* $\Gamma \vdash M : P$ *y* $\Gamma \vdash M \cong N$, *entonces* $\Gamma \vdash N : P$.

Lema 1.6. *Si* $\Gamma \vdash M : N$ *y* N *es un objeto, entonces* $\Gamma \vdash N : *$, *y si* $\Gamma \vdash M : \Delta$ *y* Δ *es un contexto, entonces* $\Gamma \vdash \Delta$.

El lema anterior quiere decir, en particular, que los tipos son, o bien proposiciones (en caso de ser objetos), o bien contextos. El siguiente lema nos dice que los tipos son únicos salvo conversión:

Lema 1.7. *Si* $\Gamma \vdash M : N_1$ *y* $\Gamma \vdash M : N_2$, *entonces* $\Gamma \vdash N_1 \cong N_2$.

Para el cálculo restringido se puede probar directamente lo siguiente:

Proposición 1.8. *El juicio* $\Gamma \vdash E$ *es decidible en el cálculo de construcciones restringido.*

En el Cálculo de Construcciones es importante la noción de *reducción* $\triangleright$ asociada a la noción de conversión $\cong$. La relación $\triangleright$ se define exactamente igual que $\cong$, pero sin la simetría, de forma que si tenemos $M \triangleright N$ significa que N se obtiene reduciendo M. Puede probarse gracias a la noción de reducción que los términos del CoC admiten una forma normal. La reducción también se conoce como *normalización*.

1.3.5 *Stripping*

El *stripping* consiste en extraer de una prueba un λ-término puro, esencialmente eliminando tipos y proposiciones y quedándonos solo con el algoritmo subyacente. En esta sección se explica cómo llevar a cabo *stripping* sobre pruebas construidas dentro del Cálculo de Construcciones, reforzando la idea de que, en esta teoría, todo término cuyo tipo es un objeto alberga un contenido computacional.

Primero definimos la sintaxis del λ-cálculo no tipado:

Definición 1.9. Definimos el conjunto λ^n de las λ-expresiones con n variables libres mediante las siguientes reglas:

$$\begin{array}{rlll} k \in \lambda^n & \text{si} & 1 \leq k \leq n & \textit{variables} \\ \lambda x \cdot N \in \lambda^n & \text{si} & N \in \lambda^{n+1} & \textit{abstracción} \\ (M\,N) \in \lambda^n & \text{si} & M, N \in \lambda^n & \textit{aplicación} \end{array}$$

Supongamos que queremos extraer el algoritmo de la prueba M de N en el consecuente $\Gamma \vdash M : N$.

El primer paso es quedarnos con los cuantificadores de Γ que actúan sobre objetos e ignorar los que actúan sobre contextos, pues son los primeros los que consideraremos como variables libres de las λ-expresiones extraídas. Para ello definimos las siguientes nociones asociadas a un contexto:

Definición 1.10. El número de parámetros o aridad α_Γ y la inyección canónica $j_\Gamma : \alpha_\Gamma \to |\Gamma|$ de un contexto Γ están determinadas por las siguientes reglas inductivas[7]:

- Si $\Gamma = *$, entonces $\alpha_\Gamma = 0$ y $j_\Gamma = \mathrm{Id}_0$
- Si $\Gamma = \Delta[x : M]$, entonces
  - si M es un contexto, $\alpha_\Gamma = \alpha_\Delta$ y $j_\Gamma(k) = j_\Delta(k) + 1$,
  - si M es un objeto, $\alpha_\Gamma = \alpha_\Delta + 1$, $j_\Gamma(1) = 1$ y $j_\Gamma(k+1) = j_\Delta(k) + 1$.

Es decir, la aridad α_Γ es el número de cuantificadores sobre objetos que hay en Γ y, para cada $k \in \{1, \dots, \alpha_\Gamma\}$, $j_\Gamma(k)$ es la posición en Γ que ocupa el k-ésimo cuantificador sobre objetos de Γ (contando de acuerdo al convenio de los índices de De Bruijn, es decir, de derecha a izquierda).

Ya podemos definir el algoritmo extraído de una prueba:

Definición 1.11. Si $\Gamma \vdash M : N$ y N es un objeto (y por tanto una proposición), definimos el algoritmo extraído $\upsilon_\Gamma(M) \in \lambda^{\alpha_\Gamma}$ por inducción en M:

1. Si $M = k$, tomamos $\upsilon_\Gamma(M) = j_\Gamma^{-1}(k)$.
2. Si $M = (M_1\,M_2)$, sabemos que $\Gamma \vdash M_2 : P_2$ y $\Gamma \vdash M_1 : P_1$ con $P_1 = [x : P_2]N$ (P_1 es un objeto). Si P_2 también es un objeto, entonces tomamos $\upsilon_\Gamma(M) = (\upsilon_\Gamma(M_1)\,\upsilon_\Gamma(M_2))$.

[7] En esta definición estamos identificando n con $\{1, \dots, n\}$. En particular, $0 = \{\}$

Si P_2 es un contexto, entonces tomamos $v_\Gamma(M) = v_\Gamma(M_1)$ (simplemente olvidamos toda la información relativa a los tipos (las proposiciones), que pueden verse ahora como comentarios del programa extraído).

3. Si $M = (\lambda x : P)A$, entonces sabemos $\Delta \vdash A : N'$, donde $\Delta = \Gamma[x : P]$ y N' es un objeto tal que $N = [x : P]N'$. Si P es un objeto, entonces $v_\Gamma(M) = \lambda x \cdot v_\Delta(A)$. Si P es un contexto, entonces $v_\Gamma(M) = v_\Delta(A)$.

1.3.6 Definición de algunos tipos en CoC

La estructura de reglas de inferencia del Cálculo de Construcciones es fija, por lo tanto, no pueden definirse nuevos tipos mediante reglas, sino declarando constantes. Así las "reglas" asociadas a los tipos definidos (introducción, eliminación...) también son constantes que han de ser probadas mediante la construcción de términos del tipo adecuado. La reglas de inferencia presentadas en las secciones 1.3.2 y 1.3.3 son la herramienta para verificar el tipo de los términos construidos.

En esta sección estudiamos algunos ejemplos de definiciones de tipos en CoC. Los ejemplos están sacados de [7].

Producto cartesiano o conjunción

La mayoría de tipos y constructores de tipos en el Cálculo de Construcciones se definen formalizando su significado operacional, es decir, describiendo *cómo se usa* el tipo, en lugar de cómo son los términos del tipo.

La conjunción $\wedge$ de dos proposiciones es un ejemplo prototípico de esta filosofía. Dadas dos proposiciones P y Q, ¿cómo se usa la conjunción $P \wedge Q$? Sea R una proposición cualquiera. Podemos usar una prueba de $P \wedge Q$ para probar R si tenemos una prueba de que P y Q juntas implican R, es decir, una prueba de $P \to Q \to R$. Dicho de otra forma, el operador $\wedge$ se define directamente como su regla de eliminación:

$$\wedge \equiv [P : *][Q : *][R : *](P \to Q \to R) \to R$$

Escribiremos $P \wedge Q$ en lugar de $\wedge\, P\, Q$. La "regla" de introducción de la conjunción viene dada por la proposición

$$\wedge_{\text{intro}} \equiv [P : *][Q : *]P \to Q \to P \wedge Q,$$

cuya prueba (o definición) es

$$\text{pair} \equiv (\lambda P : *)(\lambda Q : *)(\lambda p : P)(\lambda q : Q)(\lambda R : *)(\lambda f : P \to Q \to R)(f\ p\ q)$$

Producto dependiente o cuantificación universal

La definición de la cuantificación universal es atípica, pues el Cálculo de Construcciones parte desde el principio de la operación de ligadura $[x : M]N$, que en sí misma ya es una cuantificación universal (o producto) sobre el tipo M. La siguiente definición del producto dependiente simplemente restringe el término M a un tipo A de tipo $*$ y el término N a la proposición $P\,x$, donde $P : A \to *$ es un predicado sobre A (o una familia de tipos indexada en A):

$$\forall \equiv [A : *][P : A \to *][x : A](P\,x)$$

Escribiremos $(\forall x : A)(P\,x)$ en lugar de $\forall A\,P$. La regla de eliminación corresponde a la proposición

$$\forall_{\text{elim}} \equiv [A : *][P : A \to *][a : A]((\forall x : A)(P\,x)) \to (P\,a),$$

cuya prueba viene dada por el término

$$\text{inst} \equiv (\lambda A : *)(\lambda P : A \to *)(\lambda a : A)(\lambda h : ((\forall x : A)(P\,x)))(h\,a)$$

La regla de introducción viene dada por la proposición

$$\forall_{\text{intro}} \equiv [A : *][P : A \to *][B : *]([x : A](B \to (P\,x))) \to B \to (\forall x : A)(P\,x),$$

cuya demostración es

$$\text{gen} \equiv (\lambda A : *)(\lambda P : A \to *)(\lambda B : *)(\lambda h : [x : A](B \to (P\,x)))(\lambda b : B)(\lambda x : A)(h\,x\,b)$$

Observemos que esta regla de introducción es más general[8] que la descrita por Martin-Löf para el tipo $(\Pi x : A)P(x)$ (ver Ecuación 1.7).

[8] Aunque se puede derivar en ITT una regla análoga a $\forall_{\text{intro}}$ a partir de las reglas definidas por Martin-Löf para el producto dependiente (ver ecuaciones de 1.6 a 1.9).

Cuantificación existencial

Como en el caso de la conjunción, la cuantificación existencial (o los Σ-tipos) se define a partir de su regla de eliminación. Dada una proposición Q, podemos usar una prueba de $(\exists x : A)(P\,x)$ para probar Q si tenemos una prueba de que para todo x en A se cumple que $P\,x$ implica Q, es decir, si tenemos una prueba de $[x : A]((P\,x) \to Q)$.

$$\exists \equiv [A : *][P : A \to *][Q : *]([x : A]((P\,x) \to Q)) \to Q$$

Escribimos $(\exists x : A)(P\,x)$ en lugar de $\exists\,A\,P$. La regla de introducción se corresponde con la definida por Martin-Löf:

$$\exists_{\text{intro}} \equiv [A : *][P : A \to *][a : A](P\,a) \to (\exists x : A)(P\,x)$$

La prueba corresponde a dar un término de A que cumple P:

$$\begin{aligned}\text{exists} \equiv (\lambda A : *)(\lambda P : A \to *)(\lambda a : A)(\lambda p : P\,a)(\lambda Q : *)\\ (\lambda h : [x : A]((P\,x) \to Q))(h\,a\,p)\end{aligned}$$

Las proyecciones son un caso particular de la regla de eliminación que nos permiten extraer un elemento $a : A$ y una prueba de $P\,a$ a partir de una prueba de $(\exists x : A)(P\,x)$.

$$\exists_{\text{left}} \equiv [A : *][P : A \to *]((\exists x : A)(P\,x)) \to A$$

$$\text{witness} \equiv (\lambda A : *)(\lambda P : A \to *)(\lambda h : (\exists x : A)(P\,x))(h\,A\,(\lambda a : A)(\lambda p : (P\,a))a)$$

Sin embargo, la cuantificación existencial del Cálculo de Construcciones es más débil que la de la teoría intuicionista de Martin-Löf, pues permite extraer un elemento $a : A$ (proyección izquierda), pero no una prueba $p : P\,a$ (proyección derecha).

Igualdad

En el Cálculo de Construcciones la igualdad se define mediante el principio de Leibniz: dos objetos son iguales si satisfacen las mismas propiedades.

$$= \equiv [A : *][x : A][y : A][P : A \to *](P\,x) \to (P\,y)$$

1.4 Familias inductivas

La reglas de inferencia de la teoría de tipos de Martin-Löf pueden separarse en tres categorías: reglas generales (reflexividad, transitividad y simetría de la igualdad, reglas de sustitución, reglas de conversión...), reglas para la formación de tipos básicos y reglas para los universos.

El trabajo de Peter Dybjer en [2] tiene como objetivo extender la teoría de Martin-Löf con la posibilidad de definir nuevas reglas para la formación de tipos. Según Dybjer, existen esencialmente dos opciones para conseguir esto:

(1) Utilizar una construcción de propósito general, como una cuantificación impredicativa. Esto es precisamente lo que hacen Coquand y Huet en el Cálculo de Construcciones.

(2) Describir un esquema para la construcción de extensiones correctas.

Dybjer opta por la segunda opción para la teoría de las Familias Inductivas.

El esquema propuesto por Dybjer da lugar a inducción generalizada, iterada y estrictamente positiva. Sin embargo, esto no cubre todas las formas de inducción relevantes en el marco intuicionista. Por ejemplo, en la regla de U-introducción

$$\frac{a : U \qquad \begin{array}{c}(x : T(a)) \\ b(x) : U\end{array}}{\pi(a, (x)b(x)) : U}$$

de la sección 1.2.6 la función T aparece de forma negativa. En el capítulo 2 veremos con detalle la definición de inducción estrictamente positiva.

La teoría de las Familias Inductivas es muy cercana al Cálculo de Construcciones, que fue desarrollado independientemente. La estructura combinatorial de ambas teorías es esencialemente la misma. La diferencia principal es que CoC se desarrolla a partir de un sistema impredicativo, así que la base de la estructura de tipos es distinta.

1.4.1 La teoría de tipos lógicos

En esta sección se presenta la infraestructura sobre la que se describirán los esquemas de inducción y recursión para las familias inductivas.

La teoría de tipos lógicos es un λ-cálculo con β y η conversión. Los tipos de funciones dependientes se escriben $(a : \sigma)\tau[a]$, donde $\tau[a]$ significa que el tipo τ puede depender de a. La teoría cuenta con un tipo base especial, llamado *set*, y con una regla que dice que todo conjunto (es decir, todo objeto de tipo *set*) es también un tipo base. Llamamos *s-tipos* o *tipos puramente funcionales* a los conjuntos y a los tipos de funciones de un s-tipo en otro s-tipo. La aplicación se escribe $p(q)$ y la abstracción, $(a)p[a]$. La teoría está formulada *à la Martin-Löf*, con cuatro tipos de juicio: σ type, $p : \sigma$, $\sigma = \tau$ y $p = q : \sigma$. Los juicios pueden hacerse bajo una lista de hipótesis (o contexto) de la forma $a_1 : \sigma_1$, $a_2 : \sigma_2[a_1], \dots, a_n : \sigma_n[a_1, \dots, a_{n-1}]$. Usaremos $(a :: \sigma)$ como abreviación de $(a_1 : \sigma_1) \cdots (a_n : \sigma_n[a_1, \dots, a_{n-1}])$ y nos referiremos a σ como una *secuencia de tipos*.

Además de las reglas para manipular hipótesis y las reglas de sustitución, consideramos un sistema de inferencia con las siguientes reglas.

Reglas de formación de tipos

$$set \text{ type} \qquad \frac{A : set}{A \text{ type}} \qquad \frac{\alpha \text{ type} \qquad \begin{matrix}(a : \alpha)\\ \tau[a] \text{ type}\end{matrix}}{(a : \alpha)\tau[a] \text{ type}} \tag{1.29}$$

Reglas de formación de objetos

$$\frac{\begin{matrix}(a : \sigma)\\ p[a] : \tau[a]\end{matrix}}{(a)p[a] : (a : \sigma)\tau[a]} \qquad \frac{p : (a : \sigma)\tau[a] \qquad q : \sigma}{p(q) : \tau[q]} \tag{1.30}$$

Reglas de igualdad (β-conversión y η-conversión)

$$\frac{\begin{matrix}(a : \sigma)\\ q : \sigma \qquad p[a] : \tau[a]\end{matrix}}{((a)p[a])q = p(q) : \tau[q]} \qquad \frac{p : (a : \sigma)\tau[a]}{p = (a)(p(a)) : (a : \sigma)\tau[a]} \tag{1.31}$$

También se incluyen aquí las reglas que hacen que la igualdad (entre tipos y entre objetos) sea una relación de equivalencia, así como las reglas de sustitución de iguales por iguales.

Tal y como sucede en el Cálculo de Construcciones, las reglas que acabamos de presentar son fijas y constituyen la infraestructura de cualquier teoría que se defina sobre ellas. Una teoría se especifica por medio de la declaración de constantes y sus tipos, así como de igualdades entre ellas. En el caso de la teoría de las Familias Inductivas, las reglas de formación, introducción, eliminación e igualdad para la definición de nuevos conjuntos (tipos) se definen mediante la declaración del tipado de constantes nuevas, y no mediante reglas de inferencia. Por ejemplo, la segunda regla de introducción de los números naturales, descrita originalmente en ITT mediante la regla

$$\frac{a : \mathbb{N}}{s(a) : \mathbb{N}},$$

viene dada en esta teoría por la constante $s : (a : \mathbb{N})\mathbb{N}$.

1.4.2 Un esquema para definiciones inductivas

En esta sección se presenta el esquema para definiciones inductivas. Primero se describe la forma general que tienen las reglas de formación e introducción. Después se explica el principio de inversión para obtener las reglas de eliminación e igualdad de forma estándar a partir de las reglas de formación e introducción. En la sección 1.4.4 veremos ejemplos de uso de este esquema para definir conjuntos y familias de conjuntos.

La definición de un conjunto o familia de conjuntos puede depender de otros conjuntos definidos previamente.

Regla de formación

Hay una única regla de formación para cada conjunto definido, y tiene la siguiente forma:

$$\begin{aligned} P : {} & (A :: \sigma) \\ & (a :: \alpha[A]) \\ & \mathit{set}, \end{aligned} \tag{1.32}$$

donde

- σ es una secuencia de tipos.
- $\alpha[A]$ es una secuencia de s-tipos bajo las hipótesis $A :: \sigma$.

Las premisas $A :: \sigma$, llamadas *parámetros* de P, son globales en la definición. Dados unos parámetros A, podemos ver $P_A = P(A)$ como una familia de conjuntos con índices $a :: \alpha[A]$.

Reglas de introducción

Los conjuntos tienen un número finito (posiblemente nulo) de reglas de introducción, también llamadas *constructores*. Cada regla de introducción es de la forma

$$\begin{aligned} intro : & (A :: \sigma) \\ & (b :: \beta[A]) \\ & (u :: \gamma[A, b]) \\ & P_A(p[A, b]), \end{aligned} \tag{1.33}$$

donde

- $\beta[A]$ es una secuencia de s-tipos.
- Cada $\gamma_i[A, b]$ tiene la forma $(x :: \xi_i[A, b])P_A(p_i[A, b, x])$, donde
  - $\xi_i[A, b]$ es una secuencia de s-tipos.
  - $p_i[A, b, x]$ es de tipo $\alpha[A]$.
- $p[A, b]$ es de tipo $\alpha[A]$.

Las premisas $b :: \beta[A]$ se denominan *argumentos no recursivos* del constructor *intro*, mientras que $u :: \gamma[A, b]$ son los *argumentos recursivos*. Las variables $x :: \xi_i[A, b]$ se corresponden con los argumentos de la inducción generalizada. El conjunto P que está siendo definido no puede aparecer en ξ_i, así que se trata de inducción estrictamente positiva. El caso en el que todos los ξ_i son vacíos se corresponde con la inducción ordinaria.

Veremos que cada constructor del conjunto P se corresponde con un caso de inducción sobre los elementos de P.

Regla de eliminación

Cada conjunto definido inductivamente tiene una única regla de eliminación, obtenida a partir de las reglas de formación (1.32) e introducción (1.33) de la siguiente forma:

$$\begin{aligned} elim \ : \ & (A \ : : \ \sigma) \\ & (C \ : \ (a \ : : \ \alpha[A]) \\ & \qquad (c \ : \ P_A(a)) \\ & \qquad set) \\ & (e \ : : \ \epsilon[A]) \\ & (a \ : : \ \alpha[A]) \\ & (c \ : \ P_A(a)) \\ & C(a, c). \end{aligned} \tag{1.34}$$

La variable C es a lo que a menudo se conoce como el *motivo* u objetivo de la recursión. Dados unos índices a, $C(a, \cdot)$ es el conjunto de llegada de la función que queremos definir mediante recursión sobre $P_A(a)$. Alternativamente, $C(a, \cdot)$ es el predicado sobre $P_A(a)$ que queremos probar mediante inducción en $P_A(a)$.

La premisa $c \ : \ P_A(a)$ se denomina *premisa mayor*, mientras que las premisas $e \ : : \ \epsilon[A]$ se denominan *premisas menores*. Hay una premisa menor e_j por cada constructor del conjunto. El tipo $\epsilon_j[A]$ de la premisa menor asociada al constructor *intro* tiene la forma

$$\begin{aligned} & (b \ : : \ \beta[A]) \\ & (u \ : : \ \gamma[A, b]) \\ & (v \ : : \ \delta[A, b]) \\ & C(p[A, b], intro_A(b, u)), \end{aligned} \tag{1.35}$$

donde $\delta[A, b]$ tiene la misma longitud que $\gamma[A, b]$ y cada $\delta_i[A, b]$ es $(x \ : : \ \xi_i[A, b])C(p_i[A, b, x], u_i(x))$.

Esencialmente, ϵ_j es la proposición "dados los parámetros recursivos u y no recursivos b del constructor *intro*, y suponiendo $\delta[A, b]$, es decir, suponiendo que ya he probado/definido lo que quiero para el caso de inducción correspondiente a *intro*, entonces también puedo probar/definir lo que quiero para el objeto resultante de aplicar el constructor *intro*".

Si tenemos una prueba e_j de ϵ_j para todo j significa que todos los casos de inducción están cubiertos. En tal caso, *elim* nos permite probar/definir lo que queremos para cualquier elemento del conjunto. Dicho de otra forma, cada premisa menor e_j es un caso de inducción de nuestra prueba/definición: si *intro* no tiene argumentos recursivos, entonces la correspondiente premisa menor es el caso base; en otro caso, la premisa menor es un paso de inducción.

Reglas de igualdad

Hay una regla de igualdad por cada regla de introducción. La regla de igualdad correspondiente al constructor *intro* es

$$\begin{aligned}(A, C, e, b, u)elim_{A,C}(e, p[A, b], intro_A(b, u)) = \\ = (A, C, e, b, u)e_j(b, u, v) \ : \ & (A \ : : \ \sigma) \\ & (C \ : \ (a \ : : \ \alpha[A]) \\ & \qquad (c \ : \ P_A(a)) \\ & \qquad set) \\ & (e \ : : \ \epsilon[A]) \\ & (b \ : : \ \beta[A]) \\ & (u \ : : \ \gamma[A, b]) \\ & C(p[A, b], intro_A(b, u)),\end{aligned} \tag{1.36}$$

donde v_i es $(x)elim_{A,C}(e, p_i[A, b, x], u_i(x))$, de tipo $(x \ : : \ \xi_i[A, b])C(p_i[A, b, x], u_i(x))$.

Las reglas de igualdad describen cómo evaluar de forma efectiva las funciones definidas mediante la regla de eliminación: se utilizan reiteradamente las premisas menores para ejecutar recursivamente la regla de eliminación hasta que llegar a un elemento de P que haya sido construido sin argumentos recursivos, es decir, hasta llegar a un caso base.

1.4.3 Un esquema para definiciones recursivas

La idea es generalizar el esquema para las reglas de eliminación e igualdad permitiendo que C sea una familia arbitraria de tipos en lugar de restringirlo a una familia de conjuntos. De esta forma, el resultado de una función definida por P-recursión podría ser un objeto arbitrario, y no necesariamente un elemento de un conjunto.

Con lo descrito en la sección anterior, las reglas de eliminación e igualdad quedan determinadas por las reglas de formación e introducción. Por lo tanto, si queremos definir una función f mediante P-recursión, tenemos que hacerlo en términos de la regla de P-eliminación, y las ecuaciones recursivas de f deberán obtenerse usando las reglas de P-igualdad. En esta sección se proponen esquemas para introducir reglas de eliminación y de igualdad sobre conjuntos definidos inductivamente. Estos esquemas especifican la forma general que deben tener el tipo y las ecuaciones recursivas de una función o familia definida por P-recursión. De hecho, las reglas de eliminación e igualdad de la sección anterior son instancias de los esquemas presentados en esta sección.

Regla de eliminación esquemática

El tipo de una función f definida por P-recursión debe tener la siguiente forma:

$$\begin{aligned} f \ :\ & (B \ ::\ \tau) \\ & (a \ ::\ \alpha[Q[B]]) \\ & (c \ :\ P_{Q[B]}(a)) \\ & \psi[B, a, c], \end{aligned} \tag{1.37}$$

donde

- τ es una secuencia de tipos
- la expresión α es la que aparece en la regla de formación (1.32) de P (a es una secuencia de índices de P),
- $Q[B] \ ::\ \sigma$, siendo σ la expresión que aparece en la regla de formación (1.32) de P ($Q[B]$ es una secuencia de parámetros de P),
- $\psi[B, a, c]$ es un tipo.

Este esquema generaliza a la regla de eliminación definida en la sección anterior. En efecto, si tomamos

$$\begin{aligned} (B \ ::\ \tau) = (A \ ::\ \sigma) & \\ (C \ :\ & (a \ ::\ \alpha[A]) \\ & (c \ :\ P_A(a)) \\ & \mathit{set}) \\ (e \ ::\ & \epsilon[A]), \end{aligned}$$

$Q[B] = A$ y $\psi[B, a, c] = C(a, c)$, obtenemos la regla de eliminación 1.34.

Reglas de igualdad esquemáticas

Para definir una función f por P-recursión es necesario establecer una ecuación recursiva E_j por cada regla de introducción de P. La ecuación recursiva asociada al constructor *intro* debe tener la forma

$$\begin{aligned} E_j \ : \ & (B \ : : \ \tau) \\ & (b \ : : \ \beta[Q[B]]) \\ & (u \ : : \ \gamma[Q[B], b]) \\ & (v \ : : \ \delta[Q[B], b]) \\ & \psi[B, p[Q[B], b], intro_{Q[B]}(b, u)], \end{aligned}$$

donde las expresiones β, γ, δ y p son las que aparecen en la regla de introducción (1.33) correspondiente al constructor *intro*.

Una vez establecidas las ecuaciones recursivas de f, al regla de igualdad de f correspondiente la constructor *intro* es

$$\begin{aligned} (B, b, u) f_B(p[Q[B], b], intro_{Q[B]}(b, u)) = & \\ = (B, b, u) {E_j}_B(b, u, v) \ : \ & (B \ : : \ \tau) \\ & (b \ : : \ \beta[Q[B]]) \\ & (u \ : : \ \gamma[Q[B], b]) \\ & \psi[B, p[Q[B], b], intro_{Q[B]}(b, u)], \end{aligned} \tag{1.38}$$

donde v_i es $(x) f_B(p_i[A, b, x], u_i(x))$ y p_i hace referencia a la expresión que aparece en la regla de introducción (1.33) correspondiente al constructor *intro*.

1.4.4 Algunos ejemplos

Listas de una cierta longitud

La regla de formación para el conjunto de listas de elementos de A de longitud a es

$$\begin{aligned} List \ : \ & (A \ : \ set) \\ & (a \ : \ \mathbb{N}) \\ & set, \end{aligned}$$

El conjunto A es un parámetro y $\mathbb{N}$ es el conjunto de índices. Esta definición depende por tanto de haber definido previamente el conjunto de los números naturales.

Tenemos dos reglas de introducción: una para la lista vacía y otra para el constructor correspondiente a añadir un elemento a una lista. La primera es

$$\begin{aligned} nil &: (A : set) \\ &\quad List_A(0). \end{aligned}$$

No hay argumentos ni recursivos ni argumentos no recursivos.

La segunda regla de introducción es

$$\begin{aligned} cons &: (A : set) \\ &\quad (b_1 : \mathbb{N}) \\ &\quad (b_2 : A) \\ &\quad (u : List_A(b_1)) \\ &\quad List_A(s(b_1)). \end{aligned}$$

Hay dos argumentos no recursivos, $b_1 : \mathbb{N}$ y $b_2 : A$ y un argumento recursivo ordinario $u : List_A(b_1)$.

Ahora podemos derivar la regla de eliminación

$$\begin{aligned} listrec &: (A : set) \\ &\quad (C : (a : \mathbb{N}) \\ &\qquad (c : List_A(a)) \\ &\qquad set) \\ &\quad (e_1 : C(0, nil_A)) \\ &\quad (e_2 : (b_1 : \mathbb{N}) \\ &\qquad (b_2 : A) \\ &\qquad (u : List_A(b_1)) \\ &\qquad (v : C(b_1, u)) \\ &\qquad C(s(b_1), cons_A(b_1, b_2, u))) \\ &\quad (a : \mathbb{N}) \\ &\quad (c : List_A(a)) \\ &\quad C(a, c). \end{aligned}$$

La regla de igualdad asociada al constructor *nil* es

$$
\begin{aligned}
(A, C, e_1, e_2)\mathit{listrec}_{A,C}(e_1, e_2, 0, \mathit{nil}_A) = (A, C, e_1, e_2)e_1 \ : \ & (A \ : \ \mathit{set}) \\
& (C \ : \ (a \ : \ \mathbb{N}) \\
& \qquad (c \ : \ \mathit{List}_A(a)) \\
& \qquad \mathit{set}) \\
& (e_1 \ : \ C(0, \mathit{nil}_A)) \\
& (e_2 \ : \ (b_1 \ : \ \mathbb{N}) \\
& \qquad (b_2 \ : \ A) \\
& \qquad (u \ : \ \mathit{List}_A(b_1)) \\
& \qquad (v \ : \ C(b_1, u)) \\
& \qquad C(s(b_1), \mathit{cons}_A(b_1, b_2, u))) \\
& C(0, \mathit{nil}_A).
\end{aligned}
$$

La regla de igualdad para el constructor *cons* es

$$
\begin{aligned}
(A, C, e_1, e_2, b_1, b_2, u)\mathit{listrec}_{A,C}(e_1, e_2, s(b_1), \mathit{cons}_A(b_1, b_2, u)) = & \\
= (A, C, e_1, e_2, b_1, b_2, u)e_2(b_1, b_2, u, \mathit{listrec}_{A,C}(e_1, e_2, b_1, u) \ : & \\
: (A \ : \ \mathit{set}) & \\
(C \ : \ (a \ : \ \mathbb{N})(c \ : \ \mathit{List}_A(a))\mathit{set}) & \\
(e_1 \ : \ C(0, \mathit{nil}_A)) & \\
(e_2 \ : \ (b_1 \ : \ \mathbb{N}) & \\
\qquad (b_2 \ : \ A) & \\
\qquad (u \ : \ \mathit{List}_A(b_1)) & \\
\qquad (v \ : \ C(b_1, u)) & \\
\qquad C(s(b_1), \mathit{cons}_A(b_1, b_2, u))) & \\
(b_1 \ : \ \mathbb{N}) & \\
(b_2 \ : \ A) & \\
(u \ : \ \mathit{List}_A(b_1)) & \\
C(s(b_1), \mathit{cons}_A(b_1, b_2, u)). &
\end{aligned}
$$

Supongamos que también tenemos definido el conjunto $\mathit{List'} \ : \ (A \ : \ \mathit{set})\mathit{set}$ de listas en el que no es necesario especificar la longitud. Entonces podemos definir una función *forgetlength* por *List*-recursión que toma una lista de List_A y devuelve la

misma lista pero vista dentro de $List'_A$. Si solo tuviéramos las reglas de eliminación e igualdad que acabamos de escribir (correspondientes al esquema de la sección 1.4.2), tendríamos que definir la función en términos de *listrec*:

$$\begin{aligned} forgetlength = (A)listrec_{A,(a,c)List'_A}(nil'_A, (b_1, b_2, u, v)cons'_A(b_2, v)) \; : \; & (A \; : \; set) \\ & (a \; : \; \mathbb{N}) \\ & (c \; : \; List_A(a)) \\ & List'_A. \end{aligned}$$

De esta definición podemos derivar las ecuaciones de recursión mediante las reglas de igualdad de *listrec*:

$$forgetlength_A(0, nil_A) = nil'_A \; : \; List'_A \tag{a}$$

$$forgetlength_A(s(b_1), cons_A(b_1, b_2, u)) = cons'_A(b_2, forgetlength_A(b_1, u)) \; : \; List'_A \tag{b}$$

Sin embargo, gracias al esquema para reglas de eliminación de la sección 1.4.3 es posible definir directamente la función *forgetlength* sin pasar por *listrec*. En efecto, tomando $\tau = set$, $Q[B] = B$ y $\psi[B, a, c] = List'_B$ obtenemos la regla de tipado de *forgetlength*:

$$\begin{aligned} forgetlength \; : \; & (B \; : \; set) \\ & (a \; : \; \mathbb{N}) \\ & (c \; : \; List_B(a)) \\ & List'_B. \end{aligned}$$

Si ahora tomamos

$$E_{1\,B} = nil'_B \; : \; List'_B$$

y

$$\begin{aligned} E_{2\,B} = (b_1, b_2, u, v)cons'_B(b_2, u) \; : \; & (b_1 \; : \; \mathbb{N}) \\ & (b_2 \; : \; B) \\ & (u \; : \; List_B(b_1)) \\ & (v \; : \; List'_B) \\ & List'_B \end{aligned}$$

para las reglas de igualdad de *forgetlength*, obtenemos precisamente las ecuaciones recursivas (a) y (b).

Productos dependientes

Vemos de nuevo cómo se definen los productos dependientes, esta vez como conjunto definido inductivamente. El resultado obtenido es el mismo que el definido en la Teoría Intuicionista de Tipos (ver Sección 1.2.4), pues las definiciones de Martin-Löf son instancias del esquema propuesto en las Familias Inductivas.

La regla de formación del producto dependiente es la siguiente:

$$\begin{aligned}\Pi : &(A_1 : set)\\ &(A_2 : (x : A_1)set)\\ &set\end{aligned}$$

A_1 y A_2 son los parámetros. No hay índices.

Esta vez solo hay un constructor, correspondiente a la λ-abstracción:

$$\begin{aligned}\lambda : &(A_1 : set)\\ &(A_2 : (x : A_1)set)\\ &(b : (x : A_1)A_2(x))\\ &\Pi(A_1, A_2)\end{aligned}$$

La premisa b es el argumento no recursivo. No hay argumentos recursivos.

Podemos derivar ahora la regla de eliminación, que en este caso cuenta con una única premisa menor e:

$$\begin{aligned}F : &(A_1 : set)\\ &(A_2 : (x : A_1)set)\\ &(C : (c : \Pi(A_1, A_2))\\ &\qquad set)\\ &(e : (b : (x : A_1)A_2(x))\\ &\qquad C(\lambda_{A_1,A_2}(b)))\\ &(c : \Pi(A_1, A_2))\\ &C(c)\end{aligned}$$

Observemos que esta regla de eliminación no es la regla de eliminación (1.8) propuesta originalmente por Martin-Löf, que corresponde a la aplicación $\mathrm{Ap}(z, t)$ de un

término $z : \Pi(A_1, A_2)$ a un término $t : A_1$ para obtener un término de $A_2(t)$. Sin embargo, en el prefacio de *Intuitionistic Type Theory* [3] se explica que si definimos

$$\begin{aligned}\mathrm{Ap}_{A_1,A_2} = (z,t)F_{A_1,A_2,(c)A_2(t)}\,((b)b(t),z) : (z &: \Pi(A_1,A_2))\\ (t &: A_1)\\ A_2(t),\end{aligned}$$

obtenemos la regla original de eliminación a partir de la dada por el esquema de la sección 1.4.2. Recíprocamente, si partimos de la regla original, podemos definir

$$\begin{aligned}F_{A_1,A_2} = (C,e,c)e((x)\mathrm{Ap}_{A_1,A_2}(c,x)) : (C &: (c : \Pi(A_1,A_2))\\ &\quad set)\\ (e &: (b : (x : A_1)A_2(x))\\ &\quad C(\lambda_{A_1,A_2}(b)))\\ (c &: \Pi(A_1,A_2))\\ C(c)\end{aligned}$$

para obtener la regla de eliminación esquemática, donde se ha usado que $\lambda_{A_1,A_2}((x)\mathrm{Ap}_{A_1,A_2}(c,x)) = c$ (η-conversión).

Conjuntos bien ordenados

Como ejemplo de inducción generalizada vamos a ver la definición del constructor $\mathbb{W}$ de conjuntos bien ordenados, cuya recursión asociada corresponde a la *inducción transfinita*.

La regla de $\mathbb{W}$-formación es la siguiente:

$$\begin{aligned}\mathbb{W} : (A_1 &: set)\\ (A_2 &: (x : A_1)set)\end{aligned}$$

Tanto A_1 como A_2 son parámetros. No hay índices. La idea es que A_1 es el conjunto que queremos bien ordenado y, dado $b : A_1$, $A_2(b)$ es un conjunto de nombres para los predecesores de b.

La regla de introducción revela cómo son los elementos de $\mathbb{W}(A_1, A_2)$

$$\begin{aligned} sup : & (A_1 : set) \\ & (A_2 : (x : A_1)set) \\ & (b : A_1) \\ & (u : (x : A_2(b))\mathbb{W}(A_1, A_2)) \\ & \mathbb{W}(A_1, A_2) \end{aligned}$$

Hay un argumento no recursivo $b : A_1$ y un argumento recursivo generalizado u con variable $x : A_2(b)$. La idea es que el elemento $sup_{A_1,A_2}(b, u)$ es el supremo de todos sus predecesores, es decir, el supremo de todos los elementos $u(x)$ cuando x recorre $A_2(b)$. Dada una elección de $x : A_2(b)$ obtenemos un predecesor $u(x) = sup_{A_1,A_2}(b_1, u_1)$, del que a su vez obtenemos un predecesor $u(x_1) = sup_{A_1,A_2}(b_2, u_2)$... hasta que llegamos a un elemento mínimo $sup_{A_1,A_2}(b_n, u_n)$ para el cual el conjunto $A_2(b_n)$ es vacío.

Podemos derivar la regla de eliminación:

$$\begin{aligned} T : & (A_1 : set) \\ & (A_2 : (x : A_1)set) \\ & (C : (c : \mathbb{W}(A_1, A_2))set) \\ & (e : (b : A_1) \\ & \quad (u : (x : A_2(b))\mathbb{W}(A_1, A_2)) \\ & \quad (v : (x : A_2(b))C(u(x))) \\ & \quad C(sup_{A_1,A_2}(b, u)) \\ & (c : \mathbb{W}(A_1, A_2)) \\ & C(c) \end{aligned}$$

La regla de eliminación para el conjunto $\mathbb{W}(A_1, A_2)$ corresponde al principio de inducción transfinita: si un predicado C sobre $\mathbb{W}(A_1, A_2)$ satisface la propiedad "si C es cierto para todos los predecesores $u(x)$ de un cierto $sup_{A_1,A_2}(b, u)$, entonces C es cierto para el propio $sup_{A_1,A_2}(b, u)$" entonces el predicado C es cierto para todos los elementos de $\mathbb{W}(A_1, A_2)$.

Puede probarse (ver pág. 47 de [3]) que $\mathbb{W}(A_1, A_2)$ está habitado si y sólo si $(\exists b : A_1)\neg A_2(b)$, es decir, si y sólo si existe algún elemento b de A_1 que no tiene predecesores.

1.4.5 Inducción simultánea

Las reglas del esquema de inducción simultánea son análogas a las descritas en la sección 1.4.2. La diferencia es que en este caso se permite la definición simultánea de varias familias de conjuntos P_k en un mismo bloque, pudiendo cada familia referenciar recursivamente al resto.

Todos los conjuntos definidos en un mismo bloque deben tener los mismos parámetros ($A :: \sigma$), y cada uno puede tener sus propios índices ($a :: \alpha_k[A]$). Cada regla de formación de un bloque tiene la forma:

$$\begin{aligned} P_k \ : \ & (A :: \sigma) \\ & (a :: \alpha_k[A]) \\ & set \end{aligned} \tag{1.39}$$

En los parámetros recursivos $u :: \gamma[A, b]$ de las reglas de introducción puede aparecer cualquiera de los conjuntos definidos en el bloque. Cada regla de introducción tiene la forma

$$\begin{aligned} intro \ : \ & (A :: \sigma) \\ & (b :: \beta[A]) \\ & (u :: \gamma[A, b]) \\ & P_{k\,A}(p[A, b]), \end{aligned} \tag{1.40}$$

donde cada $\gamma_i[A, b]$ tiene la forma $(x :: \xi[A, b])P_{k_i\,A}(p_i[A, b, x])$.

Cada conjunto definido en el bloque tiene una regla de eliminación. En cada regla de eliminación ha de haber una premisa menor ($e_j :: \epsilon[A]$) por cada regla de introducción *del bloque*. También debe haber un motivo $(C_k \ : \ (a :: \alpha_k[A])(c \ : \ P_{k\,A}(a))set)$ por cada conjunto definido en el bloque. Por lo demás, la forma de cada regla de eliminación es análoga a lo visto en la sección 1.4.2:

$$\begin{aligned} elim_l \ : \ & (A :: \sigma) \\ & (C :: \phi[A]) \\ & (e :: \epsilon[A]) \\ & (a :: \alpha_l[A]) \\ & (c \ : \ P_{l\,A}(a)) \\ & C_l(a, c) \end{aligned} \tag{1.41}$$

Hay una regla de igualdad por cada regla de introducción del bloque, y son completamente análogas a las de la sección 1.4.2.

Un ejemplo típico de inducción simultánea es la definición de los predicados "par" e "impar" sobre los números naturales, que pueden definirse a partir de las reglas "0 es par", "$n + 1$ es impar si n es par" y "$n + 1$ es par si n es impar". En el capítulo 2 veremos este ejemplo como aplicación de la inducción simultánea en Lean.

2 El demostrador Lean

En este capítulo estudiamos la teoría de tipos que constituye la base fundacional del sistema Lean[1]. Acompañaremos las nociones teóricas con fragmentos de código, por lo que este capítulo también servirá en cierto modo como tutorial del lenguaje del sistema, aunque no indagaremos en las tácticas automatizadas que ofrece. La principal referencia de este capítulo es el tutorial de Lean4 [4].

2.1 Teoría de tipos de Lean

Lean está construido sobre una teoría de tipos dependientes basada en el Cálculo de Construcciones (ver Sección 1.3), con una jerarquía numerable de universos no acumulativos y con familias de tipos inductivos.

2.1.1 Los tipos tratados como objetos

Igual que sucede en el Cálculo de Construcciones, los tipos también son tratados como términos (u objetos). Por lo tanto, los propios tipos también tienen un tipo. El comando `#check` pide a Lean que muestre el tipo de un término:

```
#check Nat                -- Type
#check Bool               -- Type
#check Nat → Bool         -- Type
#check Nat × Bool         -- Type
#check List Nat           -- Type
#check Prod Nat Bool      -- Type
```

[1] https://leanprover.github.io

Sin embargo, recordemos que en el Cálculo de Construcciones el tipo de todos los tipos $*$ es un caso especial que no se considera un término y que por tanto no tiene tipo. En Lean *todas las expresiones se consideran términos* y por tanto tienen un tipo. Es natural entonces preguntarse de qué tipo es el término `Type`.

```
#check Type     -- Type 1
```

Lean tiene una jerarquía infinita de tipos: `Type = Type 0 : Type 1`, `Type 1 : Type 2` ... Puede pensarse en `Type 0` como el universo de los tipos "ordinarios" o "pequeños". `Type 1` es un universo mayor que contiene a `Type 0` como elemento, `Type 2` es un universo aún mayor, etc.

Algunas operaciones necesitan ser polimórficas con respecto a los diferentes universos. Por ejemplo, `List α` (el tipo de las listas formadas por elementos de tipo `α`) debería tener sentido independientemente del universo en el que viva el tipo `α`. Esto explica la salida que nos muestra Lean si le preguntamos el tipo de `List` o `Prod`:

```
#check List     -- Type u_1 → Type u_1
#check Prod     -- Type u_1 → Type u_2 → Type (max u_1 u_2)
```

2.1.2 Lambda-abstracción y tipos dependientes

Igual que en el Cálculo de Construcciones, la λ-abstracción, la aplicación de funciones a términos y los productos dependientes son las operaciones básicas sobre las que se construye la teoría de Lean.

En Lean se usa la palabra reservada `fun` (o `λ`) para definir una función a partir de una expresión:

```
#check fun (x : Nat) => x + 5    -- Nat → Nat
#check λ (x : Nat) => x + 5      -- Nat → Nat
-- 'λ' y 'fun' significan lo mismo
```

En general, si tenemos una variable `x : α` y podemos construir una expresión `t : β x` (el tipo `β x` depende de `x`), entonces la expresión `fun (x : α) => t` es un término de tipo `(x : α) → β x`. El tipo `(x : α) → β x` es el producto dependiente que hemos visto en las teorías de las secciones 1.2, 1.3 y 1.4.

Nótese que `(x : α) → γ` tiene sentido para cualquier expresión `γ : Type u`. Cuando `γ` depende de `x` (como en el párrafo anterior), entonces `(x : α) → γ` denota un producto dependiente. Si `γ` no depende de `a`, entonces `(x : α) → γ` es exactamente lo mismo que `α → γ` (lo segundo es notación para lo primero cuando `γ` no depende de `x`). En definitiva, en Lean se usa una notación que unifica los productos dependientes y los tipos de funciones. Es por eso que, en el contexto de Lean, los productos dependientes son conocidos más comúnmente como *tipos de funciones dependientes*.

Un ejemplo básico de función dependiente es el constructor `cons` de listas, que añade un elemento `a : α` a una lista `as : List α`. Los tipos del segundo y tercer argumentos de `cons` dependen del primero:

```
def cons (α : Type) (a : α) (as : List α) : List α :=
  List.cons a as

#check cons Nat         -- Nat → List Nat → List Nat
#check cons Bool        -- Bool → List Bool → List Bool
#check cons             -- (α : Type) → α → List α → List α
```

De la misma forma que las funciones dependientes `(a : α) → β a` generalizan el concepto de función `α → β`, los productos cartesianos dependientes[2] `(a : α) × β a` generalizan el producto cartesiano `α × β`. Los tipos de productos cartesianos dependientes también son llamados tipos *sigma* o *sumas dependientes* (ya los vimos en la sección 1.2.4 del anterior capítulo), y pueden escribirse también como `Σ a : α, β a`. Podemos crear un par dependiente mediante `Sigma.mk a b`.

Igualdad definicional

La variable ligada `x` que aparece en el término `(x : α) → β x` es simplemente un nombre provisional a la hora de escribir la λ-expresión, es decir, si la renombramos obtenemos el mismo objeto (obtenemos un término definicionalmente equivalente). Esto es lo que se conoce como α-conversión. Lean reconoce esta equivalencia.

Por otro lado, podemos aplicar un término `t : (x : α) → β` a un término `a : α` para obtener `t a : β`. Si `t` es de la forma `fun x : α => t'`, siendo

[2] No confundir con los productos dependientes `Π a : α, β a`, que son equivalentes a los tipos de funciones dependientes `(a : α) → β a`.

`t'` una expresión de tipo `β` que puede depender de `x`, entonces el término `t a` es definicionalmente equivalente al resultado de sustituir toda ocurrencia de `x` en `t'` por `a`. Como ya vimos a lo largo del anterior capítulo (por ejemplo, en la regla (1.9)), esto se conoce como β-conversión, y Lean también reconoce esa equivalencia.

En general, igual que en el Cálculo de Construcciones, los términos en Lean que no son tipos tienen un comportamiento computacional y admiten una noción de *normalización/reducción*. Dos términos que se reducen al mismo valor son llamados equivalentes o iguales definicionalmente.

2.1.3 Argumentos implícitos

Consideremos las siguientes funciones y términos sobre listas:

```
universe u    -- Para declarar la variable de universos 'u'

def cons (α : Type u) (a : α) (as : List α) : List α :=
  List.cons a as

def nil (α : Type u) : List α := List.nil

#check List           -- Type u → Type u
#check cons     -- (α : Type u) → α → List α → List α
#check nil      -- (α : Type u) → List α
```

Como los constructores de listas son polimórficos con respecto a los tipos, tenemos que insertar el tipo de los elementos de las lista cada vez que queramos usarlos. Sin embargo, esta información es redundante: puede inferirse el primer argumento en `cons Nat 5 (nil Nat)` a partir del hecho de que el segundo argumento, `5`, es de tipo `Nat`. De igual forma puede inferirse el argumento en `nil Nat`, no directamente de esa expresión, sino del hecho de que la expresión se escribe como un argumento de la función `cons`, que espera un elemento de tipo `List α` (donde `α` ya ha sido inferido).

La inferencia de argumentos a partir de los tipos es una característica central de la teoría de tipos dependientes, que es posible gracias a que el tipado de los términos proporciona mucha información. En Lean puede usarse el guión bajo `_` para especificar que el sistema debe inferir ese argumento de forma automática.

```
#check cons _ 5 (nil _)    -- List Nat
```

Sin embargo, sigue siendo tedioso tener que escribir todos los guiones bajos. Cuando una función toma un argumento que, por lo general, puede ser inferido por el contexto, Lean permite especificar que ese argumento debe, por defecto, dejarse implícito. Esto se hace escribiendo los argumento entre llaves en la definición de la función:

```
universe u

def cons {α : Type u} (a : α) (as : List α) : List α :=
    List.cons a as

def nil {α : Type u} : List α := List.nil

#check cons 5 nil    -- List Nat
```

En la librería de Lean se usa esta funcionalidad para definir una función identidad polimórfica:

```
universe u
def id {α : Type u} (x : α) := x

#check id        -- ?m → ?m  (?m es un 'placeholder'
                 -- introducido por el sistema)
#check id 1       -- Nat
#check id "hello" -- String
```

Lean posee mecanismos complejos para instanciar argumentos implícitos, y veremos que pueden ser utilizados para inferir tipos, predicados e incluso pruebas. El proceso de instanciación de los "agujeros" o "placeholders" originados por los argumentos implícitos se denomina *desarrollo* (*elaboration*).

Puede suceder que hayamos declarado un argumento de una función como implícito pero que en cierto momento queramos escribir el argumento explícitamente al usar la función. Si `f` es una función con argumentos implícitos, entonces `@f` denota la misma función con todos los argumentos convertidos en explícitos:

```
#check @id         -- {α : Type u_1} → α → α
#check @id Nat     -- Nat → Nat
#check @id Bool    -- Bool → Bool
#check @id Nat 1      -- Nat
#check @id Bool true -- Bool
```

2.2 Proposiciones y pruebas

En esta sección se explica cómo escribir afirmaciones matemáticas y pruebas en el lenguaje de la teoría de tipos dependientes de Lean.

2.2.1 El universo de las proposiciones

Como sucede en las teorías de tipos que hemos visto en el capítulo 1, Lean aprovecha la correspondencia de Curry-Howard y, más concretamente, el paradigma de las proposiciones como tipos, para integrar las proposiciones y pruebas en su lenguaje. Sin embargo, en vez de reutilizar el tipo `Type` e incluir en él las proposiciones además de los tipos básicos, Lean introduce un nuevo universo llamado `Prop`, el tipo de todas las proposiciones.

A lo largo de este capítulo veremos las particularidades que diferencian el universo `Prop` de `Type`, así como las razones que justifican esas particularidades. Para empezar, el tipo `Prop` es azúcar sintáctico para `Sort 0`, que se encuentra en la base de la jerarquía de universos descrita en la sección anterior. Más aún, cada tipo `Type u` también es azúcar sintáctico para `Sort u+1`. El universo `Prop`, igual que el resto de universos de la jerarquía, es cerrado con respecto al constructor `→`: si tenemos `p q : Prop`, entonces `p → q : Prop` (de hecho, como ya sabemos, `p → q` es la proposición "`p` implica `q`").

El universo `Prop` de Lean implementa el paradigma de la *irrelevancia de las pruebas*. Dada una proposición `p : Prop`, Lean trata dos términos cualesquiera `t1 t2 : p` como definicionalmente iguales, de la misma forma que trata los términos `(fun x => t) s` y `t[s/x]` como definicionalmente iguales. Esto significa que aunque podemos tratar las pruebas `t : p` como objetos ordinarios en el lenguaje de la teoría de tipos dependientes, estos no contienen información más allá del

hecho de que `p` es cierta. Esta es una de las diferencias fundamentales entre las proposiciones de `Prop` y los tipos de `Type`. Si `α : Type` es un tipo, los elementos `a : α` sí albergan contenido computacional relevante. Está claro, por ejemplo, que `0 : Nat` y `1 : Nat` no son definicionalmente iguales.

En la línea de lo explicado en el anterior párrafo, los términos de Lean cuyo tipo vive en `Type` (o, equivalentemente en `Sort 1`), como los naturales `Nat` o los booleanos `Bool`, se denominan datos. Por otro lado, los términos cuyo tipo vive en `Prop` (o, equivalentemente, en `Sort 0`) son los términos cuyo tipo es una proposición, es decir, son pruebas de proposiciones. La distinción entre `Type` y `Sort` es útil, por ejemplo, cuando queremos definir una función polimórfica. Si queremos que una función actúe sobre términos de cualquier tipo, incluyendo las pruebas, debemos definirla usando `Sort`. Si queremos excluir a las pruebas como posible entrada de la función, debemos usar `Type`. Por ejemplo, la identidad polimórfica que vimos en la sección anterior no es exactamente la que podemos encontrar en la librería de Lean. La verdadera función identidad también puede tomar pruebas como entrada:

```
universe u
def id {α : Sort u} (x : α) := x
```

2.2.2 Trabajando con las proposiciones como tipos

Bajo el paradigma de las proposiciones como tipos, las proposiciones que constan únicamente de la conectiva `→` pueden probarse usando únicamente λ-abstracción y aplicación. Por ejemplo:

```
variable {p : Prop}    -- Introduce la variable 'p' en el contexto
variable {q : Prop}    -- Introduce la variable 'q' en el contexto

theorem t1 : p → q → p := fun hp : p => fun hq : q => hp
```

El teorema `t1` se trata de un término de tipo `p → q → p` que, dado un término de tipo `p` y otro de tipo `q`, construye un término de tipo `p`. En concreto, dados los términos `hp : p` y `hq : q`, `t1` devuelve el término `hp`. Si comparamos esta prueba con la expresión `fun x : α => fun y : β => x` de tipo `α → β → α`, donde `α β : Type`, vemos que ambas tienen exactamente la misma forma; la única diferencia es que `p` y `q` son elementos de `Prop` y no de `Type`. Intuitivamente, la

λ-abstracción `fun hp : p` se corresponde con la introducción de la hipótesis "`p` es cierta".

Gracias a la identificación entre proposiciones y tipos podemos aplicar el teorema `t1` como si fuera una función:

```
theorem t1 (hp : p) (hq : q) : p := hp    -- Forma alternativa de
                                          -- escribir el teorema t1
axiom hp : p   -- 'axiom' sirve para declarar la
               -- prueba de cualquier proposición
theorem t2 : q → p := t1 hp
```

De acuerdo con la correspondencia de Curry-Howard, nótese que el comando `theorem` no es más que una versión alternativa del comando `def`; para el comprobador de tipos de Lean no hay diferencia entre definir un objeto de un cierto tipo y dar una prueba de cierta proposición.

Técnicamente sí que hay algunas diferencias entre teoremas y definiciones. En circunstancias normales nunca es necesario desplegar la "definición" de un teorema (sí es necesario, por ejemplo, desplegar la definición de funciones cuando queremos evaluarlas). Una vez la prueba de un teorema está completa, lo único que necesitamos saber es que la prueba existe, sin importar cuál es la prueba. Por esta razón, Lean etiqueta las pruebas como irreducibles, haciendo que el sistema no las despliegue al procesar un archivo.

Podemos usar el comando `#print` para mostrar por pantalla tanto la prueba de un teorema como la definición de un término.

2.2.3 Lógica proposicional

Como sucede en el Cálculo de Construcciones, la teoría de tipos dependientes de Lean cuenta desde el principio con las operaciones de λ-abstracción y de aplicación, que se corresponden, respectivamente, con las reglas de introducción y eliminación del constructor `→`. El resto de conectivas proposicionales están definidas en la librería de Lean, en el archivo `Prelude.core`, y cada una de ellas viene con sus reglas de introducción y eliminación.

Veremos en la sección 2.3.3 que todas estas conectivas lógicas son instancias de *tipos inductivos*. Por ahora nos limitamos a describir su uso.

Conjunción

La expresión `And.intro h1 h2` construye una prueba de `p ∧ q` a partir de las pruebas `h1 : p` y `h2 : q`. Nos referimos a `And.intro` como la regla de `∧`-introducción.

```
variable (p q : Prop)

example (hp : p) (hq : q) : p ∧ q := And.intro hp hq
```

La expresión `And.left h` crea una prueba de `p` a partir de una prueba `h: p ∧ q`, y se conoce como la regla de `∧`-eliminación izquierda. Análogamente, `And.right h` es una prueba de `q`.

```
variable (p q : Prop)

example (h : p ∧ q) : q ∧ p :=
  And.intro (And.right h) (And.left h)
```

Nótese que las reglas de introducción y eliminación de la conjunción `∧` son completamente análogas a la función `fun x y => (x,y)` y a las proyecciones del producto cartesiano `×`. La similitud entre `∧` y `×` es otra instancia de la correspondencia de Curry-Howard. Sin embargo, a diferencia de lo que sucede con la implicación y el constructor del tipo de funciones, que comparten operador `→`, los operadores `∧` y `×` se tratan de forma separada en Lean. Por ejemplo, dados `p q : Prop`, `hp : p` y `hq : q`, el tipo de `And.intro hp hq` es `p ∧ q : Prop`, mientras que `Prod hp hq` es de tipo `p × q : Type`.

Disyunción

La expresión `Or.intro_left q hp` crea una prueba de `p ∨ q` a partir de una prueba `hp : p`. Análogamente, `Or.intro_right p hq` crea una prueba de la misma proposición a partir de una prueba `hq : q`. Estos son las reglas de `∨`-introducción.

```
variable (p q : Prop)
example (hp : p) : p ∨ q := Or.intro_left q hp
example (hq : q) : p ∨ q := Or.intro_right p hq
```

La regla de `∨`-eliminación es ligeramente más complicada. La idea es que podemos obtener una prueba de `r` a partir de `p ∨ q`, probando que `r` se sigue de `p` y que `r` se sigue de `q`. Dicho de otra forma, la expresión `Or.elim` es de tipo `∀ {p q r : Prop}, p ∨ q → (p → r) → (q → r) → r` (ver sección 2.2.4 sobre la cuantificación universal).

En la mayoría de casos el primer argumento de `Or.intro_left` y `Or.intro_right` puede ser inferido por Lean, así que Lean proporciona las expresiones `Or.inl` y `Or.inr`, que funcionan como abreviaturas de `Or.intro_left _` o `Or.intro_right _`.

```
variable (p q : Prop)

example (h : p ∨ q) : q ∨ p :=
  Or.elim h (fun hp => Or.inr hp) (fun hq => Or.inl hq)
```

Negación y falsedad

La falsedad viene representada en Lean por la proposición `False`, que es, como en las teorías vistas en el capítulo 1, un tipo vacío (una proposición sin ninguna prueba válida). Así, la negación `¬p` de una proposición se define como `p → False`. Por lo tanto, la expresión `hnp hp` produce una prueba de `False` a partir de las pruebas `hp : p` y `hnp : ¬p`.

```
variable (p q : Prop)

example (hpq : p → q) (hnq : ¬q) : ¬p :=
  fun hp : p => hnq (hpq hp)
```

La conectiva `False` no tiene reglas de introducción (es un tipo vacío), y tiene una única regla de eliminación, `False.elim`, que expresa el hecho de que se puede probar cualquier cosa a partir de una falsedad. El tipo de `False.elim` es `∀ {p : Prop}, False → p` (ver sección 2.2.4 sobre la cuantificación universal).

```
variable (q : Prop)

example (hp : p) (hnp : ¬p) : q := False.elim (hnp hp)
```

Por el contrario, `True` tiene únicamente una regla de introducción: `True.intro : True`. Dicho de otra forma, la proposición `True` simplemente es cierta y tiene una prueba canónica: `True.intro`.

Equivalencia lógica

La expresión `Iff.intro h1 h2` genera una prueba de `p ↔ q` a partir de `h1 : p → q` y `h2 : q → p`. Las expresiones `Iff.mp h` y `Iff.mpr` producen pruebas de `p → q` y `q → p` a partir de `h : p ↔ q`.

```
variable (p q : Prop)

example (hpq : p ↔ q) : q ↔ p :=
  Iff.intro (Iff.mpr hpq) (Iff.mp hpq)
```

Principio del tercero excluso

Todas las reglas de introducción y eliminación que hemos visto hasta ahora son constructivas, es decir, reflejan una interpretación computacional de las conectivas lógicas dentro del paradigma de las proposiciones como tipos. La lógica clásica añade además de las conectivas anteriores, el principio del tercero excluso: `p ∨ ¬p`. Para usarlo hay que abrir el *namespace* `Classical`:

```
open Classical

variable (p : Prop)
#check em p          -- p ∨ ¬p
```

Intuitivamente, el `∨` constructivista es más fuerte que el clásico: afirmar `p ∨ q` equivale a saber cuál de las dos proposiciones es cierta. Sin embargo, un matemático clásico está dispuesto a afirmar `p ∨ ¬p` sin saber en qué caso se encuentra.

Veremos en la sección 2.5.1 que hay situaciones en lógica constructivista en las que el principio del tercero excluso está permitido, y en las que Lean permite el uso del razonamiento clásico sin necesidad de acudir al *namespace* `Classical`.

2.2.4 Cuantificadores e igualdad

El cuantificador universal

Si `α` es un tipo, podemos representar un predicado unario sobre `α` como un objeto `p : α → Prop`. Así, para cada `x : α`, `p x` denota la proposición "`x` satisface `p`". Análogamente, un objeto `r : α → α → Prop` representa una relación binaria en `α`.

La cuantificación universal `∀ x : α, p x` no es más que azúcar sintáctico para el tipo de funciones dependientes `(x : α) → p x`. Así, las reglas de introducción y eliminación del cuantificador universal son las mismas que las de las funciones dependientes: la `λ`-abstracción y la aplicación.

```
example (α : Type) (p q : α → Prop) :
    (∀ x : α, p x ∧ q x) → ∀ y : α, p y :=
  fun h : ∀ x : α, p x ∧ q x =>
  fun y : α =>
  (h y).left      -- 't.left' y 'And.left t' son equivalentes
                  -- si 't' es una conjunción
```

Es importante recordar que las expresiones que difieren en los nombres de las variables ligadas son equivalentes, así que la anterior prueba podría haberse escrito, por ejemplo, de esta otra forma:

```
example (α : Type) (p q : α → Prop) :
    (∀ x : α, p x ∧ q x) → ∀ x : α, p x :=
  fun h : ∀ x : α, p x ∧ q x =>
  fun z : α =>
  (h z).left
```

La regla de tipado de los tipos de funciones dependientes (y, en particular, del cuantificador universal) es la principal diferencia entre `Prop` y el resto de tipos. Supongamos que tenemos `α : Sort i` y `β : Sort j`, donde la expresión `β` puede depender de una variable `x : α`. Entonces `(x : α) → β` es de tipo `Sort (imax i j)`, donde `imax i j` es 0 si `j` es 0 y el máximo de `i` y `j` en caso contrario.

La idea es la siguiente: si `j` no es 0, entonces la expresión `(x : α) → β` "vive" en el mayor de los universos donde viven `α` y `β`. Sin embargo, si `j` es 0, es decir, si

`β` es una proposición, entonces `(x : α) → β` también lo es. Es decir, el universo `Prop` de las proposiciones es *impredicativo*.

Igualdad

En la sección 2.3.6 veremos cómo se define la igualdad en términos de la infraestructura lógica de Lean. Por ahora nos limitamos a describir cómo se usa.

Por supuesto, una propiedad fundamental de la igualdad es que se trata de una relación de equivalencia:

```
universe u

#check @Eq.refl.{u}   -- ∀ {α : Sort u} (a : α), a = a
#check @Eq.symm.{u}   -- ∀ {α : Sort u} {a b : α}, a = b → b = a
#check @Eq.trans.{u}  -- ∀ {α : Sort u} {a b c : α},
                      -- a = b → b = c → a = c
```

La reflexividad es más potente de lo que puede parecer. Recordemos que los términos del Cálculo de Construcciones tienen una interpretación computacional, y que Lean trata como iguales a dos términos cuya reducción es la misma. Como consecuencia, puede usarse la propiedad reflexiva de la igualdad para probar algunas identidades "no triviales":

```
variable (α β : Type)

example (f : α → β) (a : α) : (fun x => f x) a = f a :=
  Eq.refl _   -- '_' es 'f a'
example (a : α) (b : β) : (a, b).1 = a :=
  Eq.refl _   -- '_' es 'a'
example : 2 + 3 = 5 :=
  Eq.refl _   -- '_' es '5'
```

De hecho, esta funcionalidad es tan importante que la librería define una notación `rfl` para `Eq.refl _`.

La igualdad es más que una relación de equivalencia. Tiene la propiedad de que podemos sustituir iguales por iguales en afirmaciones sin alterar la veracidad de las mismas. Esto se conoce como principio de sustitución, y en Lean viene representado por el término `Eq.subst`.

```
example (α : Type) (a b : α) (p : α → Prop)
    (h1 : a = b) (h2 : p a) : p b :=
  Eq.subst h1 h2

example (α : Type) (a b : α) (p : α → Prop)
    (h1 : a = b) (h2 : p a) : p b :=
  h1 ▸ h2
```

El triángulo del segundo ejemplo[3] es una versión más potente de `Eq.subst` construida a partir de `Eq.subst` y `Eq.symm` que funciona en situaciones más generales.

El cuantificador existencial

La cuantificación existencial puede escribirse en Lean como `exists x : α, p x` o `∃ x : α, p x`. Ambas expresiones son abreviaturas para la expresión `Exists (fun x : α => p x)` definida en la librería de Lean.

La librería incluye una regla de introducción y una regla de eliminación. La regla de introducción es directa, para probar `∃ x : α, p x` basta con dar un elemento `t : α` junto con una prueba `h : p t`:

```
example : ∃ x : Nat, x > 0 :=
  have h : 1 > 0 := Nat.zero_lt_succ 0
  Exists.intro 1 h

#check @Exists.intro  -- ∀ {α : Sort u_1} {p : α → Prop}
                      -- (w : α), p w → Exists p
```

Nótese que `Exists.intro` tiene argumentos implícitos; Lean tiene que inferir el predicado `p : α → Prop`. Esto no es un problema trivial. Por ejemplo, si tenemos `hg : g 0 0 = 0` y escribimos `Exists.intro 0 hg`, hay varias posibilidades para el predicado `p`, correspondientes a las proposiciones `∃ x, g x x = x`, `∃ x, g x x = 0`, `∃ x, g x 0 = x`, etc. Lean usa el contexto para inferir cuál es la opción apropiada. En caso de ambigüedad en el contexto, o si simplemente queremos hacer la prueba más legible, podemos usar el comando `show p from h`[4]

[3] Puede escribirse en Lean usando `\t`.

[4] No debe confundirse `show` con `have`. El comando `have` se puede usar para introducir un término al contexto (consultar la sección A.1 del apéndice para más detalles).

para indicar a Lean cuál es la proposición `p` que queremos probar con `h`.

```
example : ∃ x : Nat, x > 0 :=
  have h : 1 > 0 := Nat.zero_lt_succ 0
  show ∃ x : Nat, x > 0 from Exists.intro 1 h
```

La regla de `∃`-eliminación es similar a la de `∨`-eliminación; nos permite probar una proposición `q` a partir de `∃ x : α, p x`, probando que `q` se sigue de `p x` para cualquier `x : α`. Por ejemplo:

```
variable (α : Type) (p q : α → Prop)

example (h : ∃ x, p x ∧ q x) : ∃ x, q x ∧ p x :=
  Exists.elim h
    (fun w =>
     fun hw : p w ∧ q w =>
       show ∃ x, q x ∧ p x
         from Exists.intro w (And.intro hw.right hw.left))
```

La afirmación `∃ x : α, p x` puede verse como una disyunción entre las proposiciones `p a` cuando `a` recorre los elementos de `α`. De hecho, la proposición existencial `∃ x : α, p x` es muy similar al tipo `Σ x : α, p x` (otra instancia de la correspondencia de Curry-Howard). Igual que con el cuantificador universal y los productos dependientes, la diferencia en Lean es que, dados `a : α`, `p : α → Prop` y `h : p a`, el término `Exists.intro a h` es de tipo `(∃ x : α, p x) : Prop` y `Sigma.mk a h` es de tipo `(Σ x : α, p x) : Type`.

De igual forma que el `∨` constructivista es más fuerte que el `∨` clásico, el `∃` constructivista también es más fuerte que el clásico. Desde la perspectiva constructivista, saber que no es cierto que para todo `x` se cumple `p x` no implica que haya algún `x` que satisface `¬ p x`. Esto se debe a que, en la lógica constructivista, para afirmar `∃ x : α, p x` necesitamos conocer un `x : α` tal que `p x`.

2.3 Tipos inductivos

Hemos visto que la base fundacional de Lean incluye los universos `Prop`, `Type 0`, `Type 1`... y permite la construcción de funciones dependientes `(x : α) → β`. Resulta que todos los tipos, exceptuando los universos, y todos los constructores aparte de `→` son instancias de una familia general: los tipos inductivos.

En Lean, igual que en las Familias Inductivas (ver Sección 1.4 del anterior capítulo), un tipo inductivo se define esencialmente a partir de una lista de constructores. En Lean, la sintaxis para ello tiene la siguiente forma:

```
inductive Foo where
  | constructor₁ : ... → Foo
  | constructor₂ : ... → Foo
  ...
  | constructorₙ : ... → Foo
```

Cada constructor especifica una forma de crear nuevos objetos de tipo `Foo`, pudiendo hacerse a partir de valores previamente creados. Los elementos de `Foo` son los que se pueden generar mediante los constructores.

Es decir, los constructores se corresponden con las reglas de introducción de los tipos inductivos tal y como se describen en el capítulo 1. Veremos que, igual que en las Familias Inductivas, Lean genera de forma automática a partir de la definición de un tipo inductivo sus reglas de eliminación. Las reglas de eliminación corresponden al principio de recursión (o inducción) asociado al tipo definido.

Las conectivas lógicas que hemos visto en la sección anterior son en realidad ejemplos de tipos inductivos. En esta sección se proporcionan más ejemplos de tipos inductivos y se presentan generalizaciones del esquema básico anterior, como los tipos inductivos definidos mutuamente o las familias inductivas.

2.3.1 Enumeraciones o tipos enumerados

La categoría más sencilla de tipos inductivos es la de los tipos con una lista finita de elementos. Son tipos inductivos cuyos constructores no toman argumentos. Por ejemplo:

```
inductive Weekday where
  | sunday : Weekday
  | monday : Weekday
  | tuesday : Weekday
  | wednesday : Weekday
  | thursday : Weekday
  | friday : Weekday
  | saturday : Weekday
```

Debe pensarse en `sunday`, `monday`, ... , `saturday` como elementos distinguidos de `Weekday`, sin ninguna otra propiedad. La regla de eliminación `Weekday.rec` (también conocida como *recursor*) corresponde a una definición por casos, y se genera automáticamente a partir de la definición de `Weekday`. En este caso, la regla de eliminación nos permite definir una función en `Weekday` asignando un valor a cada uno de los constructores. La idea es que un tipo inductivo es aquel generado exhaustivamente por sus constructores y no tiene más elementos más allá de los construidos de esa forma.

La expresión `match`, definida mediante el recursor `Weekday.rec`, nos permite definir funciones sobre `Weekday` de forma intuitiva:

```
open Weekday  -- El namespace 'Weekday' se genera automáticamente
              -- al definir el tipo inductivo con el mismo nombre
def numberOfDay (d : Weekday) : Nat :=
  match d with
  | sunday    => 1
  | monday    => 2
  | tuesday   => 3
  | wednesday => 4
  | thursday  => 5
  | friday    => 6
  | saturday  => 7

set_option pp.all true  -- Para mostrar universos y
                        -- argumentos implícitos
#print numberOfDay
-- ... numberOfDay.match_1
#print numberOfDay.match_1
-- ... Weekday.casesOn ...
#print Weekday.casesOn
-- ... Weekday.rec ...
```

```
#check @Weekday.rec
/-
@Weekday.rec.{u}
 : {motive : Weekday → Sort u} →
    motive Weekday.sunday →
    motive Weekday.monday →
    motive Weekday.tuesday →
    motive Weekday.wednesday →
    motive Weekday.thursday →
    motive Weekday.friday →
    motive Weekday.saturday →
    (t : Weekday) → motive t
-/
```

Nótese que, bajo el paradigma de las proposiciones como tipos, podemos usar `match` para probar teoremas, además de definir funciones. Dicho de otra forma, las pruebas por casos no son más que un ejemplo de definición por casos, donde lo que se definen son pruebas en lugar de datos.

El tipo `Bool` de la librería de Lean es una instancia de tipo definido por enumeración:

```
inductive Bool where
  | false : Bool
  | true  : Bool
```

2.3.2 Constructores con argumentos no recursivos

En general, una construcción puede depender de otros datos. Consideremos las siguientes definiciones del producto cartesiano y del tipo suma de la librería de Lean:

```
inductive Prod (α : Type u) (β : Type v) where
  | mk : α → β → Prod α β

inductive Sum (α : Type u) (β : Type v) where
  | inl : α → Sum α β
  | inr : β → Sum α β
```

En el caso del producto cartesiano, tenemos un constructor, `Prod.mk`, que toma dos argumentos de tipo `α` y `β`. Por lo tanto, si queremos definir una función en `Prod α β` asumimos que la entrada tiene la forma `Prod.mk a b` y definimos la salida en términos de `a` `b`. Esto es precisamente lo que permite la regla de eliminación `@Prod.rec`. De nuevo, la expresión `match` nos permite usar la regla de eliminación más cómodamente:

```
def fst {α : Type u} {β : Type v} (p : Prod α β) : α :=
  match p with
  | Prod.mk a b => a

def snd {α : Type u} {β : Type v} (p : Prod α β) : β :=
  match p with
  | Prod.mk a b => b
```

El tipo suma, por otro lado, tiene dos constructores, `inl` e `inr`, cada uno con un argumento. Por lo tanto, a la hora de definir una función sobre `Sum α β` debemos distinguir dos casos: o bien la entrada es de la forma `inl a`, en cuyo caso definimos la salida en términos de `a`, o bien la entrada es de la forma `inr b`, en cuyo caso damos la salida en función de `b`. Por ejemplo:

```
def sum_example (s : Sum Nat Nat) : Nat :=
  match s with
  | inl n => 2 * n
  | inr n => 2 * n + 1
```

En general, un tipo con múltiples constructores es disyuntivo, mientras que un constructor con múltiples argumentos introduce información conjuntiva[5]. En general, un tipo inductivo arbitrario puede incluir ambas opciones.

Como ya hemos visto, el tipo `Sigma` no es más que una generalización del producto cartesiano donde el tipo de la segunda componente puede depender del valor de la primera:

```
inductive Sigma {α : Type u} (β : α → Type v) where
  | mk : (a : α) → β a → Sigma β
```

[5]Los tipos inductivos no recursivos con un único constructor son tan frecuentes que reciben el nombre de *estructuras*. Las estructuras se definen en Lean mediante el comando `structure` (consultar la sección A.2 del apéndice para más detalles)

Hay otros dos ejemplos importantes de tipos inductivos sin argumentos recursivos en la librería de Lean:

```
inductive Option (α : Type u) where
  | none : Option α
  | some : α → Option α

inductive Inhabited (α : Type u) where
  | mk : α → Inhabited α
```

La teoría de tipos dependientes no incluye, inicialmente, la noción de función parcial; todo elemento del tipo `(a : α) → β` debe tener una salida para cada entrada. El tipo `Option` proporciona una forma de representar funciones parciales. Un elemento de `Option β` es, o bien `none`, o bien `some b` para algun `b : β`. Así, un elemento de `α → Option β` puede verse como una función parcial de `α` en `β`.

Por otro lado, un elemento de `Inhabited α` no es más que un testigo del hecho de que `α` contiene al menos un elemento. Veremos en la sección 2.5 que `Inhabited` es un ejemplo de *clase de tipos*.

2.3.3 Proposiciones definidas inductivamente

Los tipos definidos inductivamente pueden vivir en cualquier universo, incluido el primero, `Prop`. Así es precisamente como se definen las conectivas lógicas:

```
inductive False : Prop

inductive True : Prop where
  | intro : True

inductive And (a b : Prop) : Prop where
  | intro : a → b → And a b

inductive Or (a b : Prop) : Prop where
  | inl : a → Or a b
  | inr : b → Or a b

inductive Exists {α : Sort u} (p : α → Prop) : Prop where
  | intro (w : α) (h : p w) : Exists p
```

Estas definiciones dan lugar a las reglas de introducción y eliminación que vimos en la sección 2.2.

Es importante notar que hay ciertas reglas internas que determinan qué tipos pueden obtenerse a partir de objetos de un tipo inductivo usando el principio de eliminación, es decir, qué tipos pueden usarse como llegada de un recursor. En términos generales, lo que caracteriza a los tipos inductivos que viven en `Prop` es que solo pueden obtenerse a partir de ellos otros tipos en `Prop`. Esto es consistente con el hecho de que si `p : Prop`, entonces un elemento `hp : p` no contiene información. Hay una excepción a esta regla, que veremos en la sección 2.3.7.

Las definiciones de `False`, `True`, `And`, `Or` y `Exists` son totalmente análogas a las de `Empty`, `Unit`, `Prod`, `Sum` y `Sigma`. La diferencia es que el primer grupo da lugar a elementos de `Prop`, mientras que el segundo da lugar a elementos de `Type u` para algún `u`.

Es un buen momento para mencionar otro tipo inductivo, denotado `{x : α // p x}`, que es una especie de híbrido entre `∃ x : α, p x` y `Σ x : α, p x`.

```
inductive Subtype {α : Type u} (p : α → Prop) where
  | mk : (x : α) → p x → Subtype p
```

La idea es que `{x : α // p x}` (o, equivalentemente, `Subtype (fun x : α => p x)` denota al conjunto de elementos de `α` que tienen la propiedad `p`.

La diferencia fundamental entre `Exists` y `Subtype` es que `Exists` siempre devuelve un tipo del universo `Prop`, mientras que `Subtype` vive en `Type u` para algún `u`. Por lo tanto, la regla de eliminación de `Subtype` permite generar elementos de cualquier tipo, mientras que la de `Exists` solo permite obtener pruebas de proposiciones. Como consecuencia, a partir de un elemento `h : {x : α // p x}` es posible obtener un término `a : α` junto con una prueba `ha : p a`. Esto no es posible si partimos de una prueba de `∃ x : α, p x`.

2.3.4 Los números naturales

Hasta ahora, los argumentos de los constructores que hemos visto han sido tipos previamente definidos. Sin embargo, los constructores también pueden actuar sobre

el tipo que se está definiendo. Los números naturales son un ejemplo canónico de esto:

```
inductive Nat where
  | zero : Nat
  | succ : Nat → Nat
```

El recursor `Nat.rec` generado por Lean permite definir una función dependiente `f` sobre `Nat`, es decir, un elemento `f : (n : Nat) → motive n` para algún `motive : Nat → Sort u`.

```
#check @Nat.rec
/-
{motive : Nat → Sort u}
→ motive Nat.zero
→ ((n : Nat) → motive n → motive (Nat.succ n))
→ (t : Nat) → motive t
-/
```

El argumento implícito `motive` se corresponde con el argumento C de las reglas de eliminación que vimos en la sección 1.34 del capítulo 1: el *motivo* de la recursión. El segundo y tercer argumentos corresponden a las *premisas menores*, mientras que el último argumento `t : Nat` es la *premisa mayor*; la entrada de la función definida recursivamente.

Como hemos visto en las anteriores secciones, la expresión `match` nos permite definir funciones en `Nat` sin necesidad de recurrir de forma directa a `Nat.rec`:

```
def add (m n : Nat) : Nat :=
  match n with
  | Nat.zero   => m
  | Nat.succ n => Nat.succ (add m n)
```

El principio de inducción es un caso particular del principio de recursión donde el término `motive n` es un elemento de `Prop`. Por ejemplo:

```
instance : Add Nat where  -- para permitir la notación '+'
  add := add              -- para la definición 'add'

theorem add_succ :
  ∀ (m n : Nat), m + Nat.succ n = Nat.succ (m + n) :=
    fun _ _ => rfl  -- es una igualdad definicional
```

```
theorem zero_add (n : Nat) : Nat.zero + n = n :=
  Nat.rec (motive := fun x => Nat.zero + x = x)
    rfl  -- igualdad definicional
    (fun n ih =>
      have h : Nat.succ (Nat.zero + n) = Nat.succ n := ih.symm ▸ rfl
      h ▸ add_succ Nat.zero n)
    n
```

2.3.5 Otros tipos de datos recursivos

Para cualquier tipo `α`, el tipo `List α` de listas de elementos de `α` está definido en la librería:

```
inductive List (α : Type u) where
| nil  : List α
| cons : α → List α → List α

namespace List

def append (as bs : List α) : List α :=
  match as with
  | nil       => bs
  | cons a as => cons a (append as bs)

end List
```

También podemos definir, por ejemplo, el tipo de los árboles binarios:

```
inductive BinaryTree where
  | leaf : BinaryTree
  | node : BinaryTree → BinaryTree → BinaryTree
```

2.3.6 Familias de tipos inductivos

Hasta ahora hemos descrito cómo los constructores y recursores de Lean se corresponden con las reglas de introducción y eliminación que vimos en la sección 1.4 del capítulo 1. ¿Cuál es el análogo en Lean a las reglas de formación de tipos?

En Lean, la definición de una *familia* de tipos inductivos tiene la siguiente forma:

```
inductive Foo (a :: α) : (b :: β) → Sort u where
  | constructor₁ : ... → Foo ...
  | constructor₂ : ... → Foo ...
  ...
  | constructorₙ : ... → Foo ...
```

A diferencia de las definiciones inductivas que hemos visto hasta ahora, que construían un tipo de algún `Sort u` en función de una secuencia de *parámetros* `a :: α`, esta versión más general construye una función `(b :: β) → Sort u`, es decir, construye una familia de tipos en `Sort u` con *índices* en `(b :: β)`.

Observemos que en la definición anterior, la declaración de tipos `(a :: α) : (b :: β) → Sort u` corresponde precisamente con la regla de formación de `Foo`.

Un ejemplo de familia de tipos es la definición de `List' α n`, el tipo de las listas de elementos de `α` y de longitud `n`:

```
inductive List' (α : Type u) : Nat → Type u where
  | nil  : List' α 0
  | cons : α → {n : Nat} → List' α n → List' α (n+1)
```

Igual que sucedía en las Familias Inductivas, la diferencia entre los parámetros y los índices es que los primeros deben mantenerse fijos a lo largo de toda la definición inductiva.

Un ejemplo fundamental de familia de tipos inductivos en Lean es la definición de igualdad:

```
inductive Eq {α : Sort u} (a : α) : α → Prop where
  | refl : Eq a a
```

Nótese que, dados `α : Sort u` y `a : α`, `Eq a a` es el único tipo habitado de la familia de tipos `Eq a x` (donde `x` recorre `α`). La regla de eliminación generada por Lean es la siguiente:

```
universe u v

#check (@Eq.rec :
    {α : Sort u} → {a : α} → {motive : (x : α) → Eq a x → Sort v}
    → motive a rfl → {b : α} → (h : Eq a b) → motive b h)
```

El recursor `Eq.rec` se usa para definir la regla de sustitución:

```
theorem subst {α : Type u} {a b : α} {p : α → Sort v}
    (h₁ : Eq a b) (h₂ : p a) : p b :=
  Eq.rec (motive := fun x _ => p x) h₂ h₁
```

2.3.7 Detalles axiomáticos

Hemos visto que las definiciones de tipos inductivos pueden depender de parámetros e índices. Los argumentos de cada constructor pueden depender de tipos previamente definidos, de los parámetros e índices y de la familia inductiva `C` que está siendo definida. El único requisito es que, si un constructor tiene argumentos recursivos, entonces la familia inductiva `C` debe aparecer en ellos de forma *estrictamente positiva*. Esto significa que los argumentos recursivos deben ser funciones dependientes donde la familia `C` puede aparecer solo como tipo final, y donde los índices de la familia `C` pueden expresarse en términos de constantes y argumentos previos.

Como los tipos inductivos viven en `Sort u` para algún `u`, es razonable preguntarse en qué universo `u` se puede instanciar un tipo inductivo. Vamos a ver a continuación cuáles son las restricciones para el universo en función de los constructores de la familia.

El tipo de cada constructor `c` en la definición de una familia `C` de tipos inductivos tiene la forma

```
c : (a : α) → (b : β[a]) → C a p[a,b]
```

donde `a` es una secuencia de parámetros, `b` es la secuencia de argumentos del constructor `c` y `p[a,b]` es la secuencia de índices que determina el tipo de la familia al que pertenece el objeto construido por `c`.

Las restricciones para el universo en el que puede vivir la familia `C` dependen de si se define en `Prop` o no. Si el tipo inductivo no se define en `Prop`, entonces el universo `u` en el que vive debe satisfacer lo siguiente:

> *Para cada constructor* `c` *como el descrito antes y cada* β_k`[a]` *en la secuencia* `β[a]`*, si* β_k`[a] : Sort v`*, entonces* `u` $\geq$ `v`*.*

Si el tipo inductivo se define en `Prop`, entonces no hay ninguna restricción para el universo de los argumentos de los constructores.

Sin embargo, como ya hemos visto, cuando un tipo inductivo se define en `Prop`, el tipo de llegada del `motive` del correspondiente principio de eliminación también debe ser `Prop`. Esta regla tiene una excepción: el `motive` de la regla de eliminación puede vivir en un universo arbitrario aunque el tipo inductivo viva en `Prop` si hay un único constructor y cada argumento del constructor o bien está en `Prop`, o bien es un índice. La justificación intuitiva de esta excepción es que, en este caso, la eliminación no hace uso de información que no venga ya dada por el hecho de que el tipo del argumento está habitado. Este caso especial se conoce como *eliminación unitaria*.

Ya hemos visto un ejemplo de eliminación unitaria: el recursor de la igualdad definida inductivamente. Por ejemplo, usar un elemento `h : Eq a b` para transformar un elemento de `p a` en uno de `p b`, incluso aunque `p a` y `p b` sean de tipos de universos arbitrarios y no necesariamente proposiciones. La idea es que la transformación no genera nueva información, sino que reinterpreta la información que ya tenemos sabiendo que `Eq a b` está habitado. La eliminación unitaria se utiliza también en la recursión bien fundamentada, que veremos en la sección 2.4.3.

2.3.8 Tipos mutuamente inductivos y tipos inductivos anidados

Consideramos ahora dos generalizaciones de los tipos inductivos que suelen ser útiles. Lean procesa estas definiciones más generales, define tipos inductivos auxiliares siguiendo el esquema descrito en las secciones anteriores, y utiliza estos tipos auxiliares para hacer uso de los que realmente han sido definidos por el usuario.

Lean admite los tipos inductivos definidos mutuamente. La idea es que podemos definir dos o más tipos inductivos simultáneamente, de forma que cada uno de ellos

puede hacer referencia a los otros. Un ejemplo prototípico de inducción mutua es la siguiente definición de paridad:

```
mutual
  inductive Even : Nat → Prop where
    | even_zero : Even 0
    | even_succ : (n : Nat) → Odd n → Even (n + 1)

  inductive Odd : Nat → Prop where
    | odd_succ : (n : Nat) → Even n → Odd (n + 1)
end
```

Puede usarse la inducción mutua para definir, por ejemplo, el tipo de los árboles finitos con nodos etiquetados por elementos de `α`:

```
mutual
    inductive Tree (α : Type u) where
      | node : α → TreeList α → Tree α

    inductive TreeList (α : Type u) where
      | nil  : TreeList α
      | cons : Tree α → TreeList α → TreeList α
end
```

Sin embargo, es tedioso trabajar con esta definición. Sería mucho más conveniente si la lista de subárboles de cada nodo viniese dada por el tipo `List (Tree α)`, especialmente porque la librería de Lean ya incluye una serie de funciones y teoremas para trabajar con listas. Puede probarse que el tipo `TreeList α` es *isomorfo* a `List (Tree α)`, pero traducir resultados a través del isomorfismo es tedioso.

En realidad, Lean nos permite definir exactamente el tipo inductivo que queremos:

```
inductive Tree (α : Type u) where
  | mk : α → List (Tree α) → Tree α
```

Nótese que la recursión en esta definición no cumple los requisitos descritos en la sección anterior: el tipo `Tree α` definido no aparece de forma estrictamente en los argumentos del constructor `mk`, pues está anidado dentro del constructor `List`. Sin embargo, Lean genera automáticamente el isomorfismo entre `TreeList α` y `List (Tree α)` y define los constructores de `Tree` en términos del isomorfismo. La definición anterior es un ejemplo de tipo inductivo anidado.

2.4 Inducción y recursión

2.4.1 Pattern matching

Sin entrar en detalles, damos a continuación algunos ejemplos de *pattern matching*, una funcionalidad de Lean (y de otros demostradores automáticos) que permite definir funciones (y probar teoremas) sobre tipos inductivos simplemente especificando las ecuaciones recursivas que debe cumplir la función.

```
def sub2 : Nat → Nat
  | 0   => 0
  | 1   => 0
  | x+2 => x

def f1 : Nat → Nat → Option Nat
  | 0, _  => some 1   -- Si el primer argumento es 0
  | _, 0  => some 2   -- Si el segundo argumento es 0
  | _, _  => none     -- En cualquier otro caso

def bar : Nat → List Nat → Bool → Nat
  | 0,   _,       false => 0
  | 0,   b :: _, _     => b
  | 0,   [],      true  => 7
  | a+1, [],      false => a
  | a+1, [],      true  => a + 1
  | a+1, b :: _, _     => a + b
```

El procesador de ecuaciones de Lean se encarga de transformar estas definiciones en recursores básicos (entre los que se encuentran las reglas de eliminación que hemos visto en secciones anteriores) cuya corrección es comprobada posteriormente. En las siguientes secciones describiremos, respectivamente:

- las definiciones por recursión estructural
- las definiciones por recursión bien fundamentada
- las definiciones por recursión mutua

2.4.2 Recursión e inducción estructural

En general, el procesador de ecuaciones toma entradas de la forma

```
def foo (a : α) : (b : β[a]) → γ[a,b]
  | [patterns₁] => t₁
  ...
  | [patternsₙ] => tₙ,
```

donde `a : α` es una secuencia de parámetros, `b : β[a]` es la secuencia de argumentos sobre los que se aplica el *pattern matching* y `γ[a,b]` es un tipo. Los términos `t₁, ..., tₙ` pueden hacer uso de los parámetros `a` y de cualquiera de las variables introducidas en los correspondientes patrones. Lo que hace posible la recursión e inducción es que estos términos también pueden incluir llamadas a la función que se está definiendo.

En esta sección estudiamos la *recursión estructural*, donde los argumentos de `foo` que aparecen en los términos `t₁, ..., tₙ` son subtérminos de los patrones `patterns₁, ..., patternsₙ`. La idea es que esos argumentos sean "más pequeños estructuralmente", y por tanto aparezcan en una etapa más temprana del tipo inductivo.

Algunos ejemplos de recursión estructural son:

```
def fib : Nat → Nat
  | 0   => 1
  | 1   => 1
  | n+2 => fib (n+1) + fib n

def append : List α → List α → List α
  | [],    bs => bs
  | a::as, bs => a :: append as bs

def listAdd [Add α] : List α → List α → List α
  | [],      _       => []
  | _,       []      => []
  | a :: as, b :: bs => (a + b) :: listAdd as bs
```

Para tratar con la recursión estructural, el procesador de ecuaciones utiliza la denominada recursión *course-of-values*, por medio de las constantes `below` y `brecOn` que Lean genera automáticamente para cada tipo inductivo. A continuación se muestran los tipos de las constantes `Nat.below` y `Nat.brecOn`:

```
variable (motive : Nat → Type u)

#check (@Nat.below motive : Nat → Type u)  -- Nat → Type (max 0 u)
```

```
#reduce @Nat.below motive (3 : Nat)   -- PProd (PProd (motive 2)
                                      -- (PProd (PProd (motive 1)
                                      -- (PProd (PProd (motive 0)
                                      -- PUnit) PUnit)) PUnit))
                                      -- PUnit
#check (@Nat.brecOn motive :
    (n : Nat) → ((n : Nat) → @Nat.below motive n → motive n)
    → motive n)
```

El tipo `@Nat.below motive (3 : Nat)` es una estructura de datos que almacena elementos de `motive 0`, `motive 1` y `motive 2`. La recursión *course-of-values* está implementada mediante `Nat.brecOn`; nos permite definir el valor de una función dependiente de tipo `(n : Nat) → motive n` en términos de una entrada `n` y de todos los valores anteriores de la función.

El uso de la recursión *course-of-values* es una de las técnicas que Lean utiliza para justificar que una función termina. Esto no afecta al generador de código, que compila las funciones recursivas como los compiladores de otros lenguajes de programación. De hecho, `#eval fib n`, que pide al generador de código que evalúe la función `fib` en `n`, es exponencial en `n`, mientras que `#reduce fib n` es eficiente porque usa la definición basada en el principio `brecOn`:

```
def fib : Nat → Nat
  | 0   => 1
  | 1   => 1
  | n+2 => fib (n+1) + fib n

-- #eval fib 50 -- lento
#reduce fib 50  -- rápido

#print fib
/-
def fib : Nat → Nat :=
fun x =>
  Nat.brecOn x fun x f =>
    (match (motive := (x : Nat) → Nat.below x → Nat) x with
      | 0 => fun x => 1
      | 1 => fun x => 1
      | Nat.succ (Nat.succ n) => fun x =>
        x.fst.fst + x.fst.snd.fst.fst)
      f
-/
```

2.4.3 Recursión e inducción bien fundamentada

Cuando no puede usarse recursión estructural, puede probarse la terminación de una función usando recursión bien fundamentada. Necesitamos para ello una relación bien fundamentada y una prueba de que cada aplicación recursiva de la función definida decrece con respecto a la relación. La teoría de tipos dependientes es lo suficientemente expresiva como para codificar y justificar casos de recursión bien fundamentada.

La librería de Lean define dos predicados, `Acc r a` y `WellFounded r`, donde `r` es una relación sobre un tipo `α` y `a` es un elemento de tipo `α`:

```
variable (α : Sort u)
variable (r : α → α → Prop)

#check (Acc r : α → Prop)
#check (WellFounded r : Prop)
```

Nótese que `Acc` es un predicado definido inductivamente:

```
#print Acc
/-
inductive Acc.{u} : {α : Sort u} → (α → α → Prop) → α → Prop
number of parameters: 2
constructors:
Acc.intro : ∀ {α : Sort u} {r : α → α → Prop} (x : α),
  (∀ (y : α), r y x → Acc r y) → Acc r x
-/
```

Dado `x : α`, el predicado `Acc r x` significa "`x` es accesible desde sus predecesores por medio de la relación `r`". Observemos que, en vista de la definición anterior, `Acc r x` es esencialmente equivalente a `∀ y, r y x → Acc r y`. Si interpretamos `r y x` como una relación de orden `y ≺ x`, entonces tenemos `Acc r x` (`x` es accesible por `r`) si todos los predecesores de `x` son accesibles por `r`. En particular, todo elemento `x` sin predecesores es accesible.

La idea es que, dado un tipo `α`, deberíamos poder asignar un valor a cada elemento accesible de `α` recursivamente si le hemos dado antes un valor a todos sus predecesores.

La afirmación de que `r` está bien fundamentada, denotada `WellFounded r`,

significa que todos los elementos de `α` son accesibles:

```
#print WellFounded
/-
inductive WellFounded.{u} : {α : Sort u} → (α → α → Prop) → Prop
number of parameters: 2
constructors:
WellFounded.intro : ∀ {α : Sort u} {r : α → α → Prop},
  (∀ (a : α), Acc r a) → WellFounded r
-/
```

De acuerdo con la observación anterior, si `r` es una relación bien fundamentada sobre un tipo `α`, deberíamos tener un principio de recursión bien fundamentada en `α` con respecto a la relación `r`. Esto es precisamente lo que hace la constante `WellFounded.fix` definida en la librería:

```
#check @WellFounded.fix
/-
{α : Sort u_1} → {C : α → Sort u_2} → {r : α → α → Prop} →
  (h : WellFounded r) →
  (F : (x : α) → ((y : α) → r y x → C y) → C x) →
  (x : α) → C x
-/
```

La definición de `WellFounded.fix` se basa en la regla de eliminación `Acc.rec` generada automáticamente a partir de la definición de `Acc`. La variable `C` es análoga al `motive` que ha aparecido en secciones anteriores, `h` es la hipótesis de que `r` está bien fundamentada, y la función `F` proporciona la receta recursiva para construir un elemento de tipo `C x`, suponiendo que sabemos construir elementos de `C y` para cada predecesor `y` de `x`.

Lean sabe que el orden habitual `<` sobre los números naturales está bien fundamentado. También conoce formas de construir nuevos órdenes bien fundamentados a partir de otros, por ejemplo, usando el orden lexicográfico.

La definición de división de números naturales de la librería de Lean es esencialmente la siguiente:

```
open Nat

theorem div_lemma {x y : Nat} : 0 < y ∧ y ≤ x → x - y < x :=
```

```
  fun h => sub_lt (Nat.lt_of_lt_of_le h.left h.right) h.left

def div.F (x : Nat) (f : (x₁ : Nat) → x₁ < x → Nat → Nat) (y : Nat)
    : Nat :=
  if h : 0 < y ∧ y ≤ x then
    f (x - y) (div_lemma h) y + 1
  else
    0

noncomputable def div := WellFounded.fix (measure id).wf div.F
-- Usamos el modificador 'noncomputable' porque el generador de
-- código actualmente no es compatible con 'WellFounded.fix'
#reduce div 8 2 -- 4
```

La recursión en la definición anterior se hace sobre `x`. Aquí, `div.F` juega el papel de la receta recursiva `F` descrita anteriormente. La idea es que `div.F x f : Nat → Nat` sea la función `div x ·`, suponiendo que el segundo argumento, `f`, devuelva la función `div x₁ ·` para cada `div x₁ < x`.

El sistema está diseñado para facilitar este tipo de definiciones. Acepta lo siguiente:

```
def div (x y : Nat) : Nat :=
  if h : 0 < y ∧ y ≤ x then
    have : x - y < x := Nat.sub_lt (Nat.lt_of_lt_of_le h.1 h.2) h.1
    div (x - y) y + 1
  else
    0
```

Cuando Lean se encuentra una definición recursiva como la función anterior, primero prueba con recursión estructural, y solo cuando esta falla recurre a recursión bien fundamentada. En general, Lean usa la táctica `decreasing_tactic` para encontrar un argumento de la función y una relación bien fundamentada con respecto a la cual el argumento decrece con las aplicaciones recursivas de la función. En el ejemplo anterior, la proposición auxiliar `x - y < x` sirve como pista para esta táctica.

Otra posibilidad es utilizar `termination_by` al final de la definición de la función. Con esta cláusula podemos proporcionar a Lean una forma de "aglutinar" los argumentos de la función en un solo tipo para que Lean trate de aplicar la táctica `decreasing_tactic` sobre ese tipo. Así se define la función de Ackermann a continuación:

```
def ack : Nat → Nat → Nat
  | 0,   y   => y+1
  | x+1, 0   => ack x 1
  | x+1, y+1 => ack x (ack (x+1) y)
termination_by ack x y => (x, y)
```

De esta forma los argumentos `x y` de la función se aglutinan en el tipo `Nat × Nat` y Lean utiliza *resolución de clases de tipos* (ver Sección 2.5) para sintetizar un elemento de tipo `WellFoundedRelation (Nat × Nat)` a partir de la relación bien fundamentada que ya conoce para `Nat`.

En el siguiente ejemplo la terminación de la función se prueba mostrando que `as.size - i` decrece con cada aplicación recursiva:

```
def takeWhile (p : α → Bool) (as : Array α) : Array α :=
  go 0 #[]
where
  go (i : Nat) (r : Array α) : Array α :=
    if h : i < as.size then
      let a := as.get (Fin.mk i h)
      if p a then
        go (i+1) (r.push a)
      else
        r
    else
      r
termination_by go i r => as.size - i
```

2.4.4 Recursión mutua

Lean también admite definiciones por recursión mutua. La sintaxis es la misma que la utilizada en las definiciones mutuas de tipos inductivos:

```
mutual
  def even : Nat → Bool
    | 0   => true
    | n+1 => odd n

  def odd : Nat → Bool
    | 0   => false
```

```
    | n+1 => even n
end
```

La definición anterior se compila internamente como una función recursiva. Esta función, `even._mutual`, toma como argumento un elemento de tipo suma: o bien una entrada para `even`, o bien una entrada para `odd`. Según la entrada, `even._mutual` devuelve la salida apropiada:

```
#print even
/-
def even : Nat → Bool :=
  fun x => even._mutual (PSum.inl x)
-/
#print odd
/-
def odd : Nat → Bool :=
  fun x => even._mutual (PSum.inr x)
-/
```

Para definir `even._mutual` Lean utiliza una medida bien fundamentada adecuada. Las definiciones internas como `even._mutual` no están pensadas para ser leídas por usuarios.

2.5 Clases de tipos

Las clases de tipos se introdujeron para facilitar la definición y el uso de polimorfismos en Lean. La idea es que si queremos definir una función polimórfica, como la suma, podemos definir una función que toma como argumentos el tipo `α` de los elementos que vamos a sumar (argumento implícito), una implementación de la suma entre elementos de `α` y dos elementos de `α`.

Las clases de tipos junto con la resolución de clases de tipos (*type class resolution*) permiten que el segundo argumento, la implementación de la suma, también sea implícito. Por ejemplo:

```
class Add (α : Type) where
  add : α → α → α
```

```
#check @Add.add
-- Add.add : {α : Type} → [self : Add α] → α → α → α
```

Los corchetes del argumento de tipo `Add α` indican que se trata de un argumento implícito que Lean debe sintetizar usando resolución de clases de tipos, es decir, a través de una base de datos que contiene instancias declaradas por el usuario. Las instancias de una clase se definen de la siguiente forma:

```
instance : Add Nat where
  add := Nat.add

instance : Add Int where
  add := Int.add

instance : Add Float where
  add := Float.add
```

Esto nos permite definir la función polimórfica `double` como sigue:

```
def double [Add α] (x : α) : α :=
  Add.add x x

#eval double 10     -- 20

#eval double 2.3    -- 4.600000
```

Una instancia `instance: Add α` también nos permite escribir `a + b` para elementos `a b : α` en lugar de `Add.add a b`.

Las clases están construidas sobre los tipos inductivos. De hecho, el tipo inductivo `Inhabited` visto en la sección 2.3.2 se define en realidad en la librería de Lean como una clase de tipos:

```
class Inhabited (a : Type u) where
  default : a

#check @Inhabited.default
-- Inhabited.default : {a : Type u} → [self : Inhabited a] → a
```

Gracias a esto podemos usar la resolución de clases para inferir elementos "por defecto" de un tipo habitado. Por ejemplo, en la librería se declaran las siguientes instancias:

```
instance : Inhabited Bool where
  default := true

instance : Inhabited Nat where
  default := 0

instance : Inhabited Unit where
  default := ()

instance : Inhabited Prop where
  default := True

#eval (Inhabited.default : Nat)   -- 0

#eval (Inhabited.default : Bool)  -- true
```

Lo que hace especialmente útil a las clases de tipos es que la declaración de una instancia puede depender de otras instancias implícitas. Esto causa que la resolución de clases busque las instancias en cadena recursivamente, haciendo *backtracking* si es necesario, en una búsqueda tipo Prolog.

A continuación se muestran algunos ejemplos de declaración de instancias en cadena:

```
instance [Add a] : Add (Array a) where
  add x y := Array.zipWith x y (· + ·)

instance [Inhabited a] [Inhabited b] : Inhabited (a × b) where
  default := (default, default)

instance [Inhabited b] : Inhabited (a -> b) where
  default := fun _ => default
```

2.5.1 Proposiciones decidibles

En esta sección consideramos otro ejemplo importante de clase de tipos definida en la librería de Lean: la clase `Decidable` de las proposiciones decidibles. Esencialmente, una proposición es decidible si podemos decidir si es verdadera o falsa. Por supuesto, esta distinción solo es útil en el contexto de las matemáticas constructivas, pues en el ámbito clásico toda proposición es decidible.

En términos computacionales, la clase `Decidable` puede usarse para inferir un procedimiento que determina de forma efectiva si una proposición es cierta o no. La definición en la librería de Lean es la siguiente:

```
class inductive Decidable (p : Prop) where
  | isFalse (h : ¬p) : Decidable p
  | isTrue  (h : p)  : Decidable p
```

Observemos que, dada `p : Prop`, `Decidable p` no es una proposición, sino de tipo `Type`. Por lo tanto, tener un elemento `t : Decidable p` es más fuerte que tener un elemento `t : p ∨ ¬p`, pues permite definir valores de tipo arbitrario dependiendo de si `p` es cierta o no (recordemos que, según lo visto en la sección 2.3.7, a partir de una prueba `t : p ∨ ¬p` solo podemos definir otras pruebas, pues `p ∨ ¬p` es de tipo `Prop`).

Para que una expresión de la forma `if p then a else b` sea computable, es necesario que `p` sea decidible. El hecho de que `Decidable` sea una clase y no simplemente un tipo inductivo hace posible la siguiente definición en la librería de Lean:

```
def ite {α : Sort u} (c : Prop) [h : Decidable c] (t e : α) : α :=
  Decidable.casesOn (motive := fun _ => α) h (fun _ => e)
  (fun _ => t)
```

La expresión `if p then a else b` es en realidad azúcar sintáctico para `ite p a b`.

2.6 Axiomas y computación

Hemos visto que la versión del Cálculo de Construcciones implementada en Lean incluye los tipos de funciones dependientes, los tipos inductivos y una jerarquía de universos empezando por el universo `Prop`, que es impredicativo y con pruebas irrelevantes. En esta sección vamos a considerar algunas extensiones del cálculo mediante nuevos axiomas y reglas.

Ha de tenerse en cuenta que, aunque por lo general extender un sistema fundacional puede ser útil para probar nuevos teoremas o para simplificar algunas pruebas,

también puede acarrear consecuencias negativas a nivel computacional que van más allá de la corrección.

La librería estándar de Lean define un axioma adicional, la extensionalidad proposicional, y una construcción del cociente que a su vez implica el principio de extensionalidad de funciones. A pesar de que usar estos axiomas puede bloquear la evaluación en el sistema, como Lean elimina tipos e información proposicional al compilar definiciones a *bytecode* y estos axiomas solo añaden nuevas proposiciones, resulta que son compatibles con la interpretación computacional de los términos de Lean: toda expresión cerrada tiene una forma canónica que puede ser evaluada por el sistema.

Por otra parte, en la librería se define también un axioma de elección que es completamente antitético a cualquier posible interpretación computacional, pues genera datos "mágicamente" a partir de una proposición que afirma la existencia de esos datos. Su uso es esencial en algunas construcciones y teorías clásicas, pero las expresiones que usan este axioma para generar datos no tienen ningún contenido computacional y deben ser marcadas en Lean como `noncomputable` (no computables).

2.6.1 Contexto histórico y filosófico

Durante la mayor parte de su historia, las matemáticas han sido esencialmente computacionales; de la prueba de un teorema del tipo "para todo x existe un y tal que..." podía por lo general extraerse un algoritmo capaz de computar tal y dado el x.

En el siglo XIX, sin embargo, el incremento en la complejidad de los argumentos matemáticos provocó el desarrollo de nuevos estilos de razonamiento que suprimían información algorítmica y aplicaban descripciones abstractas de objetos matemáticos que escondían los detalles sobre la representación exacta de esos objetos.

A día de hoy aún hay acuerdo generalizado en que la computabilidad es importante para las matemáticas, pero hay varios enfoques sobre cómo abordar las cuestiones computacionales. Desde el punto de vista puramente constructivista, toda afirmación matemática debe tener una interpretación computacional directa. Desde un punto de vista clásico, es más productivo hacer una separación de problemas: podemos usar un lenguaje y unos métodos formales para escribir programas, conservando a su vez libertar para usar teorías y métodos no constructivistas a la hora de razonar sobre esos programas. Lean está diseñado para ser compatible con ambos enfoques.

La parte más computacionalmente pura de la teoría de tipos dependientes evita

por completo el uso de `Prop`. El hecho de introducir un universo `Prop` con pruebas irrelevantes y marcar los teoremas como definiciones irreducibles supone un primer paso hacia la separación de problemas descrita en el párrafo anterior. Aun así pueden definirse objetos computacionales que incorporan elementos de tipo `Prop`; la idea es que estos elementos pueden ayudarnos a razonar sobre los efectos de la computación, pero pueden ser ignorados cuando extraemos los programas subyacentes a los objetos.

Al introducir un universo `Prop` con pruebas irrelevantes se puede considerar legítimo el uso de, por ejemplo, el principio del tercero excluso. Y en efecto, este axioma no provoca el bloqueo de la evaluación en *bytecode*, siempre y cuando se utilice para obtener otras proposiciones y no para generar datos. El axioma de elección, sin embargo, elimina por completo de la teoría la distinción entre la irrelevancia de las pruebas y la relevancia de los datos.

2.6.2 Extensionalidad proposicional

El axioma de extensionalidad proposicional es el siguiente:

```
axiom propext {a b : Prop} : (a ↔ b) → a = b
```

Este axioma es consistente con las interpretaciones conjuntistas de la irrelevancia de las pruebas en las que un elemento `a : Prop` es o bien vacío, o bien el conjunto unitario `{*}` para un cierto elemento distinguido `*`.

Como consecuencia de `propext`, podemos sustituir proposiciones equivalentes en cualquier contexto, por ejemplo:

```
example (a b c d e : Prop) (h : a ↔ b) :
    (c ∧ a ∧ d → e) ↔ (c ∧ b ∧ d → e) :=
  propext h ▸ Iff.refl _

example (a b : Prop) (p : Prop → Prop) (h : a ↔ b)
    (h₁ : p a) : p b :=
  propext h ▸ h₁
```

2.6.3 Extensionalidad funcional

El principio de extensionalidad funcional afirma que si dos funciones de tipo `(x : α) → β x` coindicen en todos los valores de entrada, entonces son iguales:

```
universe u v
#check (@funext :
       {α : Type u} →
       {β : α → Type u} →
       {f g : (x : α) → β x} →
       (∀ (x : α), f x = g x) →
       f = g)
```

Desde un punto de vista clásico/conjuntista, esto es exactamente lo que significa que dos funciones sean iguales. Esto se conoce como un enfoque *extensional* de las funciones. Desde una perspectiva constructivista, es más natural pensar en las funciones como algoritmos o programas descritos de forma explícita. Así, dos programas pueden devolver la misma salida para cada entrada a pesar de ser muy distintos sintácticamente. Esto se conoce como un enfoque *intensional* de las funciones.

La extensionalidad funcional se sigue de la existencia de cocientes (que describiremos en la siguiente sección):

```
#print funext
/-
theorem funext.{u, v} : ∀ {α : Sort u} {β : α → Sort v}
  {f₁ f₂ : (x : α) → β x}, (∀ (x : α), f₁ x = f₂ x) → f₁ = f₂ :=
fun {α} {β} {f₁ f₂} h =>
  let_fun this := congrArg extfunApp (Quotient.sound h);
  this
-/
```

2.6.4 Cocientes

Sean `α` un tipo y `r` una relación de equivalencia sobre `α`. El cociente `α / r` es el tipo de los elementos de `α` módulo `r`. Si `f : α → β` es una función que respeta la relación de equivalencia, entonces existe una función `f'`, denominada elevación de `f`, tal que `f' ⟦x⟧ = f x` para toda clase de equivalencia `⟦x⟧`. La

librería estándar de Lean cuenta con constantes que introducen estas construcciones, e implementa la última ecuación como una regla de reducción definicional.

En su forma más básica, la construcción del cociente no necesita que la relación `r` sea de equivalencia:

```
universe u v

axiom Quot : {α : Sort u} → (α → α → Prop) → Sort u

axiom Quot.mk : {α : Sort u} → (r : α → α → Prop) → α → Quot r

axiom Quot.ind :
    ∀ {α : Sort u} {r : α → α → Prop} {β : Quot r → Prop},
      (∀ a, β (Quot.mk r a)) → (q : Quot r) → β q

axiom Quot.lift :
    {α : Sort u} → {r : α → α → Prop} → {β : Sort u} → (f : α → β)
    → (∀ a b, r a b → f a = f b) → Quot r → β
```

Dados una función `f : α → β` y una prueba `h` de que `f` respeta la relación `r`, la idea es que para cada elemento `a : α` la función `Quot.lift f h` envía la clase `Quot.mk r a` (la clase de `a`) en `f a`. De hecho, este principio de computación para la función `Quot.lift f h` se declara como una regla de reducción definicional, como puede verse en el siguiente ejemplo:

```
def mod7Rel (x y : Nat) : Prop :=
  x % 7 = y % 7

-- the quotient type
#check (Quot mod7Rel : Type)

def f (x : Nat) : Bool :=
  x % 7 = 0

theorem f_respects (a b : Nat) (h : mod7Rel a b) : f a = f b := by
  simp [mod7Rel, f]
  exact h ▸ rfl

-- El principio de computación:
example (a : Nat) :
    Quot.lift f f_respects (Quot.mk mod7Rel a) = f a :=
  rfl
```

Las constantes `Quot`, `Quot.mk`, `Quot.ind` y `Quot.lift` no son muy fuertes por sí mismas. Puede comprobarse que todas se satisfacen en la infraestructura lógica si tomamos `Quot r` como `α` y `Quot.lift` como la función identidad (ignorando `h`). Por esta razón, estas constantes no se consideran axiomas adicionales.

Lo que convierte la construcción `Quot` en un cociente de verdad es el siguiente axioma:

```
axiom Quot.sound :
    ∀ {α : Type u} {r : α → α → Prop} {a b : α},
    r a b → Quot.mk r a = Quot.mk r b
```

2.6.5 Axioma de elección

Para declarar el axioma de elección de la librería de Lean necesitamos el tipo `Nonempty`, que se define de la siguiente forma:

```
class inductive Nonempty (α : Sort u) : Prop where
  | intro (val : α) : Nonempty α
```

Como `Nonempty α` es de tipo `Prop` y su único constructor contiene datos, solo pueden generarse pruebas de elementos de `Prop` partiendo de una prueba de `Nonempty α`, a diferencia de lo que sucede con `Inhabited α`. De hecho, `Nonempty α` es equivalente a `∃ x : α, True`.

El axioma de elección de Lean es el siguiente:

```
axiom choice {α : Sort u} : Nonempty α → α
```

Dada la afirmación `h` de que `α` es no vacío, `choice h` genera mágicamente un elemento de `α`. Esto bloquea cualquier computación posible pues, debido a la interpretación de `Prop`, `h` no contiene ninguna información sobre cómo conseguir tal elemento.

Principio del tercero excluso

Puede probarse dentro del propio Lean (ver Capítulo 12 de [4] que el principio del tercero excluso se sigue del axioma de extensionalidad proposicional, el principio de

extensionalidad funcional y el axioma de elección.

Recordemos que en la sección 2.5.1 se explicó que `Decidable p` es más fuerte que `p ∨ ¬p`. Con el axioma de elección ambos son equivalentes:

```
noncomputable def decidable_of_excluded {p : Prop} :
  p ∨ ¬p → Decidable p :=
    fun h =>
      have nonempty : Nonempty (Decidable p) :=
        h.elim
        (fun hp =>
          Nonempty.intro (Decidable.isTrue hp))
        (fun hnp =>
          Nonempty.intro (Decidable.isFalse hnp))
      choice nonempty
```

En definitiva, con el axioma de elección todas las proposiciones son decidibles.

3 Formalización en Lean

En este último capítulo vamos a ver de forma práctica cómo la infraestructura fundacional de Lean puede usarse para formalizar matemáticas. Concretamente vamos a formalizar nociones sobre Topología, con el objetivo final de definir la topología compacto-abierta sobre el espacio de las funciones continuas entre dos espacios topológicos.

Tanto lo explicado en el capítulo 2 como el código de este capítulo corresponden a Lean4, la última versión de Lean. Esto es importante por dos motivos. En primer lugar porque Lean4 no es retrocompatible con Lean3, que es la versión más extendida de Lean (pues Lean4 aún es muy reciente). Y en segundo lugar, y relacionado con lo primero, porque la librería mathlib[1] [5] aún no ha sido adaptada para Lean4. Como consecuencia no es posible importar los ficheros de mathlib en Lean4, por lo que cualquier formalización en esta versión debe partir de la librería estándar de Lean4. En particular, el código que veremos durante el capítulo no depende de ningún otro fichero más allá de la librería estándar de Lean4.

La elección de la Topología en lugar de otra teoría matemática se debe a que considero que es una teoría lo suficientemente elemental como para no requerir mucha formalización previa y al mismo tiempo lo suficientemente rica como para poder hacer uso de las bondades de Lean durante su formalización. Por un lado, la única base fundacional previa necesaria para formalizar conceptos de Topología es la Teoría de Conjuntos, que se puede formalizar fácilmente a partir de la lógica de primer orden de Lean. Por otro lado, la Topología cuenta con nociones estructuralmente "complejas" que permiten mostrar de forma explícita la expresividad que proporciona la teoría de tipos dependientes de Lean. Un ejemplo claro de esto es el *tipo* de las funciones

[1] Mathlib (https://github.com/leanprover-community/mathlib) es una librería matemáticas formalizadas en Lean mantenida por la comunidad. Actualmente es compatible únicamente con Lean3.

continuas entre dos espacios topológicos, que definiremos en la sección 3.2.

Los ficheros de Lean correspondientes al contenido de este capítulo pueden encontrarse en la carpeta `Scripts` del siguiente repositorio de GitHub:

Repositorio del capítulo 3[2]

Es importante notar que aunque todo el código que veremos aquí es original, muchas de las ideas están inspiradas (o tomadas directamente) de las formalizaciones en mathlib para Lean3. El objetivo de este capítulo no es llevar a cabo formalizaciones novedosas ni pruebas complicadas[3], sino mostrar de forma didáctica cómo puede ponerse en marcha la maquinaria de tipos de Lean para formalizar matemáticas. Se recomienda encarecidamente leer el apéndice A para entender el capítulo 3 en su totalidad.

3.1 Teoría de Conjuntos

Para poder definir los conceptos de Topología necesitaremos trabajar con conjuntos, así que antes de formalizar el concepto de espacio topológico debemos formalizar al menos la parte más básica de la Teoría de Conjuntos. Los ficheros del repositorio correspondientes a esta sección son `set_theory.lean` (secciones 3.1.1 y 3.1.2) y `finite.lean` (Sección 3.1.3).

3.1.1 Definiciones básicas

Dado un conjunto X, es muy común en numerosas partes de las matemáticas identificar un conjunto S de elementos de X con el predicado "x pertenece a S" sobre los elementos de X. Recíprocamente, un predicado P sobre X suele identificarse con el conjunto de elementos de X que verifican P. Dado un tipo `α`, esta es precisamente la forma en que definimos en Lean el tipo `Set α` de los conjuntos formados por elementos de `α`:

```
def Set (α : Type u) := α → Prop
```

Así, dados un conjunto `s : Set α` y un término `a : α`, la proposición `s a` significa "`a` pertenece a `s`".

[2] https://github.com/rodripalUS/TFM_cap3

[3] Para realizar formalizaciones de conceptos matemáticos más profundos o probar teoremas más complejos es inevitable el uso de las tácticas de Lean, lo cual se sale del objetivo del trabajo.

La librería de Lean cuenta con la clase `Membership α β` para declarar que hay una noción de pertenencia de tipo `α → β → Prop` entre los elementos de `α` y los elementos de `β`.

```
instance : Membership (α : Type u) (Set α) :=
  Membership.mk (fun a s => s a)
```

Nótese que `fun a s => ...` es equivalente a `fun a => fun s => ...`.

La declaración de una instancia de `Membership α (Set α)` habilita el uso de la notación `a ∈ s` para `s a`.

A continuación se definen los conjuntos vacío y total y las operaciones básicas entre conjuntos.

```
-- Conjunto vacío
def empty : Set α := fun _ => False

-- Para usar la notación ∅ para el conjunto vacío:
instance : EmptyCollection (Set α) := EmptyCollection.mk empty

-- Conjunto total (conjunto formado por todos los elementos de α)
def univ : Set α := fun _ => True

-- Subconjunto
def subset (s1 s2 : Set α) : Prop :=
  ∀ {a}, a ∈ s1 → a ∈ s2

-- Unión finita
def union (s t : Set α) : Set α :=
  fun a => a ∈ s ∨ a ∈ t

-- Intersección finita
def inter (s t : Set α) : Set α :=
  fun a => a ∈ s ∧ a ∈ t

-- Conjunto complementario
def compl (s : Set α) : Set α :=
  fun a => ¬ a ∈ s

-- Para la notación '-a' para 'compl a'
instance : Neg (Set α) := Neg.mk compl
```

Observemos que, en las definiciones anteriores, la sintaxis `fun a : α => p a`

para referirse a un conjunto (suponiendo `p : α → Prop`) puede leerse como el conjunto $\{a \in \alpha \mid p(a)\}$, es decir, el conjunto de los elementos de `α` que verifican `p`. En las definiciones de los conjuntos `empty` y `univ` el elemento `a` se sustituye por `_` porque no es necesario[4] para declarar la propiedad que tienen que cumplir los elementos del conjunto.

Lean permite al usuario introducir su propia notación para hacer más legibles las definiciones y pruebas.

```
-- Notación s ⊆ t para 'subset s t'
infix:50 " ⊆ " => subset

-- Notación s ∪ t para 'union s t'
infixl:65 "∪" => union

-- Notación s ∩ t para 'inter s t'
infixl:65 "∩" => inter
```

El comando `infix` sirve para introducir notación infija, mientras que `infixl` introduce notación infija con asociatividad por la izquierda. Los valores 50 y 65 indican la prioridad de agrupación que tiene la notación introducida, siendo 1024 la prioridad máxima.

En la definición de espacio topológico juega un papel importante el concepto de *familia arbitraria de conjuntos*, pues lo necesitamos para poder hablar de la unión arbitraria de conjuntos. Una familia arbitraria de conjuntos de elementos de `α` puede verse simplemente como un conjunto cuyos elementos son conjuntos de `α`, es decir, cómo un término de tipo `Set (Set α)`. De esta forma podemos definir la unión y la intersección arbitrarias de conjuntos:

```
-- Union de familia arbitraria de conjuntos
def unionF (F : Set (Set α)) : Set α :=
  fun a => ∃ s, s ∈ F ∧ a ∈ s

-- Notación ⋃₀ F para 'unionF F'
prefix:110 "⋃₀" => unionF

-- Intersección de familia arbitraria de conjuntos
def interF (F : Set (Set α)) : Set α :=
 fun a => ∀ s, s ∈ F → a ∈ s
```

[4] Cuando el nombre de una variable introducida no se está utilizando, Lean avisa (aunque algunas veces se equivoca) subrayando la variable de amarillo.

```
-- Notación ⋂₀ F para 'interF F'
prefix:110 "⋂₀" => interF

-- Familia de complementarios
def complF (F : Set (Set α)) : Set (Set α) :=
  fun s => -s ∈ F
```

Otras nociones básicas sobre conjuntos que usaremos son las siguientes:

```
-- Imagen inversa
def preimage (f : α → β) (s : Set β) : Set α := fun a => f a ∈ s

-- Notación f⁻¹(s) para 'preimage f s'
notation f "⁻¹(" s ")" => preimage f s

-- Imagen
def image (f : α → β) (s : Set α) : Set β :=
  fun b => ∃ a : α, a ∈ s ∧ f a = b

-- Notación Im(f,s) para 'image f s'
notation  "Im(" f "," s ")" => image f s

-- Conjuntos disjuntos
def disjoint (s t : Set α) : Prop := (s ∩ t) ⊆ ∅

-- Conjunto unitario
def singleton (a : α) : Set α := fun b => b = a

-- Notación { a } para 'singleton a'
notation "{ " a " }" => singleton a
```

3.1.2 Extensionalidad y algunas propiedades sobre conjuntos

Recordemos que, por defecto, Lean es intensional, es decir, a no ser que usemos `propext` (el axioma de extensionalidad proposicional) o `funext` (el principio de extensionalidad funcional) dos términos son iguales únicamente cuando se reducen a términos iguales definicionalmente. Por esta razón, Lean no considera iguales a los conjuntos `∅ : Set α` y `fun a => 0 = 1`, a pesar de que obviamente tienen los mismos elementos (ninguno). En efecto, si Lean los considerara definicionalmente iguales, la siguiente prueba sería válida:

```
theorem obv : (∅ : Set α) = (fun _ => 0 = 1) := rfl
```

Para probar `obv` necesitamos una propiedad de extensionalidad para los conjuntos, es decir, necesitamos una prueba de que si dos conjuntos `s` y `t` tienen los mismos elementos entonces son iguales. Gracias a la identificación entre conjuntos y predicados, esta propiedad puede probarse fácilmente usando `propext` y `funext`:

```
theorem setext {s t : Set α} (h: ∀ {a}, a ∈ s ↔ a ∈ t) : s = t :=
  funext (fun _ => propext h))
```

Esto nos permite probar propiedades sobre conjuntos directamente a partir de las propiedades análogas en lógica proposicional[5].

```
-- Una de las leyes de Morgan (versión proposicional)
theorem or_not_iff_not_and {p q : Prop} : ¬ (p ∧ q) ↔ ¬p ∨ ¬q :=
  ⟨fun hnpq : ¬(p ∧ q) =>
    Or.elim (Classical.em p)
    (fun hp => Or.inr (fun hq => hnpq ⟨hp,hq⟩))
    (fun hnp => Or.inl hnp),
  fun hnpnq : ¬p ∨ ¬q =>  fun ⟨hp,hq⟩ =>
    hnpnq.elim (fun hnp => hnp hp) (fun hnq => hnq hq)⟩

-- La versión conjuntista
theorem eq_compl_inter_union_compl {s t : Set α} :
    -(s ∩ t) = (-s) ∪ (-t) :=
  setext or_not_iff_not_and

-- Equivalencia entre ¬¬p y p
theorem iff_not_not {p : Prop} : ¬¬p ↔ p :=
  Iff.intro
  Classical.byContradiction
  (fun hp : p => fun hnp : ¬p => (hnp hp))

-- Propiedad involutiva del complementario
theorem compl_compl_eq {α : Type u} {s : Set α} :  -(-s) = s :=
  setext iff_not_not
```

La Teoría de Conjuntos es una teoría basada en la lógica clásica. Observemos que para probar las propiedades anteriores hemos necesitado hacer uso

[5]Todos los resultados sobre lógica proposicional que se han necesitado en la formalización están demostrados en el fichero `logic.lean` del repositorio.

de `Classical.em` y `Classical.byContradiction`. Recordemos que `em p` (omitimos el identificador `Classical`) produce una prueba de `p ∧ ¬p`. Dada una prueba `hnnp` de `¬¬p`, las expresión `byContradiction hnpp` produce una prueba de `p`. Observemos que `byContradiction hnpp` es equivalente al siguiente término:

```
(em p).elim (fun hp => hp) (fun hnp => (hnnp hnp).elim)
```

La correspondencia entre lógica y la Teoría de Conjuntos se manifiesta de forma menos directa en los resultados que involucran a familias de conjuntos. Consideremos por ejemplo la siguiente prueba de que el complementario de la unión de una familia de conjuntos está contenido en la intersección arbitraria de la familia de los complementarios:

```
theorem compl_unionF_mp {F : Set (Set α)} :
    -(⋃₀ F) ⊆ ⋂₀ (complF F) :=
  fun h_compl_union =>
    forall_comp (fun s =>
      implies_of_not_and (forall_of_not_exists h_compl_union s))
```

El enunciado "análogo" en lógica de primer orden sería `¬(∃ a, p a) → ∀ a, ¬ p a,` que es precisamente el teorema `forall_of_not_exists`. Sin embargo, para completar la prueba necesitamos también los resultados siguientes resultados, cuyas pruebas pueden consultarse en el fichero `logic.lean` del repositorio:

```
implies_of_not_and : ¬ (p ∧ q) → (p → ¬ q)

forall_comp : (∀ s, s ∈ F → p (-s)) → ∀ s, s ∈ (complF F) → p s
```

3.1.3 Conjuntos finitos

Para definir la compacidad en un espacio topológico usaremos el concepto de recubrimiento y, más concretamente, el concepto de recubrimiento finito. Un recubrimiento de un conjunto `s` se definirá como una familia `F` cuya unión contiene a `s`. Tal recubrimiento será finito si la familia `F` es finita como conjunto. En definitiva, necesitamos una noción de finitud para conjuntos, es decir, para términos de tipo `Set α`.

Una forma natural de garantizar que un conjunto `s` es finito es dar una lista que contiene todos los elementos de `s`. Esto puede formalizarse en Lean como una estructura[6] con dos campos:

```
structure finite_set (s : Set α) where
  list : List α
  all_in_list : ∀ a, a ∈ s → a ∈ list
```

Es decir, un elemento de `finite_set s` es una lista `list` de elementos de `α` junto con una prueba `all_in_list` de que todos los elementos de `s` están en `list`.

Observemos que, en la definición anterior, se ha utilizado la notación `a ∈ list`. Esto es posible porque en la librería de Lean hay declarada una instancia de `Membership α (List α)`. Más concretamente `a ∈ list` es equivalente a `List.Mem a list`, que se trata de una proposición definida inductivamente con dos constructores:

```
List.Mem.head : ∀ {α : Type u} {a : α} (as : List α),
                List.Mem a (a :: as)

List.Mem.tail : ∀ {α : Type u} {a : α} (b : α) {as : List α},
                List.Mem b as → List.Mem b (a :: as)
```

Aunque solo la usaremos para conjuntos, la definición de `finite_set` puede generalizarse usando clases de tipos. En efecto, para que la definición `finite_set` tenga sentido sobre términos de un cierto tipo `β` lo único que necesitamos es que exista un tipo `α` para el cual haya declarada una instancia de la clase `Membership α β`.

```
structure finite {α : Type u} {β : Type v} [Membership α β] (b : β)
where
  list : List α
  all_in_list : ∀ {a}, a ∈ b → a ∈ list
```

Ahora bien, observemos que, dado `b : β`, `finite b` es de tipo `Type u` para algún `u`, es decir, `finite b` es un tipo de datos y no una proposición. Para poder enunciar teoremas sobre conjuntos finitos necesitamos un predicado `is_finite b`

[6] Las estructuras son tipos inductivos no recursivos que tienen un único constructor. En la sección 2.4.2 del apéndice se explican las estructuras con más detalle.

que sea análogo a `finite b` pero que sea de tipo `Prop`. Un primer intento natural podría ser escribir lo siguiente:

```
structure is_finite {α : Type u} {β : Type v} [Membership α β]
    (b : β) : Prop
where
  list : List α
  all_in_list : ∀ {a}, a ∈ b → a ∈ list
```

Sin embargo, el código anterior provoca un error. La razón es que la definición de la estructura `is_finite` supone la definición automática de las proyecciones `is_finite.list` e `is_finite.all_in_list`. Con la segunda no habría problema, pues esta proyección produciría una prueba de `∀ a, a ∈ b → a ∈ list` a partir una prueba de `is_finite b`. Sin embargo, la proyección `is_finite.list` produciría una lista a partir de una prueba de `is_finite b`. Recordemos que esto no es posible, pues las reglas de eliminación[7] de los tipos que viven en `Prop` solo pueden producir pruebas; no pueden generar datos, como son las listas.

En definitiva, los campos de una estructura que vive en `Prop` deben ser también proposiciones. Para evitar este problema podemos definir `is_finite` a partir de `finite` de la siguiente forma:

```
inductive is_finite {α : Type u} {β : Type v} [Membership α β]
    (b : β) : Prop
where
  | intro : finite b → is_finite b
```

Otra posibilidad equivalente a efectos prácticos habría sido omitir la definición de finite y definir directamente el predicado `is_finite` de la siguiente forma:

```
inductive is_finite {α : Type u} {β : Type v} [Membership α β]
    (b : β) : Prop
where
  | intro : (list : List α) → (∀ {a}, a ∈ b → a ∈ list) →
            is_finite b
```

La ventaja de definir primero la estructura `finite` es que tenemos acceso directo a las proyecciones `finite.list` y `finite.all_in_list`.

[7] Las proyecciones son un caso particular de reglas de eliminación.

Con esta definición de finitud se puede probar fácilmente que el conjunto vacío es finito, que los conjuntos unitarios son finitos y que la unión de dos conjuntos finitos es finita.

```
-- El conjunto vacío es finito
theorem is_finite_empty {α : Type u} : is_finite (∅ : Set α) :=
  let list := []
  have all_in_list : ∀ {a}, a ∈ (∅ : Set α) → a ∈ list :=
    fun h => h.elim
  is_finite.intro ⟨list, all_in_list⟩

-- Los conjuntos unitatios son finitos
theorem is_finite_singleton (a : α) : is_finite { a } :=
  let list := [a]
  have obv : a ∈ list := List.Mem.head []
  have all_in_list : ∀ {b}, b ∈ ({ a } : Set α) → b ∈ list :=
    fun h => h ▸ obv
  is_finite.intro ⟨list, all_in_list⟩

-- La unión de dos conjuntos finitos es finita
theorem is_finite_union {s t : Set α} :
        is_finite s → is_finite t → is_finite (s ∪ t) :=
  fun (is_finite.intro ⟨list_s, all_in_list_s⟩)
      (is_finite.intro ⟨list_t, all_in_list_t⟩) =>
    let list := list_s ++ list_t
    have all_in_list : ∀ {a}, a ∈ (s ∪ t) → a ∈ list :=
      fun hunion =>
        hunion.elim
       (fun hsa =>
         List.mem_append_of_mem_left list_t (all_in_list_s hsa))
       (fun hta =>
         List.mem_append_of_mem_right list_s (all_in_list_t hta))
    is_finite.intro ⟨list, all_in_list⟩
```

Los términos `List.mem_append_of_mem_left` y `List.mem_append_of_mem_right` forman parte de la librería de Lean:

```
#check @List.mem_append_of_mem_left
-- ∀ {α : Type u_1} {a : α} {as : List α} (bs : List α),
--   a ∈ as → a ∈ as ++ bs)

#check @List.mem_append_of_mem_right
-- ∀ {α : Type u_1} {b : α} {bs : List α} (as : List α),
--   b ∈ bs → b ∈ as ++ bs
```

De hecho puede probarse que `is_finite` es precisamente el menor predicado que satisface estas tres propiedades. Para probar esto consideramos el siguiente predicado sobre conjuntos:

```
inductive is_finite_alt : Set α → Prop where
  | empty : is_finite_alt ∅
  | singleton : ∀ a, is_finite_alt { a }
  | union : ∀ {s t}, is_finite_alt s → is_finite_alt t →
            is_finite_alt (s ∪ t)
```

Observemos que, por definición, `is_finite_alt` es el menor predicado `p` para el que se cumplen las tres propiedades:

- `p ∅`.
- `p { a }` para todo `a : α`.
- Para todos `s t : Set α`, si `p s` y `p t` entonces `p (s ∪ t)`.

El predicado `is_finite_alt` puede verse como una definición alternativa de la noción de finitud de conjuntos; los conjuntos que satisfacen `is_finite_alt` son aquellos que pueden expresarse como una unión finita (posiblemente vacía) de conjuntos unitarios.

Ya hemos visto que se cumple `is_finite → is_finite_alt`. Para probar el recíproco necesitamos tres resultados previos:

(1) Si todos los elementos de un conjunto `s` están en la lista vacía, entonces `s` es vacío:

```
theorem lemma1 {s : Set α} (h : ∀ {a}, a ∈ s → a ∈ []) :
    s = ∅ := ...
```

(2) El conjunto formado por los elementos que pertenecen a una lista `a::as` es la unión del conjunto unitario `{ a }` y el conjunto formado por los elementos que pertenecen a la lista `as`:

```
theorem lemma2 {a : α} {as : List α} :
    (fun b => b ∈ (a::as)) = { a } ∪ (fun b => b ∈ as) := ...
```

(3) El conjunto formado por los elementos que pertenecen a una lista `as` es finito de acuerdo con la definición `is_finite_alt`:

```
theorem lemma3 : ∀ as : List α,
    is_finite_alt (fun a => a ∈ as) := ...
```

La prueba de `lemma1` hace uso de la táctica `contradiction` [8] para probar que si todos los elementos de un conjunto `s` pertenecen a la lista vacía, entonces `s` está contenido en el conjunto vacío:

```
theorem lemma1 {s : Set α} (h : ∀ {a}, a ∈ s → a ∈ []) : s = ∅ :=
  eq_empty_of_subset_empty
  (fun hsa =>
    have contr := h hsa
    by contradiction)
```

En este caso la táctica `contradiction` intenta unificar la prueba `contr` de `a ∈ []` (para un cierto `a`) con los constructores `List.Mem.head` y `List.Mem.tail` del predicado de pertenencia para listas. En ambos casos la conclusión es que la lista vacía `[]` debe ser de la forma `x::xs` para cierto elemento `x : α` y cierta lista `xs`. Esto entra en contradicción con el hecho de que dos constructores distintos de un mismo tipo inductivo (en este caso el tipo `List`) generan elementos distintos.

Una vez demostrada la proposición `s ⊆ ∅`, la prueba de `lemma1` concluye con el teorema `eq_empty_of_subset_empty`, que afirma que si un conjunto `s` es subconjunto del vacío, entonces `s` es vacío.

La prueba de `lemma2` utiliza *pattern matching* para desglosar la definición de `b ∈ (a::as)`; o bien se tiene `a = b`, o bien se tiene `b ∈ as`.

```
theorem lemma2 {a : α} {as : List α} :
    (fun b => b ∈ (a::as)) = { a } ∪ (fun b => b ∈ as) :=
  setext
    ⟨fun hbinlist =>
    match hbinlist with
    | List.Mem.head as =>         -- a = b
        Or.inl rfl
    | List.Mem.tail a h_b_in_as => -- b ∈ as
        Or.inr h_b_in_as,
    fun hunionb =>
      hunionb.elim
      (fun hsingle =>
```

[8] Consultar la sección A.3 del apéndice.

```
        hsingle ▸ List.Mem.head as)
      (fun hbintail =>
        List.Mem.tail a hbintail)⟩
```

Para demostrar el tercer lema se ha usado recursión estructural. La estructura de la prueba es la siguiente:

- Si la lista `as` es vacía, entonces utilizamos el primer lema para probar que el conjunto `fun a => a ∈ as` es el conjunto vacío y por tanto satisface `is_finite_alt` por definición.
- Si la lista es de la forma `a::as`, entonces usamos el segundo lema para probar que el conjunto `fun b => b ∈ (a::as)` es la unión de `{ a }` y el conjunto `fun b => b ∈ as`. Suponiendo que ya hemos probado `lemma3 as` (es decir, que el conjunto `fun b => b ∈ as` es finito), se sigue que `fun b => b ∈ (a::as)` es finito por ser unión de dos conjuntos finitos.

```
theorem lemma3 : ∀ as : List α, is_finite_alt (fun a => a ∈ as)
  | [] =>
    have hempty : (fun a => a ∈ []) = (∅ : Set α) := lemma1 id
    hempty ▸ is_finite_alt.empty
  | a::as =>
    have div : (fun b => b ∈ (a::as)) = { a } ∪ (fun b => b ∈ as)
        := lemma2
    div ▸ is_finite_alt.union
          (is_finite_alt.singleton a)
          (lemma3 as)
```

La prueba de `is_finite s → is_finite_alt s` se sigue de forma directa a partir de `lemma3` y del teorema `subset_of_finite`, que afirma que todo subconjunto de un conjunto finito (en el sentido de `is_finite_alt`) es también finito.

```
theorem is_finite_alt_of_is_finite {s : Set α} :
    is_finite s → is_finite_alt s :=
  fun (is_finite.intro ⟨list, all_in_list⟩) =>
  -- 'list' es una lista con los elementos de 's'
  -- 'all_in_list' es la prueba de que todos los
  -- elementos de 's' están en 'list'
    let finiteSet : Set α := (fun b => b ∈ list)
    have hsub : s ⊆ finiteSet := all_in_list
    have hfin : is_finite_alt finiteSet := lemma3 list
    subset_of_finite hsub hfin
```

3.2 Topología

En esta sección se formalizan algunas nociones y resultados de Topología, a saber: los espacios topológicos, el interior de un conjunto, la convergencia de sucesiones, la convergencia en conjuntos cerrados, las funciones continuas, la topología generada por una subbase, la compacidad, los espacios de Hausdorff y la topología compacto-abierta. Los ficheros del repositorio correspondientes a esta sección son `topology.lean` (Sección 3.2.1) `topology_defs.lean` (Sección 3.2.2).

3.2.1 Espacios topológicos

Para dotar de topología a un conjunto basta con describir los abiertos y probar que se satisfacen los siguientes axiomas:

- El conjunto vacío es abierto.
- El conjunto total es abierto.
- La intersección finita de abiertos es abierta.
- La unión arbitraria de abiertos es abierta.

En Lean esto puede definirse de forma natural como un tipo inductivo con un único constructor, es decir, como una estructura:

```
structure topological_space (α : Type u) where
  is_open : Set α → Prop
  is_open_univ : is_open univ
  is_open_inter : ∀ (s t : Set α),
                  is_open s → is_open t → is_open (s ∩ t)
  is_open_union : ∀ (F : Set (Set α)),
                  (∀ s, s ∈ F → is_open s) → is_open (⋃₀ F)
```

Es decir, un espacio topológico sobre un tipo `α` viene dado por un predicado `is_open` sobre conjuntos de elementos de `α` junto con las pruebas `is_open_univ`, `is_open_inter` e `is_open_union` de que dicho predicado satisface los axiomas básicos de la topología. Observemos que no hemos incluido en la estructura `topological_space` un campo `is_open_empty` de tipo `is_open ∅` para asegurarnos de que el conjunto vacío siempre es abierto. Ese axioma no es necesario porque puede obtenerse a partir de `is_open_union` tomando la familia `F` como el conjunto vacío `∅ : Set (Set α)`.

Aunque un espacio topológico `T` sobre un tipo `α` venga introducido mediante cuatro campos (`is_open`, `is_open_univ`, `is_open_inter` e `is_open_union`), en realidad está determinado por el predicado `is_open`. Dicho de otra forma, dos espacios tienen la misma topología si tienen los mismos abiertos:

```
theorem topological_space_eq : ∀ T T' : topological_space α,
    T.is_open = T'.is_open → T = T' :=
  fun ⟨p, _, _, _⟩ ⟨p', _, _, _⟩ h => by simp [h]
```

La idea de la prueba anterior es que, como estamos suponiendo `h : T.is_open = T'.is_open`, los tipos de los campos `is_open_univ`, `is_open_inter` e `is_open_union` de `T` y `T'` coinciden. Como las pruebas de proposiciones en Lean no albergan ningún tipo de información, se tiene por tanto que los respectivos campos de `T` y `T'` son definicionalmente iguales. En definitiva, como todos los campos de `T` son iguales a los de `T'`, concluimos que `T` y `T'` son iguales. Escribir esta prueba de forma explícita es tedioso y requiere un conocimiento muy profundo del funcionamiento interno de Lean, por eso hemos utilizado la táctica automática `simp [h]`.

Por comodidad declaramos la estructura `topological_space` como una clase y definimos el predicado `is_open` de la siguiente forma:

```
attribute [class] topological_space
def is_open [T : topological_space α] : Set α → Prop := T.is_open
```

Esto nos permite usar `is_open` sobre conjuntos de `α` en lugar de `T.is_open` siempre que Lean pueda inferir una instancia `T` de `topological_space α`.

También puede definirse un espacio topológico sobre un tipo `α` describiendo los cerrados de la topología, junto con las pruebas de que se satisfacen los axiomas de los cerrados. Eso es justo lo que hace el término `topological_space_of_is_closed`:

```
def topological_space_of_is_closed
    (is_closed : Set α → Prop)
    (is_closed_empty : is_closed empty)
    (is_closed_union : ∀ (s t : Set α),
      is_closed s → is_closed t → is_closed (s ∪ t))
    (is_closed_inter : ∀ (F : Set (Set α)),
      (∀ s, s ∈ F → is_closed s) → is_closed (⋂₀ F)) :
    topological_space α :=
```

```
{is_open := fun s => is_closed (-s),
is_open_univ := show is_closed (-univ)
                from eq_compl_univ ▸ is_closed_empty,
is_open_inter :=
  fun s t hs ht =>
    show is_closed (-(s ∩ t))
    from compl_inter_eq_union_compl ▸ (is_closed_union
                                      (-s) (-t) hs ht),
is_open_union :=
  fun F h_open =>
    have h_closed_cF : ∀ s, s ∈ complF F → is_closed s :=
      fun s (h_cF_s : complF F s ) =>
        have h_ccs : is_closed (-(-s)) := h_open (-s) h_cF_s
        @compl_compl_eq α s ▸ h_ccs
    show is_closed (-(⋃₀ F))
    from compl_unionF ▸ (is_closed_inter
                         (complF F) h_closed_cF)}
```

En la definición anterior se han utilizado las siguientes propiedades relacionadas con la operación de tomar complementarios:

```
#check @eq_compl_univ
-- ∀ {α : Type u_1}, -univ = ∅

#check @compl_inter_eq_union_compl
-- ∀ {α : Type u_1} {s t : Set α}, -(s ∩ t) = -s ∪ -t

#check @compl_compl_eq
-- ∀ {α : Type u_1} {s : Set α}, - -s = s

#check @compl_unionF
-- ∀ {α : Type u_1} {F : Set (Set α)}, -⋃₀F = ⋂₀complF F
```

3.2.2 Algunos conceptos de Topología

Interior de un conjunto

El interior de un conjunto `s` puede definirse como la unión de todos los abiertos contenidos en `s`:

```
def interior [topological_space α] (s : Set α) : Set α :=
  ⋃₀ (fun g => g ⊆ s ∧ is_open g)
```

Sucesión convergente

Una sucesión `x : Nat → α` en un espacio topológico `α` converge a un cierto `x₀` para cualquier abierto `g` que contenga a `x₀` existe un `n : Nat` a partir del cual todos los términos de la sucesión están en `g`:

```
def sequence_limit [topological_space α] (x : Nat → α) (x₀ : α)
    : Prop :=
  ∀ {g : Set α}, is_open g → x₀ ∈ g → (∃ (n : Nat),
                                       ∀ m : Nat, n ≤ m → (x m) ∈ g)
```

Usando `Classical.byContradiction` (es decir, usando el principio del tercero excluso) podemos probar que los conjuntos cerrados son en efecto cerrados con respecto a la convergencia de sucesiones:

```
theorem convergent_limit_in_closed [topological_space α]
    (c : Set α) (h_cc : is_closed c)
    (x : Nat → α) (h_cx : ∀ n : Nat, (x n) ∈ c)
    (x₀ : α) (h_xconv : sequence_limit x x₀) :
    x₀ ∈ c :=
  Classical.byContradiction
  (fun hncx0 =>
    -- Como '-c' es abierto podemos usar 'hncx0' (la hipótesis
    -- de que 'x₀' pertenece a '-c') para encontrar un 'n' a
    -- partir del cual la sucesión se queda contenida en '-c'
    have h_exn : (∃ (n : Nat),
                  ∀ m : Nat, n ≤ m → x m ∈ (-c)) :=
      h_xconv h_cc hncx0
    h_exn.elim
    (fun n hn =>
      -- Por un lado tenemos que 'x n' está en 'c' y por otro
      -- tenemos que 'x n' está en -c
      have h_ccxn : x n ∈ (-c) := hn n (Nat.le_refl n)
      have h_cn : x n ∈ c := h_cx n
      (h_ccxn h_cn).elim))
```

En la prueba anterior `Nat.le_refl` es un teorema de la librería de Lean que afirma que todo natural es menor o igual que sí mismo.

Funciones continuas

Una función `f` entre dos espacios topológicos es continua si la imagen inversa por `f` de cualquier abierto es abierto:

```
def continuous [topological_space α] [topological_space β]
    (f : α → β) : Prop :=
  ∀ s : Set β, is_open s → is_open (f⁻¹(s))
```

Dados dos espacios topológicos `α` y `β`, podemos definir el tipo `C(α,β)` de las funciones continuas de `α` en `β`:

```
structure continuous_functions (α : Type u) (β : Type v)
    [topological_space α] [topological_space β]
where
  function : α → β
  is_continuous : continuous function

notation "C(" α "," β ")" => continuous_functions α β
```

Un elemento `f` de `C(α,β)` se trata de una función `f.function : α → β` junto con una prueba `f.is_continuous` de que `f` es continua.

Nótese la facilidad con la que hemos definido en Lean el tipo `C(α,β)`, que no solo depende de los tipos `α` y `β`, sino también de las instancias de `topological_space α` y `topological_space β`. Un tipo como `C(α,β)` no puede definirse en sistemas basados en la teoría de tipos simples, donde los tipos no pueden depender de otros tipos. El hecho de que `C(α,β)` sea un tipo nos permitirá dotarlo de topología.

Igual que con los espacios topológicos, podemos probar que los elementos de `C(α,β)` vienen determinados por el campo `function`:

```
theorem continuous_functions_eq
    [topological_space α] [topological_space β]
    {f g : C(α,β)} :
    f.function = g.function → f = g :=
  fun heq => show (⟨f.function, f.is_continuous⟩ : C(α,β)) =
                  ⟨g.function, g.is_continuous⟩
             from by simp [heq]
```

Topología generada por una subbase

Dada una familia `B : Set (Set α)` de conjuntos, se define la topología de `α` generada por `B` como la menor topología sobre `α` en la que los conjuntos de `B` son abiertos. La familia `B` recibe el nombre de subbase de la topología generada.

Como la definición de topología que estamos usando viene determinada por el predicado `is_open` que determina sus abiertos, para definir la topología generada por una subbase `B` basta con definir el correspondiente predicado `is_open`. Esto puede hacerse de forma inductiva:

```
inductive generated_topology_open (B : Set (Set (α))) :
    Set α → Prop
where
  | basic : (b : Set α) → B b → generated_topology_open B b
  | univ : generated_topology_open B total
  | inter : (b1 b2 : Set α) → generated_topology_open B b1 →
       generated_topology_open B b2 →
       generated_topology_open B (b1 ∩ b2)
  | unionF : (Fb : Set (Set α)) →
       (∀ b, Fb b → generated_topology_open B b) →
       generated_topology_open B (⋃₀ Fb)
```

Es decir, los conjuntos que satisfacen el predicado `generate_topology_open B` son:

- Los conjuntos de `B`, llamados conjuntos básicos.
- El conjunto total.
- La intersección finita de conjuntos que satisfacen el predicado.
- La unión arbitraria de conjuntos que satisfacen el predicado.

El predicado `generate_topology_open` satisface, por definición, los axiomas necesarios para constituir una topología. Podemos entones definir la topología generada por una subbase de la siguiente forma:

```
def generated_topology (B : Set (Set α)) : topological_space α :=
  {
    is_open := generated_topology_open B,
    is_open_univ := generated_topology_open.univ,
    is_open_inter := generated_topology_open.inter,
    is_open_union := generated_topology_open.unionF
  }
```

Compacidad

Vamos a definir la noción de compacidad usando recubrimientos. Un recubrimiento de un conjunto `s` es una familia `R` de conjuntos cuya unión contiene a `s`:

```
def cover (R : Set (Set α)) (s : Set α) : Prop := s ⊆ ⋃₀ R
```

Un recubrimiento `R` se dice abierto si todos sus conjuntos son abiertos. Dado un recubrimiento `R` de un cierto conjunto `X`, un subconjunto `R'` de `R` que también es recubrimiento de `X` se denomina subrecubrimiento.

Un conjunto `k` es compacto si dado cualquier recubrimiento abierto de `k`, puede extraerse un subrecubrimiento finito del mismo.

```
def compact [topological_space α] (k : Set α) : Prop :=
  ∀ R, cover R k → (∀ s, R s → is_open s) →
    (∃ R', R' ⊆ R ∧ is_finite R' ∧ cover R' k)
```

La noción de compacidad es una generalización topológica de la noción de finitud de conjuntos. Sin embargo, probar que todo espacio finito es compacto usando la definición por listas `is_finite` puede ser muy complejo. Esta es la razón por la que se definió en la sección 3.1.3 la caracterización `is_finite_alt` en términos de uniones finitas de conjuntos unitarios. Gracias a esta caracterización basta con probar lo siguiente:

- El conjunto vacío es compacto.
- Los conjuntos unitarios son compactos.

- La unión de conjuntos unitarios es compacta.

Para probar que el conjunto es vacío es compacto basta con tomar el recubrimiento vacío como subrecubrimiento finito:

```
theorem compact_empty [topological_space α] : compact (∅ : Set α) :=
  fun R _ _ =>
    let R' : Set (Set α) := ∅
    Exists.intro R'
                 ⟨empty_subset R,
                 is_finite_empty,
                 empty_subset (⋃₀ R')⟩
```

Si tenemos un recubrimiento abierto `R` de un conjunto unitario `{ a }` debe haber un conjunto abierto `s ∈ R` que contenga a `a`. Basta entonces con tomar `R' = { s }` como subrecubrimiento finito:

```
theorem compact_singleton [topological_space α] (a : α) :
    compact { a } :=
  fun R hRcover _ =>
    have huniona : a ∈ ⋃₀ R := hRcover rfl
    huniona.elim
    (fun s ⟨hRs,hsa⟩ =>
      let R' : Set (Set α) := { s }
      have hsub : R' ⊆ R :=
        (@in_set_iff_singleton_subset _ R s).mp hRs
      have hfinite : is_finite R' := is_finite_singleton s
      have hcover : cover R' { a } :=
        unionF_singleton s ▸ (in_set_iff_singleton_subset.mp hsa)
      Exists.intro R' ⟨hsub, hfinite, hcover⟩)
```

En la prueba anterior se han usado los siguientes teoremas, cuya demostración puede consultarse en el fichero `set_theory.lean` del repositorio:

```
-- Un elemento 'a' pertenece a un conjunto 's' si y solo
-- si el conjunto unitario { a } está contenido en 's'
in_set_iff_singleton_subset : ∀ {α : Type u} {s : Set α} {a : α},
                              a ∈ s ↔ { a } ⊆ s

-- La unión de la familia unitaria { s } es igual a 's'
unionF_singleton : ∀ {α : Type u} (s : Set α), ⋃₀{ s } = s
```

Si tenemos un recubrimiento abierto `R` de la unión de dos compactos `k` y `k'`,

entonces `R` es recubrimiento abierto de `k` y de `k'` por separado, así que pueden extraerse respectivos subrecubrimientos finitos `Rk` y `Rk'`. Basta con tomar `R' = Rk ∪ Rk'` como subrecubrimiento finito:

```
theorem compact_union [topological_space α] {k k' : Set α} :
    compact k → compact k' → compact (k ∪ k') :=
  fun hk hk' => fun R hRcover hRabto =>
    have hRcoverk : cover R k :=
      (iff_union_of_subsets.mp hRcover).left
    have hRcoverk' : cover R k' :=
      (iff_union_of_subsets.mp hRcover).right
    (hk R hRcoverk hRabto).elim
    (fun Rk ⟨hsubRk, hfinitek, hRkcover⟩ =>
      (hk' R hRcoverk' hRabto).elim
      (fun Rk' ⟨hsubRk', hfinitek', hRk'cover⟩ =>
        let R' := Rk ∪ Rk'
        have hsub : R' ⊆ R :=
          iff_union_of_subsets.mpr ⟨hsubRk,hsubRk'⟩
        have hfinite : is_finite R' :=
          is_finite_union hfinitek hfinitek'
        have hcoverk : k ⊆ ⋃₀ R' :=
          eq_unionF_of_union ▸
          (subset_of_union (⋃₀ Rk') hRkcover).left
        have hcoverk' : k' ⊆ ⋃₀ R' :=
          eq_unionF_of_union ▸
          (subset_of_union (⋃₀ Rk) hRk'cover).right
        have hcover : cover R' (k ∪ k') :=
          iff_union_of_subsets.mpr ⟨hcoverk, hcoverk'⟩
        Exists.intro R' ⟨hsub, hfinite, hcover⟩))
```

En la prueba anterior se han usado los siguientes teoremas:

```
iff_union_of_subsets : ∀ {α : Type u} {s t u : Set α},
                       s ∪ t ⊆ u ↔ s ⊆ u ∧ t ⊆ u

eq_unionF_of_union : ∀ {α : Type u} {A B : Set (Set α)},
                     ⋃₀ (A ∪ B) = ⋃₀ A ∪ ⋃₀ B

subset_of_union : ∀ {α : Type u_1} {s t : Set α} (u : Set α),
                  s ⊆ t → s ⊆ t ∪ u ∧ s ⊆ u ∪ t
```

Una vez demostrados `compact_empty`, `compact_singleton` y `compact_union`, la compacidad de los conjuntos finitos (en el sentido de `is_finite_alt`, y por tanto también en el sentido de `is_finite`) se tiene

trivialmente por inducción:

```
theorem compact_finite_alt [topological_space α] {s : Set α} :
    is_finite_alt s → compact s :=
  fun h =>
    match h with
    | is_finite_alt.empty => compact_empty
    | is_finite_alt.singleton _ => compact_singleton _
    | is_finite_alt.union ha hb => compact_union
                                   (compact_finite_alt ha)
                                   (compact_finite_alt hb)
```

Propiedad de Hausdorff

Un espacio topológico tiene la propiedad de Hausdorff si dos puntos distintos cualesquiera pueden separarse por abiertos disjuntos:

```
def hausdorff_space (α : Type u) [topological_space α] : Prop :=
  ∀ x y : α, x ≠ y → ∃ (u v : Set α), is_open u ∧
                                      is_open v ∧
                                      x ∈ u ∧
                                      y ∈ v ∧
                                      disjoint u v
```

La topología compacto-abierta

La topología compacto-abierta es una topología definida sobre el conjunto de las funciones continuas entre dos espacios topológicos. Se define como la topología generada por los conjuntos de la forma

$$V(K,G) = \{f \in \mathscr{C}(X,Y) \mid f(K) \subseteq G\},$$

donde K es un compacto de X y G es un abierto de Y.

Es decir, dados los espacios topológicos `α` y `β`, la topología compacto-abierta sobre `C(α,β)` es la topología generada por la siguiente subbase:

```
def compact_open_subbasis [topological_space α]
    [topological_space β] : Set (Set C(α,β)) :=
  fun (s : Set C(α,β)) =>
    ∃ (k : Set α) (_ : compact k) (u : Set β) (_ : is_open u),
    (∀ f : C(α,β), f ∈ s →  Im(f.function,k) ⊆ u)

instance compact_open_topology [topological_space α]
    [topological_space β] :  topological_space C(α,β) :=
  generated_topology compact_open_subbasis
```

Gracias a la instancia `compact_open_topology` Lean generará automáticamente una instancia de `topological_space C(α,β)` usando la topología compacto-abierta, siempre y cuando encuentre previamente instancias de topologías sobre `α` y `β`.

Para finalizar el capítulo vamos a probar que, dados los espacios topológicos `α` y `β`, si `β` es un espacio de Hausdorff entonces `C(α,β)` con la topología compacto-abierta también lo es.

Vamos a ver la prueba paso a paso, comenzando por el enunciado:

```
theorem compact_open_hausdorff [T : topological_space α]
    [topological_space β] (h : hausdorff_space β) :
    @hausdorff_space C(α,β) compact_open_topology := ...
```

Para hacer más legible la prueba, llamamos `B` a la subbase de la topología compacto-abierta:

```
let B : Set (Set C(α,β)) := compact_open_subbasis
```

Posteriormente consideramos dos funciones continuas distintas `f` y `g`:

```
fun f g hneq => ...
-- 'hneq' es la prueba de que 'f' y 'g'  son distintas
```

El objetivo es encontrar dos abiertos `uf` y `ug` disjuntos que contengan a `f` y `g` respectivamente.

Recordemos que los elementos de `C(α,β)` tienen dos campos, `function` y `is_continuous`. Como `f` y `g` son elementos distintos de `C(α,β)`, el contrarrecíproco de `continuous_functions_eq` nos proporciona una prueba `hneqfun`

de que las funciones `f.function` y `g.function` son distintas a partir del hecho `hneq` de que los elementos `f` y `g` son distintos.

```
have hneqfun : f.function ≠ g.function :=
  contrapositive continuous_functions_eq hneq
```

Usando el principio de extensionalidad funcional (`funext`) y el principio del tercero excluso (más concretamente, `Classical.byContradiction`), puede probarse `funext_converse` , el recíproco de la extensionalidad funcional, que afirma que si dos funciones son distintas, entonces debe existir un valor en el que difieren.

Gracias a `funext_converse` y a la prueba `hneqfun` de que las funciones `f.function` y `g.function` son distintas, encontramos un elemento `x : α` tal que `f.function x ≠ g.function x` . Por brevedad llamamos `fx` y `gx` a las respectivas imágenes de `x` por `f` y `g` .

```
(funext_converse hneqfun).elim
(fun x hnfxgx =>
  let fx := f.function x
  let gx := g.function x
  -- 'hnfxgx' es la prueba de que 'fx' y 'gx' son distintos
  ... )
```

Como `fx` y `gx` son elementos distintos de `β` , podemos aplicar la hipótesis `h` de que `β` es Hausdorff para encontrar dos abiertos `u` y `v` que separan a `fx` y `gx`

```
(h fx gx hnfxgx).elim
(fun u hexv =>
  hexv.elim
  (fun v ⟨hopenu,hopenv,hufx,hvgx,hdisj⟩ => ...))
  -- hopenu : is_open u
  -- hopenv : is_open v
  -- hufx : fx ∈ u
  -- hvgx : gx ∈ v
  -- hdisj : disjoint u v
```

Consideramos ahora el compacto `k := { x }` de `α` (es compacto porque es unitario):

```
let k := { x }
have hcompact : compact k := compact_singleton x
```

Consideramos ahora el conjunto `uf` de funciones continuas `t` tales que la imagen del compacto `k` por `f` está contenida en el abierto `u`. Está claro que `f` pertenece a `uf`, pues la imagen de `k` por `f` es el conjunto unitario `{ fx }`, que está contenido en `u` por `hufx : fx ∈ u`. Para `g` hacemos una definición análoga.

```
let uf : Set C(α,β) := fun t => Im(t.function, k) ⊆ u
have huff : f ∈ uf :=
  image_singleton f.function x ▸
  in_set_iff_singleton_subset.mp hufx

-- image_singleton : ∀ {α : Type u_1} {β : Type u_2}
--                   (f : α → β) (a : α),
--                   Im(f,{ a }) = { f a }

let ug : Set C(α,β) := fun t => Im(t.function, k) ⊆ v
have hugg : g ∈ ug :=
  image_singleton g.function x ▸
  in_set_iff_singleton_subset.mp hvgx
```

Falta demostrar que `uf` y `ug` son, en efecto, abiertos disjuntos de la topología compacto-abierta.

Para demostrar que son abiertos basta en este caso con probar que son conjuntos básicos de la topología, es decir, que pertenecen a la familia `B`. Esto se tiene por definición, pues, siguiendo la notación del principio de la sección, hemos definido `uf := V(k,u)` y `uf := V(k,v)`.

```
have hBuf : uf ∈ B := ⟨k,hcompact,u,hopenu, fun _ huft => huft⟩
have hBug : ug ∈ B := ⟨k,hcompact,v,hopenv, fun _ hugt => hugt⟩

have hopenuf : is_open uf := generated_topology_open.basic uf hBuf
have hopenug : is_open ug := generated_topology_open.basic ug hBug
```

Nótese que, en la demostración `hBuf` de que `uf` pertenece a `B`, el constructor anónimo[9] `⟨k,hcompact,u,hopenu, fun _ huft => huft⟩` constituye una prueba de que existen un conjunto `k : Set α` compacto (`hcompact`) y un conjunto `u : Set β` abierto (`hopenu`) tales que para toda función `t` en `uf` (se ha sustituido `t` por `_` porque no es necesario declarar su nombre), se cumple que la imagen de `k` por `t` está contenida en `u` (`huft : t ∈ uf` es prueba de ello porque esa es precisamente la definición de "pertenecer a `uf`").

[9] Consultar la sección A.2 del apéndice.

Resta únicamente probar que `uf` y `ug` son disjuntos. Para ello supongamos que tenemos una función `t` que pertenece a la intersección.

```
have hdisj2 : disjuntos uf ug :=
  fun {t} hinter => ...
```

Por un lado, usando el teorema `nonempty_singleton`, que afirma que los conjuntos unitarios no son vacíos, y el teorema `image_of_nonempty`, que afirma que la imagen de un conjunto no vacío es no vacía, llegamos a la conclusión de que la imagen de `k` (recordemos que `k := { x }`), por `t` es no vacía:

```
have hnotempty : ¬ Im(t.function, k) = ∅ :=
  image_of_nonempty t.funcion (nonempty_singleton x)
```

Por otro lado, recordemos que `hinter.left : t ∈ uf` es por definición una prueba de que la imagen de `k` por `t.function` está contenida en `u` y que `hinter.right : t ∈ ug` es por definición una prueba de que la imagen de `k` por `t.function` está contenida en `v`. Así, podemos utilizar el hecho `hdisj` de que `u` y `v` son disjuntos, junto con el teorema `empty_of_subset_of_disjoints`, que afirma que si un conjunto `s` es subconjunto de conjuntos disjuntos, entonces `s` es vacío, para llegar a la conclusión de que la imagen de `k` por `t` es vacía:

```
have hempty : Im(t.function, k) = ∅ :=
  empty_of_subset_of_disjoints hdisj hinter.left hinter.right
```

Aplicando `hnotempty` a `hempty` obtenemos una prueba de `False` y finalizamos la prueba de que `uf` y `ug` son disjuntos:

```
have hdisj2 : disjuntos uf ug :=
  fun {t} hinter =>
    have hnotempty : ¬ Im(t.function, k) = ∅ := ...
    have hempty : Im(t.function, k) = ∅ := ...
    hnotempty hempty
```

Para completar la prueba solo falta introducir las condiciones del cuantificador existencial del predicado `hausdorff_space` en el orden adecuado:

```
⟨uf, ug, hopenuf, hopenug, huff, hugg, hdisj2⟩ :
    ∃ (uf ug : Set C(α,β)), is_open uf ∧ is_open ug ∧
                            f ∈ uf ∧ g ∈ ug ∧ disjoint uf ug
```

4 Conclusiones

A lo largo del trabajo se han estudiado las bases fundacionales del demostrador Lean. El capítulo 1 puede verse como una introducción a la teoría de tipos, en particular, a la teoría de tipos dependientes. Empezando por la teoría intuicionista de Martin-Löf, hemos visto cómo la correspondencia de Curry-Howard proporciona un paradigma potente con el que formalizar nociones fundamentales en matemáticas: el paradigma de las proposiciones como tipos. La teoría de Martin-Löf sienta las bases de las dos teorías más influyentes en el funcionamiento de Lean: el Cálculo de Construcciones de Coquand y Huet y las Familias Inductivas de Dybjer. Lean toma varias ideas del Cálculo de Construcciones, como la impredicatividad del tipo de las proposiciones, la filosofía de tratar a los tipos como términos y la noción de reducibilidad de los términos. Por otro lado, Lean utiliza de forma directa el trabajo de Dybjer para la implementación de la inducción y la recursión en el sistema; las Familias Inductivas están introducidas en Lean con el nombre de familias de tipos inductivos. En el capítulo 2 se han estudiado las propiedades y particularidades de la teoría de tipos de Lean. Este estudio se ha complementado con fragmentos de código escritos en el propio lenguaje de Lean. A lo largo del capítulo 2 hemos visto cómo la correspondencia de Curry-Howard aparece de forma explícita en la lógica subyacente de Lean, tanto en las definiciones de los diferentes tipos y proposiciones como en la analogía entre teoremas (construcción de términos cuyo tipo es una proposición) y definiciones (construcción de términos que representan datos). Finalmente, en el capítulo 3 se ha puesto en práctica la maquinaria de tipos de Lean para formalizar algunos resultados y nociones elementales sobre Teoría de Conjuntos y Topología.

4.1 Posibles futuras líneas de trabajo

El modo táctica

El segundo capítulo es esencialmente un tutorial de Lean centrado en la demostración de teoremas por medio de la construcción explícita de términos. Siempre que ha sido posible hemos dejado a un lado la parte de Lean correspondiente a la demostración automática, esto es, las tácticas. Por esta razón, y a pesar de que los conceptos y demostraciones tratados en capítulo 3 son muy básicos, algunas de las pruebas presentadas son considerablemente tediosas de escribir, y aún más tediosas de leer.

En definitiva, para formalizar matemáticas modernas en Lean, igual que sucede en otros demostradores similares, es inevitable el uso de tácticas automáticas. Ejemplo de ello son las pruebas que pueden encontrarse en la librería mathlib. No obstante, esto no significa que el conocimiento de teoría de tipos y la capacidad para construir términos explícitos no sean útiles en la formalización de matemáticas en Lean. Más bien, la construcción explícita de términos y el modo táctica deben verse como herramientas complementarias. Del mismo modo que escribir pruebas usando exclusivamente teoría de tipos puede ser muy tedioso, escribir pruebas por medio de tácticas sin conocer la teoría subyacente en ocasiones puede resultar frustrante y confuso, pues a menudo intentaremos tácticas fallidas sin saber analizar qué ha ido mal.

El modo táctica de Lean constituye un tema lo suficientemente complejo como para ser objeto de estudio en profundidad. Un mayor conocimiento de las técnicas y algoritmos implementados en las tácticas de Lean proporcionaría un mejor desempeño a la hora de comprender y elaborar formalizaciones. Además Lean ofrece a los usuarios la posibilidad de diseñar e implementar sus propias tácticas; esta funcionalidad también puede ser objeto de estudio.

Matemáticas en Lean

La comunidad de Lean ha demostrado, tanto en mathlib como en otros proyectos[1], que Lean posee capacidad expresiva suficiente como para formalizar matemáticas muy profundas de forma relativamente cómoda. Tomando como punto de partida el conocimiento recopilado en este trabajo, futuros proyectos podrían tratar de formalizar nuevas teorías matemáticas en Lean, ya sea con el objetivo de hacer nuevas aportaciones a mathlib, o como proyectos independientes.

[1] https://leanprover-community.github.io/lean_projects.html

A | Algunos detalles de Lean

A.1 Contextos

A lo largo de esta sección nos referiremos indistintamente a la definición de términos y a la demostración de teoremas. Sean `α` y `β` tipos y supongamos que dado `a : α`, la expresión `t(a)` denota a un término de tipo `β`. Hay dos formas equivalentes de escribir la definición de un término `foo` de tipo `α → β`:

```
def foo : α → β := fun a => t(a)

def foo (a : α) : β := t(a)
```

La diferencia es el *contexto* del que partimos a la hora de escribir la definición. En el primer caso, el contexto del que partimos está vacío, y cuando escribimos `:=` Lean espera un término de tipo `α → β`. En el momento en el que escribimos `fun a =>`, la variable `(a : α)` [1] se introduce en el contexto, y Lean espera a partir de ese punto un término de tipo `β`. En cambio, en el segundo caso, el contexto inicial contiene la variable a de tipo `α`, luego al escribir `:=` la meta es directamente un término de tipo `β`.

Podemos ver tanto el contexto como la meta en cada punto de una definición escribiendo `_` el punto que queramos consultar.

```
example {p q : Prop} (hp : p) (hq : q) : p ∧ q := _
```

Si situamos el cursor al final de la línea de código del ejemplo anterior, Lean mues-

[1] Observemos que no es necesario declarar el tipo de `a` en `fun a =>`, pues Lean está esperando un término de tipo `α → β` y sabe por tanto que la próxima variable introducida por λ-abstracción será de tipo `α`.

tra por pantalla el contexto (antes de `⊢`) y la meta (después de `⊢`):

```
p q : Prop
hp : p
hq : q
⊢ p ∧ q
```

Siempre que sea posible, Lean añade automáticamente al contexto cualquier información necesaria que el usuario haya omitido. Por ejemplo, consideremos las siguientes pruebas de que `p` y `q` implican `p ∧ q`:

```
theorem prueba1 : p → q → p ∧ q := fun hp hq => And.intro hp hq

theorem prueba2 (hp : p) (hq : q) : p ∧ q := And.intro hp hq

theorem prueba3 {p q : Prop} (hp : p) (hq : q) : p ∧ q :=
  And.intro hp hq
```

En los teoremas `prueba1` y `prueba2` no se especifica en ningún momento qué son las variables `p` y `q`. Dicho de otro modo, el término `p → q → p ∧ q` declarado para el tipo de `prueba1` no es un término cerrado y el contexto `(hp : p) (hq : q)` declarado en `prueba2` no es válido (pues no es cerrado). Sin embargo, Lean no muestra ningún error y, de hecho, si comprobamos con `#check` los tipos de `@prueba1`, `@prueba2` y `@prueba3` [2], veremos que los tres son de tipo `∀ {p q : Prop}, p → q → p ∧ q`. Es decir, Lean añade automáticamente las variables `{p : Prop} {q : Prop}` (o, equivalentemente, `{p q : Prop}`) al contexto si no las especifica el usuario.

A.1.1 Los comandos `variable` y `universe`

Cuando utilizamos el comando `variable` para declarar una o varias variables, en realidad lo que estamos haciendo es introducir esas variables en el contexto de las definiciones que se escriban en lo sucesivo. Lean detecta de forma automática qué términos del contexto se han usado en una definición y los añade al tipo del término definido.

[2] Recordemos que debemos colocar el símbolo `@` antes de definiciones y teoremas que tengan argumentos implícitos para hacerlos explícitos.

```
variable (α : Type)

def n : Nat := 0

def empty : List α := []

#check (n : Nat)
#check (empty : (α : Type) → List α)
```

Vemos que la variable `α` se ha añadido al tipo de `empty` pero no al de `n`.

El comando `universe` funciona de forma muy parecida a `variable`; la diferencia es que `universe` se usa exclusivamente para declarar variables de universos (que no pueden ser declaradas con `variable`).

A.1.2 Los comandos `have` y `let`

Cuando estamos demostrando una teorema, puede ser útil declarar resultados intermedios que nos ayuden a escribir la prueba y que la hagan más legible para el lector. Con `have` podemos hacer justamente eso; el comando `have h : p := ...` introduce al contexto de una prueba la variable `h` de tipo `p`, aunque `p` no forme parte de la meta actual de la prueba. Los puntos suspensivos `...` han de constituir una prueba válida de `p`.

```
theorem ejemplo : ¬(p ↔ ¬p) :=
  fun h : p ↔ ¬p =>
    have hnp : ¬p := (fun hp : p => h.mp hp hp)
    hnp (h.mpr hnp)
```

En el ejemplo anterior se utiliza el comando `have` para introducir al contexto una prueba `hnp` de `¬p`. El tipo de la variable introducida por `have` puede omitirse siempre que Lean pueda inferirlo a partir de su propia definición (es decir, a partir de lo escrito tras `:=`). En este caso podríamos haber escrito lo siguiente:

```
theorem ejemplo : ¬(p ↔ ¬p) :=
  fun h : p ↔ ¬p =>
    have hnp := (fun hp : p => h.mp hp hp)
    hnp (h.mpr hnp)
```

También es posible utilizar el comando `have` sin declarar un nombre para la variable introducida. En ese caso Lean introduce la variable al contexto con un nombre interno al que el usuario no tiene acceso. Esta opción no tiene mucha utilidad a no ser que estemos usando el modo táctica de Lean (ver Sección A.3).

Por otro lado, dada una expresión `b` de tipo `α`, el comando `let a : α := b` (de nuevo podemos omitir el tipo `α` siempre que pueda inferirse) puede utilizarse en una prueba o definición a modo de abreviatura: la expresión `let a := b; c` (donde `;` puede sustituirse por un salto de línea) es equivalente a la expresión `c[b/a]`, es decir, a la expresión `c` donde cada ocurrencia de `a` se ha sustituido por `b`. Recordemos que la expresión `(fun a : α => c) b` también es definicionalmente equivalente a `c[b/a]`. Sin embargo, `let a := b; c` y `(fun a : α => c) b` son muy distintos en el fondo: para que la segunda expresión sea válida, `c` debe tener sentido para cualquier `a` de tipo `α`, mientras que en la primera basta con que tenga sentido para el término concreto `b`.

```
def foo1 := let a := Nat; fun x : a => x + 2

def foo2 := (fun a => fun x : a => x + 2) Nat
```

La primera definición es correcta mientras que la segunda causa un error.

No debemos confundir el comando `have` con el comando `let`. Aunque el resultado de utilizar `let` es muy parecido al de utilizar `have`, hay una diferencia muy importante: `let` no añade ninguna variable al contexto de la definición. El uso de `let` es puramente sintáctico. Por lo general, `have` se usa para introducir pruebas de resultados intermedios en el contexto, mientras que `let` se usa para declarar definiciones locales.

A.2 Estructuras

En la sección 2.3.2 vimos que un tipo inductivo con múltiples argumentos es disyuntivo, mientras que un único constructor con varios argumentos introduce información conjuntiva. Los tipos inductivos no recursivos con un único constructor son tan frecuentes que tienen nombre propio en Lean: las estructuras. El tipo producto `Prod` y su versión dependiente, `Sigma`, son ejemplos de estructuras. Cuando definimos una estructura `S`, es natural definir inmediatamente proyecciones que nos permitan acceder a cada uno de los campos almacenados en términos de tipo `S`. Por

ejemplo, los términos `Prod.fst` y `Prod.snd`, que devuelven la primera y segunda componente de un par, son las proyecciones asociadas al tipo inductivo `Prod`.

El comando `structure` nos permite definir una estructura `S` dando una lista de los campos que la conforman:

```
structure S (α :: σ) where
  field₁ : β₁
  ...
  fieldₙ : βₙ
```

Lo anterior es equivalente a definir un tipo inductivo `S` con un único constructor `mk : (field₁ : β₁) → ... → (fieldₙ : βₙ)` junto con las proyecciones `S.field₁ : S (α :: σ) → β₁, ... , S.fieldₙ : S (α :: σ) → βₙ`.

Consideremos por ejemplo la siguiente estructura:

```
structure Point (α : Type u) where
  x : α
  y : α
```

A partir de la definición anterior Lean genera automáticamente los correspondientes constructores, recursores y proyecciones:

```
#check Point       -- El tipo definido
#check @Point.rec  -- El recursor
#check @Point.mk   -- El constructor
#check @Point.x    -- Una proyección
#check @Point.y    -- Una proyección
```

Nótese que dado un punto `p : Point α`, podemos escribir `p.x` en lugar de `Point.x p` para obtener la primera coordenada de `p`.

Volviendo a la definición general de una estructura `S`, podemos crear un objeto de tipo `S (α :: σ)` escribiendo `S.mk b₁ ... bₙ`, donde cada `bᵢ` es de tipo `βᵢ`. Sin embargo, esto exige recordar el orden en el que se definieron los campos de S, así que puede ser muy tedioso para estructuras con muchos campos. Lean permite definir objetos de tipo `S (α :: σ)` usando los nombres de cada campo mediante la siguiente sintaxis:

```
{ field₁ := b₁, ... , fieldₙ := bₙ : S (α :: σ) }
```

Usando la sintaxis anterior, los campos `fieldᵢ` no necesitan estar en orden. Además, el sufijo `: S (α :: σ)` puede omitirse siempre y cuando el nombre de la estructura pueda inferirse. Por ejemplo, volviendo a la estructura `Point`, las siguientes líneas de código generan el mismo objeto:

```
#check Point.mk 10 20
#check { x := 10, y := 20 : Point Nat }
#check { y := 20, x := 10 : Point Nat }
#check ({ x := 10, y := 20 } : Point Nat)
```

Constructores anónimos

Cuando Lean espera un término cuyo tipo es un tipo inductivo con un único constructor no recursivo podemos escribir dicho término usando un constructor anónimo. Un constructor anónimo con *n* argumentos tiene la forma `⟨arg₁, arg₂, ... , argₙ⟩`[3]. En este caso el orden de los argumentos sí es importante. Esto nos permite, por ejemplo, escribir `⟨hp,hq⟩` en vez de `And.intro hp hq` cuando Lean espera una prueba de `p ∧ q` (suponiendo `hp : p` y `hq : q`).

El siguiente ejemplo muestra cómo los constructores anónimos pueden usarse para escribir pruebas y definiciones de forma más compacta, aunque a costa de (potencialmente) ofuscar la legibilidad de la prueba.

```
example : p ∧ q ↔ (q ∧ p) ∧ p :=
  ⟨fun ⟨hp,hq⟩ => ⟨⟨hq,hp⟩,hp⟩, fun ⟨⟨hq,hp1⟩,hp2⟩ => ⟨hp1,hq⟩⟩
```

Nótese que en el ejemplo anterior se han usado los constructores anónimos para formar términos de dos tipos distintos: de tipo `Iff` y de tipo `And`.

Es importante también notar que en el ejemplo anterior la expresión `fun ⟨hp,hq⟩ => ⟨⟨hq,hp⟩,hp⟩` es azúcar sintáctico para la siguiente expresión:

[3] Los paréntesis angulares pueden escribirse en Lean usando `\<` y `\>`.

```
fun h =>
  match h with
  | ⟨hp,hq⟩ => ⟨⟨hq,hp⟩,hp⟩
```

Un caso ligeramente especial de tipo inductivo no recursivo con un solo constructor, y con el que a menudo usaremos los constructores anónimos, es `Exists`. Recordemos la definición de `Exists`:

```
inductive Exists {α : Sort u} (p : α → Prop) : Prop where
  | intro (w : α) (h : p w) : Exists p
```

Es decir, el constructor `Exists.intro` toma como entrada un único dato `w` junto con una prueba `h` de `p w`, dando lugar a un término de tipo `Exists p` o, equivalentemente, de tipo `∃ a, p a`. Sin embargo, Lean nos permite escribir expresiones como `∃ (a : α) (b : β) : p a ∧ q b` para la proposición "existen `a` y `b` tales que se cumple `p a ∧ q b`". En realidad la expresión `∃ (a : α) (b : β) : p a ∧ q b` es azúcar sintáctico para los siguientes términos equivalentes:

```
@Exists α (fun a => @Exists β (fun b => p a ∧ q b))

∃ (a : α), (∃ (b : β), p a ∧ q b) -- es equivalente al anterior
```

Por lo tanto, si tenemos los términos `a : α`, `b : β`, `hpa : p a` y `hqb : q b`, podemos crear una prueba de `∃ (a : α) (b : β) : p a ∧ q b` mediante constructores anónimos escribiendo `⟨a, ⟨b, And.intro hpa hqb⟩⟩`. De hecho, Lean asocia de forma automática los constructores anónimos de derecha a izquierda, así que podríamos escribir directamente `⟨a, b, And.intro hpa hqb⟩`, `⟨a, b, ⟨hpa, hqb⟩⟩` o incluso `⟨a, b, hpa, hqb⟩`.

A.3 El modo táctica

Cuando Lean espera un término, el usuario tiene esencialmente dos opciones para proporcionarlo: construirlo explícitamente usando el lenguaje de la teoría de tipos dependientes, o darle instrucciones a Lean de cómo construir tal término. Esas instrucciones reciben el nombre de *tácticas*.

Aunque el presente trabajo se ha centrado en la construcción explícita de términos, las tácticas constituyen una herramienta fundamental a la hora de formalizar matemáticas complejas en Lean. En esta sección se explica lo mínimo necesario para entender las formalizaciones del capítulo 3. Se recomienda consultar el capítulo 6 de [8] para leer sobre las tácticas de Lean en mayor profundidad.

Siempre que Lean espere un término, al que llamamos *meta* u *objetivo*, podemos activar el modo táctica mediante la palabra clave `by`. Las pruebas en el modo táctica consisten en una secuencia `<tactics>` de comandos (o tácticas) separadas por `;` o por saltos de línea[4].

Consideremos por ejemplo la siguiente prueba:

```
theorem prueba1 (p q : Prop) (hp : p) (hq : q) : p ∧ q ∧ p :=
  And.intro hp (And.intro hq hp)
```

La prueba análoga en modo táctica sería la siguiente:

```
theorem prueba2 (p q : Prop) (hp : p) (hq : q) : p ∧ q ∧ p :=
  by apply And.intro
     exact hp
     apply And.intro
     exact hq
     exact hp
```

Intuitivamente la táctica `apply` aplica a la meta una expresión, vista como una función de cero o más argumentos. Más concretamente, `apply` unifica dicha expresión con la meta actual y crea nuevas submetas para los argumentos restantes. En el ejemplo anterior, el primer comando `apply And.intro` da lugar a las siguientes submetas:

```
    case left
    p q : Prop
    hp : p
    hq : q
    ⊢ p
```

[4]Si usamos saltos de línea para la secuencia de tácticas, esta debe estar alineada verticalmente para que Lean la interprete correctamente. Lean ofrece herramientas sintácticas para que este tipo de pruebas sean fácilmente legibles. Se recomienda consultar el capítulo 5 del tutorial de Lean [4] para más detalles.

```
    case right
    p q : Prop
    hp : p
    hq : q
    ⊢ q ∧ p
```

La primera meta se consigue con el comando `exact hp`. La táctica `exact` es una variante de `apply` que indica a Lean que la expresión proporcionada tiene que resolver la meta de forma exacta.

Una vez concluida una prueba en modo táctica, podemos ver el término explícito que Lean ha encontrado para la prueba usando el comando `print`:

```
#print prueba2

-- La prueba de Lean:
theorem prueba2 : ∀ (p q : Prop), p → q → p ∧ q ∧ p :=
  fun p q hp hq =>
    { left := hp, right := { left := hq, right := hp } }
```

Es importante notar que podemos activar el modo táctica en cualquier punto de la prueba en el que Lean espere un término. Dicho de otra forma, podemos alternar entre la construcción explícita de términos y el modo táctica. Por ejemplo, podría escribirse la prueba anterior de la siguiente forma:

```
example (p q : Prop) (hp : p) (hq : q) : p ∧ q ∧ p :=
  And.intro hp (by apply And.intro ; exact hq ; exact hp)
```

A.3.1 La táctica `simp`

En la librería estándar de Lean hay numerosas identidades etiquetadas con el atributo `[simp]`. La táctica `simp` usa esas identidades para reescribir de forma iterada subtérminos en la expresión de la meta principal hasta concluir la prueba.

```
example (x y z : Nat): (x + 0) * (0 + y * 1 + z * 0) = x * y :=
  by simp

example (x y z : Nat) (p : Nat → Prop) (h : p (x * y))
```

```
        : p ((x + 0) * (0 + y * 1 + z * 0)) :=
by simp [h]
```

En el primer ejemplo la parte izquierda de la igualdad se simplifica usando identidades relativas a 0 y 1, reduciendo así la meta a `x * y = x * y`. Finalmente `simp` utiliza la reflexividad de la igualdad para concluir la prueba. En el segundo ejemplo se utiliza `simp [h]` para indicar que la táctica `simp` también debe hacer uso de la hipótesis `h` en el proceso de simplificación, además de los teoremas de la librería marcados en con el atributo `[simp]`. En este caso Lean reduce la meta a `p (x * y)` y concluye la prueba usando `h`.

A.3.2 La táctica `contradiction`

La táctica `contradiction` resuelve la meta principal cuando hay hipótesis en el contexto que son "trivialmente contradictorias". Esto incluye los siguientes casos:

- Una de las hipótesis es de un tipo inductivo vacío (como `False`).

```
example (p : Sort u) (h : False) : p :=
  by contradiction
```

- Una de las hipótesis es de la forma `c_1 ... = c_2 ...`, donde `c_1` y `c_2` son constructores distintos de un mismo tipo inductivo[5].

```
example (p : Sort u) (h : none = some 0) : p :=
  by contradiction
```

- Una de las hipótesis es una proposición decidible falsa.

```
example (p : Sort u) (h : 2 + 2 = 3) : p :=
  by contradiction
```

- Las hipótesis contienen la prueba de una proposición y la prueba de su negación.

[5]Esto es contradictorio porque los constructores de un tipo inductivo son inyectivos; dos constructores distintos dan lugar a elementos distintos.

```
example (p : Prop) (h : p) (h' : ¬ p) : q :=
  by contradiction
```

- Una de las hipótesis es contradictoria con la propiedad reflexiva de la igualdad.

```
example (p : Sort u) (x : Nat) (h : x ≠ x) : p :=
  by contradiction
```

Bibliografía

[1] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2):95–120, 1988.

[2] Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6:440–465, 01 1994.

[3] P. Martin-Löf and G. Sambin. *Intuitionistic Type Theory*. Lecture notes. Bibliopolis, 1984.

[4] J. Avigad, L. De Moura, S. Kong, and S. Ullrich. Theorem Proving in Lean 4. https://leanprover.github.io/theorem_proving_in_lean4/, 2022.

[5] The mathlib Community. The Lean Mathematical Library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, Jan 2020.

[6] N.G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972.

[7] Thierry Coquand and Gérard Huet. Constructions: A higher order proof system for mechanizing mathematics. In Bruno Buchberger, editor, *EUROCAL '85*, pages 151–184, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg.

[8] J. Avigad, G. Ebner, and S. Ullrich. The Lean Reference Manual. https://leanprover.github.io/reference/tactics.html, 2018.