Universidad de Cantabria

Facultad
de
Ciencias

FORMALIZACIÓN DEL ALGORITMO DE DIVISIÓN MULTIVARIADA EN LEAN 4

(Multivariate Division Algorithm in Lean 4)

Trabajo de Fin de Máster
para acceder al
MÁSTER EN MATEMÁTICAS Y COMPUTACIÓN

Autor: David Ignacio Alcántara García

Director: Luis Felipe Tabera Alonso

Septiembre - 2023

Resumen

Lean es un asistente de demostración interactivo y un lenguaje de programación funcional muy expresivo, que permite formalizar y verificar demostraciones matemáticas, y construir herramientas para interactuar con estas en un mismo entorno.

En este trabajo, presentamos el Cálculo de las Construcciones Inductivas sobre el que se fundamenta la axiomática de Lean, introduciremos brevemente algunas de las capacidades del lenguaje que serán de utilidad, e introduciremos Mathlib, un esfuerzo colectivo por formalizar las matemáticas contemporáneas en Lean. Finalmente, presentaremos una formalización propia del algoritmo de división multivariada en Lean 4, junto con una prueba de su corrección.

Palabras clave: Lean, Cálculo de las Construcciones Inductivas, Correspondencia de Curry-Howard, Demostración Interactiva de Teoremas, Corrección de Algoritmos.

Abstract

Lean is an interactive theorem prover and a very expressive functional programming language, which can be used to formalize and verify mathematical proofs and build tools to interact with these in a single environment.

In this document, we present the Calculus of Inductive Constructions on which Lean's axiomatic foundation is based, we briefly present some of the language's capabilities of interest, and introduce Mathlib, a collective effort to formalize contemporary mathematics in Lean. Finally, we present our own formalization of the multivariate division algorithm in Lean 4, along with a proof of its correctness.

Keywords: Lean, Calculus of Inductive Constructions, Curry-Howard Correspondence, Interactive Theorem Proving, Algorithm Correctness.

Índice

Capítulo 1

Introducción

Una de las principales tareas de las matemáticas es la formalización su razonamiento. Desde tiempos de Euclides, la manera en la que escribimos las matemáticas ha visto muchos cambios. A finales del siglo XIX, las ideas de Cantor y Russell entre otros, motivaron la búsqueda de una formalización robusta de las matemáticas, que no diera lugar a paradojas.

En 1920, Hilbert plantea el problema de encontrar un sistema formal que permita formalizar y demostrar cualquier resultado matemático. Esta búsqueda vio su fin cuando en 1931 Gödel publica sus Teoremas de Incompletitud, demostrando que siempre existirán afirmaciones indecidibles que no se siguen de los axiomas en cualquier sistema axiomático suficientemente complejo. Años después, los resultados de indecidibilidad de Turing probarían que ni siquiera es posible construir un algoritmo general que sea capaz de decidir la veracidad de todas las afirmaciones matemáticas.

A pesar de estos resultados, hay mucho que ganar en la búsqueda de diferentes fundamentos para el razonamiento matemático. Los asistentes de demostración interactivos permiten verificar la corrección de demostraciones existentes, y en algunos casos, facilitar la búsqueda de nuevas demostraciones. La doble naturaleza de algunos de estos sistemas como lenguajes de programación permite aplicar la capacidad formal de las matemáticas al desarrollo de programas correctos, y por contraparte, desarrollar herramientas para construir e interactuar con resultados y demostraciones.

En este trabajo, hemos utilizado uno de estos sistemas, Lean 4, para formalizar el algoritmo de división multivariada, demostrando su corrección y terminación. Dado que el Cálculo de Construcciones Inductivas y la Teoría de Tipos, sobre los que se fundamenta este lenguaje, son conceptos que no son ampliamente conocidos en la comunidad matemática, al encontrarse a medio camino entre estas y la Ciencia de la Computación, los presentaremos brevemente en el Capítulo 2. En el Capítulo 3, introduciremos Lean 4, describiendo algunas de sus capacidades, así como Mathlib, un esfuerzo colectivo por formalizar las matemáticas contemporáneas en Lean, sobre el que construimos nuestras demostraciones.

Finalmente, en el Capítulo 4, presentaremos nuestra formalización del algoritmo, detallando el proceso que hemos seguido y algunas de las dificultades encontradas.

El desarrollo de los capítulos 2 y 3 sigue en parte la excelente introducción a la demostración de teoremas en Lean presentada en *Theorem Proving in Lean 4* [1], complementado con los libros *Functional Programming in Lean*[3], *Lean 4 Metaprogramming Book*[10], así como la propia fuente de Lean 4 [7] y Mathlib [4].

En este documento, aparecerán ejemplos de demostraciones y código escrito en Lean 4. Durante los primeros capítulos, estos ejemplos serán introducidos siempre de manera gradual, pues no se asume el conocimiento previo de Lean 4 por parte del lector.

Sin embargo, la notación que aparecerá en las demostraciones del Capítulo 4 es demasiado compleja como para introducirla en detalle. Por brevedad, con frecuencia omitiremos partes poco relevantes de las demostraciones que incluímos, utilizando la palabra `sorry`, dedicada en Lean para este propósito. El código completo, en el que aparecen todas las demostraciones que presentamos en este documento, sin omisiones, se encuentra disponible en un repositorio de GitHub [1], que contiene instrucciones sobre como utilizar Lean paar leer y comprobar las demostraciones.

Las demostraciones escritas en Lean 4 no están pensadas para ser leídas directamente, sino con la asistencia de un editor adecuado, capaz de proporcionar información adicional sobre el contexto de una demostración en cada paso. Incluso con esta ayuda, las demostraciones de este estilo, completamente formales, son difíciles de leer, y a menudo involucran numerosos resultados intermedios de poca relevancia, que serían considerados triviales por un matemático en otro contexto.

Por estos motivos, facilitamos un apéndice en el que hemos recopilado una descripción clásica de todos los resultados que hemos formalizado en Lean 4.

[1] `https://github.com/endorh/mv-divide`

Capítulo 2

El Cálculo de Construcciones Inductivas

El sistema axiomático sobre el que se construye Lean 4, es el Cáculo de Construcciones Inductivas (CIC), introducido inicialmente por Thierry Coquand en 1989 en su versión original, el Cálculo de Construcciones [11]. La versión del CIC que presentamos en este capítulo es la utilizada por Lean 4.

2.1 El lenguaje del Cálculo de Construcciones Inductivas

El CIC es una extensión del λ-cálculo con tipos, dotado de un sistema de tipos dependientes particularmente expresivo que, como veremos en secciones posteriores, lo hace susceptible de ser utilizado como lenguaje para la formalización del razonamiento matemático. Sin embargo, en su fundamento, el CIC es comparativamente sencillo si tenemos en cuenta las aplicaciones que admite, y esta es precisamente la razón que lo hace aplicable en la práctica: es fácil verificar automáticamente que una demostración escrita en el CIC es correcta.

El objetivo de esta sección es introducir el lenguaje del CIC, si bien no entraremos a describir completamente la formalización, ya que esta es laboriosa y puede diferir entre distintas versiones. En esta presentación no asumiremos la familiaridad del lector con el λ-cálculo ni con la Teoría de Tipos, cuya aparición en esta sección será breve y superficial. El lector interesado en la formalización precisa del CIC puede consultar su sección dedicada en la documentación de Coq [11], o la tesis de Mario Carneiro sobre la Teoría de Tipos de Lean [2], que describe las diferencias entre esta y la usada en Coq.

Para mantener la consistencia en este documento, la notación que utilizaremos en adelante será la utilizada por Lean 4, que difiere en algunos aspectos de la tradicional en la literatura del λ-cálculo.

2.1.1 Expresiones

Al igual que el λ-cálculo sin tipos, el CIC describe un lenguaje de expresiones en el que una serie de reglas determinen qué expresiones están *bien formadas*, y, por tanto, tiene sentido considerar. Por ejemplo, las reglas más sencillas para construir expresiones, comunes al λ-cálculo y al CIC son.

1. Las variables (denotadas por nombres, `x`, `y`, `…`) son expresiones.

2. Si `A` y `B` son expresiones, la yuxtaposición, `A B`, es una expresión, que denota la *aplicación de* `A` *a* `B`, entendiendo `A` como una función.

Sin embargo, llegados a este punto, encontramos la primera diferencia entre el λ-cálculo con y sin tipos. La definición de una función en el λ-cálculo sin tipos solo precisa del nombre de un parámetro y una expresión, mientras que en el λ-cálculo con tipos el parámetro de una función debe estar restringido a un tipo.

3. Si `x` es una variable y `A` y `T` son expresiones, `λx : T ↦ A`, que denotará una función que recibe un parámetro de tipo `T` y devuelve el resultado de sustituir `x` por este parámetro en `A`, es una expresión.

Esta notación se conoce tradicionalmente como *abstracción* λ, y en la expresión `λx : T ↦ A` se dice que la variable `x` está ligada, a diferencia de ser una variable libre. En la práctica, dado que las expresiones se interpretan en un determinado contexto, se permite que una expresión contenga variables libres, si el contexto dota a estas de un significado. Sin embargo, por simplicidad, resulta más sencillo entender que una expresión solo está *bien formada* si no contiene variables libres, es decir, sin ligar, y que, en algunos casos casos, entenderemos que una variable puede estar *ligada a un contexto*.

Observamos que la elección de la letra `T` para la expresión del tipo de `x` tiene únicamente el objetivo de facilitar la lectura, y no denota ninguna diferencia conceptual entre `T` y `A`, por lo menos en lo que respecta a la formación de expresiones.

Distanciándonos finalmente del λ-cálculo con tipos, en el CIC las siguientes también son expresiones bien formadas:

4. Si `x` es una variable y `A` y `T` son expresiones, `∀x : T, A`, que denotará el tipo de funciones dependientes de `T` en `A`, es una expresión.

5. Si `u` es un número natural, `Sort u`, que denotará el universo de tipos de nivel `u`, es una expresión.

La notación `∀x : T, A`, denominada habitualmente *flecha dependiente*, al igual que la abstracción λ, liga la variable `x`, y admite muchas variaciones. Cuando `A` es una expresión que no involucra a la variable `x`, se denota simplemente por `T → A`, la *flecha no dependiente*. Asímismo, cuando `A` no es una proposición, una distinción que se explicará más adelante, por claridad se suele utilizar la notación `Πx : T, A`, proveniente de la teoría dependiente de

tipos, donde este concepto se denomina Π-tipo, si bien en Lean es más frecuentemente representado como `(x : T) → A` por su similitud y compatibilidad con la notación de la flecha no dependiente.

Como es habitual, tanto el cuantificador universal como la abstracción λ se suelen contraer por brevedad, de manera que escribimos `λx y : T ↦ A` en lugar de `λx : T ↦ λy : T ↦ A`, y `λ(x : T₁) (y : T₂) ↦ A` cuando los tipos de las variables sean diferentes. También entendemos que estos operadores son asociativos a derecha, de manera que `(x : T₁) → (y : T₂) → A` es equivalente a `(x : T₁) → ((y : T₂) → A)`.

Si bien las reglas que hemos presentado son suficientes para representar cualquier expresión del CIC, habitualmente las distintas implementaciones de este lenguaje añaden una serie de reglas adicionales para construir expresiones, que permiten en algunos casos reducir el tamaño de expresiones frecuentes, etiquetar expresiones con información externa al lenguaje o denotar huecos durante el proceso de construcción de una expresión completamente bien formada [10, §3].

2.1.2 Tipos

La principal diferencia entre el λ-cálculo sin tipos y aquel con tipos es la existencia de un segundo criterio de buena formación de las expresiones, la *corrección del tipo* de una expresión. Este segundo criterio también existe en el CIC, y es precisamente su complejidad la que distingue al CIC del λ-cálculo con tipos.

De manera similar a las reglas de formación de expresiones, este criterio se define a partir de una serie de reglas que permiten *juzgar* el tipo de una expresión. Sin embargo, estas reglas no se definen teniendo en cuenta únicamente la expresión juzgada, sino también un contexto, formado esencialmente por el juicio de los tipos de otras expresiones.

Tradicionalmente, un juicio de tipos se denota por $\Gamma\ C\ \vdash e : t$, donde C es el contexto, e la expresión juzgada y t el tipo que se afirma que tiene e; y las reglas de inferencia de juicios de tipos se denotan de manera similar a las reglas de deducción en la lógica clásica, separando por una línea horizontal las premisas de la conclusión. Sin embargo, dado que estas reglas no son el objeto de estudio de este documento, por claridad nos conformaremos con introducirlas desde la perspectiva de cómo se manifiestan en el lenguaje.

De manera similar a como exigimos que una expresión bien formada no tenga variables sin ligar (ya sea a un cuantificador o *al contexto*), para que una expresión tenga un tipo correcto, también exigiremos que exista un juicio del tipo de todas sus variables.

El juicio de tipos más sencillo, es el de las variables ligadas.

1. En la expresión `λx : T ↦ A`, juzgaremos que las ocurrencias de `x` en `A` tienen tipo `T`.
2. En la expresión `∀x : T, A`, juzgaremos que las ocurrencias de `x` en `A` tienen tipo `T`.

El segundo juicio más básico, que dará finalmente un significado a la notación de flecha dependiente es:

3. Si la expresión `A` tiene tipo `B`, juzgaremos que la expresión `λx : T ↦ A` tiene tipo `∀x : T, B`, donde tanto `A` como `B` pueden depender de `x`.

4. Si la expresión `A` tiene tipo `Sort u`, y `T` tiene tipo `Sort v`, juzgaremos que la expresión `∀x : T, A` tiene tipo `Sort (smax u v)`, donde `smax u v` denota el máximo `u` y `v` si `v` es distinto de `0`, y `0` en caso contrario.

Es decir, la notación de flecha dependiente describe el tipo de las funciones, conteniendo una descripción tanto del tipo del parámetro, `T`, como del resultado de aplicar la función.

4. Si el tipo de `f` es `∀x : T, B`, y el tipo de `y` es `T`, juzgaremos que el tipo de `f y` es `B`.

Por otro lado, en el CIC, los universos de tipos forman una jerarquía ascendente de tipos.

5. Si `u` es un número natural, juzgaremos que el tipo del universo `Sort u` es `Sort (u+1)`.

En base a estas reglas, si en una expresión todas las variables están ligadas y los tipos de todas las funciones concuerdan con los de sus parámetros, es posible inferir el tipo de la expresión, y diremos que la expresión tiene un tipo correcto.

No obstante, con estas reglas, las únicas expresiones con tipo correcto que se pueden construir son aquellas que involucran universos de tipos, pues los únicos tipos que aparecen en la conclusión de una regla son los universos y las flechas dependientes, que solo son capaces de combinar estos. La verdadera flexibilidad del sistema de tipos del CIC proviene de la existencia de una serie de reglas adicionales para introducir tipos en el contexto.

El criterio de corrección de tipos junto con estas reglas para introducir tipos forman la noción que habitualmente se denomina como un *sistema de tipos*, y la *Teoría de Tipos* es el área que estudia estos sistemas.

2.1.3 Tipos inductivos

En el CIC, solo existe una forma de introducir tipos, los denominados *tipos inductivos*, que dan su nombre a la teoría. En contraposición, a los universos de tipos se los denomina en ocasiones tipos básicos, y a los tipos formados por flechas dependientes, tipos funcionales.

La definición de los tipos inductivos varía entre distintas versiones del CIC, pues, como veremos en las próximas secciones, los tipos inductivos serán las herramientas principales para formalizar razonamientos matemáticos en el sistema de tipos, razón por la cual, sus distintas variaciones son el objeto de mucha investigación.

Además, la descripción formal de las reglas que permiten introducir un tipo inductivo utilizando el lenguaje de los juicios de tipo y reglas de inferencia es sumamente compleja y carente de interés para nuestro propósito. Por este motivo, en esta sección nos limitaremos a presentar de manera informal la forma en que se introducen estos tipos, concretamente en el caso de Lean 4.

Un tipo inductivo se define con un nombre y una serie de constructores, que describen las reglas básicas que permiten construir expresiones del tipo. El tipo inductivo define una constante con su mismo nombre. Desde el punto de vista de las expresiones es una variable (ligada al contexto), y, por tanto, una expresión, a la que debemos asignar un tipo.

El tipo de un tipo inductivo es siempre un tipo básico, es decir, un tipo inductivo debe pertenecer a un universo de tipos de la forma `Sort u`. Esta idea respeta la noción intuitiva de que los universos de tipos describen el tipo de los tipos.

Por ejemplo, el siguiente fragmento de código, describe la definición en Lean 4 del tipo `Bool`, al que pertenecen solo dos expresiones, `Bool.true` y `Bool.false`. A su vez el tipo `Bool` como expresión tiene tipo `Sort 1`.

```
inductive Bool : Sort 1
  | true : Bool
  | false : Bool

-- #check permite comprobar el tipo de una expresión
#check (Bool : Sort 1)
#check (Bool.true : Bool)
```

Los constructores de un tipo inductivo, en este caso `Bool.true` y `Bool.false`, son a su vez constantes del tipo que construyen. En este ejemplo, los constructores tienen directamente el tipo que construyen, pero también es posible declarar constructores con tipos funcionales, siempre y cuando tengan como tipo de destino el tipo que construyen, y el tipo de los parámetros no pertenezca a un universo superior al del tipo que estamos definiendo, si bien más adelante distinguiremos una excepción en el caso de `Sort 0`.

```
inductive BoolPair : Sort 1
  | make : Bool → Bool → BoolPair
-- `Bool → BoolPair` es abreviación de `∀x : Bool, BoolPair`
#check (BoolPair.make : ∀a : Bool, (∀b : Bool, BoolPair))
```

Sin embargo, la verdadera utilidad de los tipos inductivos, y a la que deben su nombre, es la posibilidad de que los constructores tomen como parámetros expresiones del propio tipo que construyen. Esto permite, por ejemplo, definir el tipo de los números naturales, de manera análoga a la axiomática de Peano.

```
inductive Nat : Sort 1
  | zero : Nat
  | succ : Nat → Nat

-- El número `2` es la expresión:
#check (Nat.succ Nat.succ Nat.zero : Nat)
```

Esta definición es, de hecho, la definición de los números naturales utilizada en Lean 4.

Finalmente, lo que hace a los tipos inductivos tipos dependientes es la posibilidad de definir *familias inductivas*, que solo se diferencian de los tipos inductivos básicos, en que el propio tipo definido no es directamente un tipo básico, sino un tipo funcional (con destino en un tipo básico).

```
inductive List : (α : Sort 1) → Sort 1
| nil : List α
| cons : α → List α → List α

-- Para cada `α : Sort 1`, `List α` es un tipo inductivo
#check (List Nat : Sort 1)
-- `List` tiene como tipo una flecha dependiente
#check (List : (α : Sort 1) → Sort 1)
-- La lista que contiene al 0 como único elemento es:
#check (List.cons Nat.zero List.nil : List Nat)
```

En cierto modo, podemos pensar en los tipos inductivos como constructores de algún universo de tipos, de manera que las reglas para introducir tipos inductivos se pueden entender también como reglas para introducir constructores de los tipos básicos.

Existen más variaciones de los tipos inductivos, como los tipos inductivos mutuos o compuestos, pero estas no aumentan realmente la flexibilidad del sistema de tipos, por lo que nos abstendremos de describirlos. Estas reglas, definen todas las formas que existen en el CIC de Lean para construir tipos, por lo que, en retrospectiva, un tipo en el CIC se puede caracterizar como una expresión cuyo tipo sea un tipo básico. Esto incluye a los tipos básicos, `Sort u`, los tipos funcionales, `∀x : T, A`, y a los tipos inductivos.

Sin embargo, para hacer estos tipos útiles, necesitaremos definir una forma general de operar con ellos, que estudiaremos en la próxima sección.

2.1.4 Reglas de eliminación

Hasta ahora, solo hemos visto reglas para introducir tipos y construir expresiones de dichos tipos. También hemos visto que la forma de construir expresiones de los tipos básicos se corresponden con las reglas para introducir tipos inductivos. Por completitud, es posible entender también la abstracción λ como el constructor de los tipos funcionales, es decir, de las flechas dependientes.

Desde esta perspectiva, el criterio de corrección de tipo que hemos introducido en el lenguaje se asemeja a un criterio de corrección gramatical como el de cualquier lenguaje natural, en el que asignamos a cada *palabra* una *categoría* (a cada expresión un tipo), y luego exigimos que la forma de componer palabras respete una serie de reglas gramaticales, definidas en términos de las categorías.

No obstante, como veremos, los tipos de este sistema, mucho más expresivos que una cantidad finita de categorías gramaticales, también describen una cierta interpretación semántica intrínseca, y, como se verá más adelante, computacional.

De manera análoga a las reglas de construcción que hemos estudiado, existen *reglas de eliminación* para cada tipo, salvo para los tipos básicos. Las reglas de eliminación describen en general de qué forma es posible acceder a partir de una expresión a las expresiones que fueron utilizadas para construirla.

La regla de eliminación más sencilla es la de los tipos funcionales. Dada una expresión de un tipo funcional, es decir, una función, sú única regla de eliminación es la aplicación a un elemento del tipo correcto, de manera que, si `λx : T ↦ A` tiene tipo `∀x : T, B`, e `y` tiene tipo `T`, `(λx : T ↦ A) y` tiene tipo `B[x/y]`, el resultado de sustituir `x` por `y` en `B`.

Análogamente, cada familia o tipo inductivo siempre está acompañado de una regla de eliminación, su *función recursora*, que permite construir funciones sobre expresiones del tipo.

La función recursora de un tipo inductivo no es más que una constante, que construye una función sobre el tipo a partir de funciones que tratan el caso de cada constructor por separado.

```
-- `def a : T := e` introduce la constante `a` con tipo `T` y valor `e`
def Bool.not : Bool → Bool :=
  λb : Bool ↦ Bool.casesOn b
    Bool.false -- Resultado de `Bool.not Bool.true`
    Bool.true  -- Resultado de `Bool.not Bool.false`

#eval Bool.not Bool.true -- Bool.false
```

Una función recursora más potente es la del tipo de los números naturales, pues describe esencialmente el axioma de inducción.

```
def Nat.add : Nat → Nat → Nat :=
  λa b : Nat ↦ Nat.brecOn b                -- Inducción sobre `b`
    a                                      -- Caso `0`
    (λ b' add_a_b' ↦ Nat.succ add_a_b')    -- Caso inductivo

#eval Nat.add Nat.zero (Nat.succ Nat.zero) -- 0 + 1 = 1
```

En esta definición de la suma de números naturales, el primer caso de `Nat.brecOn b`, define el resultado de la suma cuando `b` es `Nat.zero`, y el segundo caso, describe como definir la suma de `a` y `Nat.succ b'`, conocida la de `a` y `b'`, que se denota como `add_a_b'`.

En la práctica, sin embargo, Lean proporciona muchas facilidades para utilizar las funciones recursoras de tipos inductivos de manera mucho más intuitiva. Por ejemplo, la definición real de la suma de números naturales en Lean es esencialmente la siguiente:

```
def Nat.add : Nat → Nat → Nat
  | a, Nat.zero   => a
  | a, Nat.succ b => Nat.succ (Nat.add a b)
```

Por debajo, el lenguaje traduce esta notación recursiva, más fácil de entender, a una aplicación de la regla de eliminación para los naturales. En la sección 3.4 prestaremos más atención a esta traducción, pues será de interés en nuestra formalización del algoritmo de división multivariada.

De hecho, las funciones `casesOn` y `brecOn` que hemos presentado en estos ejemplos son alias simplificados de la verdadera regla de eliminación de sus tipos, `rec`, que hemos preferido no usar por claridad.

La naturaleza axiomática de las reglas de eliminación de los tipos inductivos es la que nos lleva a compararlas, en el caso de los tipos no finitos como `Nat`, con el axioma de inducción. El axioma de inducción obtiene, a partir de un método que permite construir una demostración para cada caso, una demostración general, mientras que las funciones recursoras construyen, a partir de un método que permite evaluar una función para cada caso, una función general.

2.1.5 Polimorfismo

Una última propiedad del CIC que debemos introducir es la noción de polimorfismo. El polimorfismo en un sistema de tipos puede definirse como la capacidad de que una expresión dependa de uno o más tipos.

En el CIC esto es natural, ya que todos los tipos son de por sí expresiones y pueden, por tanto, participar en otras expresiones. De hecho, ya hemos visto un ejemplo de polimorfismo en la definición de la familia inductiva `List α`, donde `α` no es sino un tipo del que depende `List α`.

Como ilustra este ejemplo, el polimorfismo permite definir construcciones genéricas como la noción de una lista, aplicables a no solo un tipo, sino en este caso a cualquier tipo de `Sort 1`. Para eliminar esta restricción, el CIC admite un tipo de polimorfismo aún más genérico, que hasta ahora hemos omitido por simplicidad, el polimorfismo a nivel de universos. En realidad, toda expresión puede involucrar una cantidad arbitraria de variables de nivel, que solo pueden aparecer en una expresión como argumento de `Sort`.

El objetivo de esta generalización es permitir extender nuestros tipos genéricos como `List` a múltiples universos. Por ejemplo, la verdadera definición de `List` en Lean es casi equivalente a:

```
universe u -- Usamos `u` como variable de nivel
inductive List (α : Sort u) : Sort u where
  | nil : List α
  | cons : (head : α) → (tail : List α) → List α

-- Las variables de nivel se sustituyen según sea necesario.
-- En la lista de tipos [Nat, Bool], `u` es `1`:
#check (List.cons Nat (List.cons Bool List.nil) : List (Sort 1))
#check (List (Sort 1) : Sort 2)
```

Podemos pensar en una expresión que involucra variables de nivel como una familia de expresiones, que describen la misma estructura en cada universo, y puede utilizarse libremente en cualquiera de ellos.

Un ejemplo de expresiones polimórficas en el CIC son las reglas de eliminación de los tipos inductivos, cuyo tipo hemos preferido no mencionar hasta ahora.

```
universe u
#check (Nat.recOn :
```

```
-- Tipo (dependiente del número) de la imagen
{result : ℕ → Sort u}
-- Número sobre el que *recurrimos*
→ (a : ℕ)
-- Imagen del `0`
→ (zero : result Nat.zero)
-- Imaget del sucesor de `n`, en función de la de `n`
→ (succ : (n : ℕ) → result n → result (Nat.succ n))
-- Tipo de la imagen
→ result a)
```

El parámetro `result` en realidad recibe el nombre `motive`, pero hemos preferido usar otro nombre por claridad.

En la descripción de este tipo, observamos que el parámetro `result` aparece descrito entre llaves. Esto indica que es un parámetro implícito, lo que permite aplicar la función `Nat.recOn` sin especificar el valor de `result` cuando este puede determinarse (por las reglas de inferencia de tipos) a partir del resto de parámetros.

La distinción entre parámetros implícitos y explícitos es ajena al sistema de tipos. En Lean, previamente al análisis del tipo de una expresión, existe una fase denominada *elaboración* en la que el lenguaje intenta rellenar distintos huecos y ambigüedades en expresiones. Este proceso de elaboración es muy complejo, pero es el factor más importante que hace el polimorfismo eficaz en la práctica, permitiendo definir funciones aplicables a múltiples tipos sin aumentar la complejidad que supone utilizarlas.

2.2 La Correspondencia de Curry-Howard

La motivación del desarrollo de sistemas de tipos en el diseño de lenguajes de programación era en su inicio una cuestión práctica. El sistema de tipos de un lenguaje limita el conjunto de expresiones admisibles, algo que permite detectar algorítmicamente muchas clases de errores comunes en la compleja tarea de escribir una expresión a la que queremos más adelante dar una interpretación computacional.

Con el tiempo, los sistemas de tipos utilizados para este propósito fueron creciendo en complejidad, con el objetivo de poder detectar cuantos más errores como fuera posible en los programas.

Remontándonos mucho antes, H. Curry empieza a construir en 1934 la Teoría de la Funcionalidad, dentro del marco de la Lógica Combinatoria de M. Schönfinkel. Esta teoría, como describe en su libro *Combinatoric logic* [5, p. 8C], intenta construir sobre el lenguaje formal de la lógica combinatoria lo que esencialmente es un sistema de tipos, a los que H. Curry llama en su momento *categorías*.

En el desarrollo de esta teoría, H. Curry observa una sorprendente analogía entre esta y el Álgebra Proposicional, probando que es posible traducir entre los axiomas de su teoría y los de la lógica intuicionista positiva (con la implicación como única conectiva), en ambas direcciones [5, 9E].

Inspirado por estas ideas, W. A. Howard describe en 1969 una correspondencia entre términos del λ-cálculo con tipos y derivaciones de la lógica intuicionista positiva, que asocia el tipo de los términos a proposiciones lógicas [6]. Eventualmente, esta pasaría a ser conocida como la *Correspondencia de Curry-Howard*.

Motivado por ella, el Cálculo de Construcciones, creado por T. Coquand, (sobre el que se construye el CIC), extiende esta equivalencia entre términos al caso de la lógica intuicionista de primer órden. El desarrollo de este sistema llevó a la creación de Coq y muchos otros lenguajes, entre ellos Lean, dando respuesta también a la búsqueda de un sistema de tipos capaz de describir cualquier clase de restricciones en un lenguaje de programación.

2.2.1 Proposiciones como tipos

La Correspondencia de Curry-Howard en el CIC identifica tipos con proposiciones lógicas, y, análogamente, expresiones de un tipo con demostraciones constructivas de una tesis.

El objetivo de esta correspondencia en el CIC es permitirnos escribir demostraciones en este lenguaje, donde pueden ser verificadas automáticamente, de una forma que permita traducir estas expresiones en demostraciones clásicas.

Con el objetivo de distinguir los tipos que describen proposiciones lógicas de otros tipos, como `Nat` o `List`, el universo `Sort 0` está reservado para los tipos que describen proposiciones, y se denota por `Prop`, el *tipo de las proposiciones*.

En contraste, el resto de universos se denotan por `Type`, de manera que `Type u` es equivalente a `Sort (u + 1)`.

En la lógica intuicionista, que no asume axiomas clásicos como el principio de exclusión del tercero ni la eliminación de la doble negación, la única forma de probar proposiciones es constructiva.

Por ejemplo, si p y q son proposiciones, para probar la implicación lógica $p \Rightarrow q$, es necesario dar una demostración que, a partir de una demostración de p, construye una demostración de q. No es posible probar en su lugar $\neg q \Rightarrow \neg p$, porque sin el axioma de eliminación de la doble negación, es imposible transformar una demostración del recíproco en una de $p \Rightarrow q$.

En el CIC, si `p` y `q` son proposiciones, es decir, tipos de `Prop`, la implicación entre `p` y `q` se corresponde con la flecha no dependiente `p → q`. De manera análoga a la lógica intuicionista, la única regla de introducción de este tipo es el la abstracción λ, que describe una función que, a partir de una demostración de `p`, construye una demostración de `q`.

Por ejemplo, la demostración de `p → (q → p)`, uno de los axiomas de la lógica intuicionista, es simplemente una función que toma una demostración de `p` y devuelve una función que ignora la demostración de `q`, devolviendo la demostración original de `p`.

```
theorem then_1 : p → (q → p) :=
  λhp : p ↦ (λhq : q ↦ hp)
```

En este ejemplo, `theorem` es equivalente a `def`, y simplemente define la constante `then_1`, con tipo `p → (q → p)` y el valor correspondiente. Como

en toda definición, Lean comprueba por nosotros que el tipo de la expresión coincide con el que esperamos.

Una versión más corta de esta demostración, contrayendo las abstracciones λ, dejando que el elaborador infiera el tipo de `hp` y omitiendo el nombre de `hq` sería `λhp _ ↦ hp`.

Una forma de entender como las expresiones de un tipo proposicional representan *demostraciones* es identificando la veracidad de la proposición que describe un tipo con su no vacuidad, es decir, con la existencia de una expresión del tipo. Una expresión del tipo es testigo de que el tipo es no vacío, y, desde el punto de vista de la lógica constructiva, de que la proposición es cierta.

Esta correspondencia se extiende al resto de conectivas lógicas mediante la introducción de familias de tipos inductivos en `Prop`. Por ejemplo, la conjunción lógica, $p \wedge q$, que en la lógica intuicionista solo se puede probar de manera constructiva a partir de una demostración de p y otra de q, ya que no nos sirve probar $\neg(\neg p \vee \neg q)$, se define como una familia inductiva con dos índices, `p` y `q`, y un único constructor, que requiere una demostración de `p` y otra de `q`:

```
inductive And : (p : Prop) → (q : Prop) → Prop
  | intro : (left : p) → (right : q) → And p q

notation p " ∧ " q => And a b
```

El comando `notation` permite introducir abreviaciones sintácticas en el lenguaje. La definición real del operador $\wedge$ en Lean es más compleja, pues define como debe combinarse con otros operadores, pero por simplicidad solo utilizaremos `notation`.

Esta definición de `And`, hace que su constructor coincida con la regla de introducción constructiva de la conjunción. Análogamente, su función recursora, coincide con las reglas de eliminación de la conjunción, permitiéndonos construir demostraciones que usan demostraciones de `p` o de `q` a partir de una demostración de `p ∧ q`.

```
#check (And.recOn : {p q : Prop} → {result : p ∧ q → Sort u}
  → (h : p ∧ q)
  → (intro : (left : p) → (right : q) → result h)
  → result h)

theorem and_comm : (p ∧ q) → (q ∧ p) :=
  λh_pq ↦ And.recOn h_pq (λhp hq ↦ And.intro hq hp)
```

Dado que `And` solo tiene un constructor, Lean nos permite aplicar esta regla de eliminación implícitamente, descomponiendo el constructor de `p ∧ q` a la izquierda de `↦`. Para ello, podemos usar la notación de constructor anónimo, `⟨a, b, ...⟩`, que construye (y deconstruye cuando se usa a la izquierda de `↦`) cualquier tipo inductivo con un único constructor:

```
theorem and_comm' : (p ∧ q) → (q ∧ p) :=
  λ⟨hp, hq⟩ ↦ ⟨hq, hp⟩
```

Por otro lado, la disyunción lógica, $p \vee q$, solo admite dos formas de demostración constructiva, a partir de una demostración de p, o a partir de una de q. Estas reglas de introducción se corresponden con los dos constructores de la familia inductiva `Or`:

```
inductive Or : (p : Prop) → (q : Prop) → Prop
  | inl : (left : p) → Or p q
  | inr : (right : q) → Or p q

notation p " ∨ " q => Or p q

theorem and_implies_or : (p ∧ q) → (p ∨ q) :=
  λ⟨hp, hq⟩ ↦ Or.inl hp
```

La regla de eliminación intuicionista de la disyunción, para probar algo a partir de $p \vee q$, requiere dar una demostración en cada caso, una asumiendo una demostración de p, y otra asumiendo una de q. Esta regla se corresponde con la función recursora de `Or`, que requiere proporcionar un caso para cada constructor, en cada cual es posible utilizar la demostración de `p` o de `q`:

```
#check (Or.elim : {p q r : Prop} → (h : p ∨ q)
  → (left : p → r) → (right : q → r) → r)

theorem or_comm : (p ∨ q) → (q ∨ p) :=
  λh_pq ↦ Or.elim h_pq
    (λhp ↦ Or.inr hp)
    (λhq ↦ Or.inl hq)
```

En este caso, también existe una notación más intuitiva para utilizar la regla de eliminación implícitamente.

```
theorem or_comm' : (p ∨ q) → (q ∨ p)
  | .inl hp => .inr hp
  | .inr hq => .inl hq
```

Por otro lado, la negación lógica en la lógica intuicionista se define como una implicación a $\bot$, de manera que $\neg p$ se traduce en $p \Rightarrow \bot$, donde $\bot$ denota la proposición falsa, para la que no existe demostración.

En el CIC, esta proposición se traduce en un tipo inductivo sin constructores:

```
inductive False : Prop
```

Esta idea concuerda con la identificación que hicimos entre la veracidad de una proposición y el que su tipo sea no vacío. Al no tener constructores, es imposible construir una expresión de tipo `False`.

En este caso, ocurre algo notable con la función recursora. Recordamos que, para construir una función que reciba una expresión de tipo `False`, su función recursora necesita que proporcionemos una forma de definir esta función por cada constructor de `False`. Dado que `False` no tiene constructores, la regla de eliminación no requiere definir ningún caso, y produce directamente una expresión de cualquier tipo. Esta regla de eliminación coincide con el principio de explosión, también presente en la lógica intuicionista.

```
#check (False.elim : {C : Sort u} → False → C)
```

Análogamente a `False`, también podemos definir un tipo que se corresponda con $\top$, la proposición trivialmente cierta. En este caso, el tipo inductivo `True` tiene un único constructor, que no requiere ningún parámetro. Es por definición una demostración de `True`.

```
inductive True : Prop
  | intro : True
```

En este caso, la regla de eliminación es inútil, pues no hay parámetros del constructor que nos permita utilizar, y se reduce a una abstracción λ.

Podemos probar `False → True` usando tanto la regla de introducción de `True` como la de eliminación de `False`:

```
theorem false_implies_true : False → True :=
  λhf ↦ True.intro
theorem false_implies_true' : False → True :=
  λhf ↦ False.elim hf
```

Con estas definiciones, al igual que en la lógica intuicionista, la negación lógica se define como una simple traducción de notación:

```
-- `Not` es una función que transforma proposiciones, no demostraciones
def Not : Prop → Prop :=
  λp ↦ p → False

-- Denotaremos `Not p`, es decir, `p → False` por `¬p`
notation "¬" p => Not p
```

Las reglas de introducción y eliminación de la negación coinciden, por tanto, con las de la implicación, que en el CIC son la abstracción λ y la aplicación de funciones, respectivamente.

Esto puede hacer que el siguiente ejemplo sea más difícil de leer que los anteriores:

```
theorem reciprocal : (p → q) → (¬q → ¬p) :=
  λh_pq    :  p → q ↦      -- Debemos construir `¬q → ¬p`
    λh_nq  : ¬q     ↦      -- Debemos construir `¬p`, i.e., `p → False`
      λh_p :  p     ↦      -- Suponemos `p` y queremos construir `False`
        h_nq (h_pq h_p)
```

En la última línea, `h_p` tiene tipo `p`, por lo que el resultado de aplicar `h_pq` a `h_p`, `(h_pq h_p)` es una expresión de tipo `q`. Finalmente, el resultado de aplicar `h_nq`, de tipo `q → False` a esta expresión tiene tipo `False`, como necesitábamos construir.

Sin axiomas adicionales, al igual que en la lógica intuicionista, es imposible dar una demostración constructiva del recíproco de esta implicación.

2.2.2 Lógica de primer orden

Todas las familias inductivas que hemos considerado en la sección anterior tenían sus índices en el universo de las proposiciones, `Prop`. Cuando consideramos familias con índices en `Sort 1`, la correspondencia de Curry-Howard identifica tipos con fórmulas de la lógica de primer órden intuicionista.

Por ejemplo, podemos definir el predicado de que un número natural sea positivo como

```
notation "ℕ" => Nat

inductive PosNat : ℕ → Prop
  | intro : (n : ℕ) → PosNat (Nat.succ n)
```

La regla de introducción de este predicado requiere proporcionar el predecesor de un número para probar que es positivo, y, análogamente, la regla de eliminación nos permite utilizar el predecesor de un número positivo para construir expresiones.

Finalmente, podemos observar la razón por la que los tipos funcionales dependientes se denotan con ∀. Cuando un tipo funcional dependiente tiene como resultado un tipo proposicional, está describiendo una proposición de primer orden con el cuantificador universal, pues sus expresiones son funciones que reciben un valor arbitrario del tipo cuantificado y construyen una demostración de la proposición especializada para este valor.

Por ejemplo, podemos probar que la propiedad de ser positivo se preserva por la suma:

```
notation a " + " b => Nat.add a b

example: ∀a b : ℕ, PosNat b → PosNat (a + b) :=
  λa b ↦ λ⟨b_pred⟩ ↦ ⟨a + b_pred⟩
```

En este ejemplo, `example` es equivalente a `theorem`, sin dar un nombre a la definición.

Dado que es frecuente escribir teoremas precedidos de múltiples cuantificadores, también es posible mover estos a la izquierda de los dos puntos que denotan el tipo del ejemplo o teorema, eliminando la necesidad de duplicar los nombres de `a` y `b` en el cuantificador ∀ y la abstracción λ:

```
example (a b : ℕ) : PosNat b → PosNat (a + b) :=
  λ⟨b_pred⟩ ↦ ⟨a + b_pred⟩
```

Como mencionamos brevemente en la sección 2.1.3, los tipos inductivos proposicionales en Lean son una excepción a la regla sobre el nivel que pueden tener los parámetros de sus constructores e índices. Cuando un tipo inductivo se declara en `Prop`, se permite que sus parámetros puedan estar en cualquier universo, pues de lo contrario no podríamos describir fórmulas de primer orden o superiores. Sin embargo, a cambio de esta libertad, sus reglas de eliminación solo pueden construir otras proposiciones.

```
#check (PosNat.rec : {result : (n : ℕ) → PosNat n → Prop}
  → ((n_pred : ℕ) → result (Nat.succ n_pred) (PosNat.intro n_pred))
  → {n : ℕ} → (h : PosNat n) → result n h)
```

En esta regla de eliminación, el tipo del resultado, `result`, está restringido a `Prop`. La razón para esta limitación proviene de la interpretación computacional que da Lean a las proposiciones, como se verá en el próximo capítulo.

No obstante, esta restricción puede levantarse en algunos casos, y, de hecho, ya lo hemos visto con `False`. Cuando un tipo o familia inductiva de `Prop` no tiene más de un constructor, y todos sus parámetros que no son proposiciones son índices, se permite que la regla de eliminación construya expresiones de cualquier tipo, ya que la función recursora no necesita extraer valores de la demostración que no sean proposiciones.

Esta propiedad del sistema de tipos de Lean se denomina *eliminación de subsingletons*, y en la siguiente sección veremos un ejemplo muy relevante.

2.2.3 La igualdad

Uno de los predicados más importante en las matemáticas, y que, como veremos, tiene propiedades bastante especiales en el CIC de Lean, es la igualdad. Como todo predicado proposicional, la igualdad se define en Lean como una familia inductiva:

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

notation a " = " b => Eq a b
```

Al igual que la conjunción y la disyunción, la igualdad depende de dos índices. Sin embargo, en este caso los índices no son tipos de `Prop`, sino expresiones de otro tipo, `α`, arbitrario, que forma un índice más, implícito, ya que puede inferirse de los otros.

No obstante, el constructor de la igualdad es diferente a todos los que hemos visto. A partir de una expresión `a`, `Eq.refl` construye una demostración de `a = a`. Es decir, `Eq.refl` es el axioma de reflexividad, y existen más formas de construir una igualdad.

Aunque esta regla de introducción parezca algo débil, es capaz de probar cualquier igualdad en la que ambos lados pueden reducirse a una misma forma normal del λ-cálculo por medio de aplicaciones.

```
-- Podemos denotar números naturales con la notación decimal habitual
example: 1 + 1 = 2 := Eq.refl 2
```

En este ejemplo, el resultado de reducir 2 veces la aplicación de `Nat.add` en el lado izquierdo tiene como resultado `Nat.succ (Nat.succ Nat.zero)`, que es exactamente `2`, por lo que `Eq.refl 2` es una demostración de `1 + 1 = 2`.

Aún más, en este caso el elaborador es capaz de inferir el argumento de `Eq.refl`, por lo que podrímos escribir simplemente `Eq.refl _`. Este tipo de

demostraciones por reflexividad son tan comunes que Lean define un alias, `rfl`, para esta forma de utilizar `Eq.refl` infiriendo el parámetro `a`.

Tener este único constructor, dota a esta definición de la igualdad de una interpretación intrínseca en el sistema de tipos del CIC. Dado que el único parámetro de `Eq.refl` es utilizado como índice en `Eq a a`, la regla de eliminación de la igualdad permite construir cualquier tipo, y, por la definición de `Eq.refl`, esta eliminación se corresponde precisamente con la noción de *substituir* el parámetro de un tipo dependiente por otro, a partir de una demostración de que son iguales.

Esto permite en la práctica substituir dentro de la expresión de un tipo cualquier subexpresión por otra, a partir de una demostración de que ambas sean iguales. Es asombroso que el CIC sea capaz de describir una noción tan fundamental de las matemáticas y la lógica, únicamente con las reglas elementales de su sistema de tipos.

```
#check (Eq.rec : {α : Sort u₁} → {a : α}
  → {motive : (b : α) → a = b → Sort u₂} → motive a (rfl : a = a)
  → {b : α} → (h : a = b)
  → motive b h)
```

Si ignoramos la dependencia de `motive` de la demostración concreta de `a = b`, fruto innecesario de la generalidad del sistema de tipos dependientes del CIC, a partir de una expresión de tipo `motive a`, y una demostración de que `a = b`, `Eq.rec` traduce nuestra expresión en una de tipo `motive b`, esencialmente substituyendo `a` por `b` en el tipo de la expresión.

Existe un alias con menos parámetros para esta función, eliminando esta dependencia innecesaria de la demostración concreta de la igualdad (que solo puede tener una forma, `Eq.refl a`). Esta operación de reescritura es tan frecuente, que tiene su propio operador en Lean.

```
#check (Eq.subst : {α : Sort u} → {motive : α → Prop}
  → {a b : α} → (h₁ : a = b)
  → (h₂ : motive a) → motive b)

-- `h ▸ e` *reescribe* el tipo de `e` usando la igualdad `h`
notation h " ▸ " e => Eq.subst h e
```

Usando esta regla, podemos probar, por ejemplo, que la imagen de dos expresiones iguales coincide para cualquier función.

```
theorem congrArg {α : Sort u} {β : Sort v} {a₁ a₂ : α}
    (f : α → β) (h : a₁ = a₂) : f a₁ = f a₂ :=
  h ▸ rfl
```

En esta demostración, `rfl` se traduce en `Eq.refl (f a₁)`, cuyo tipo es `f a₁ = f a₁`, y el mecanismo de inferencia de tipos encuentra un valor de `motive` que permite substituir `a₁` por `a₂` en el lado derecho del tipo de `rfl`.

Capítulo 3

Lean 4

Lean es un asistente de demostración de teoremas interactivo basado en el Cálculo de Construcciones Inductivas, creado por Leonardo de Moura en 2013 [1]. El nombre de Lean proviene de una de las acepciones inglesas de la palabra, *esbelto*, describiendo su sistema de tipos y conjunto de axiomas minimalista. Uno de los objetivos de Lean es unir los mundos de la demostración interactiva y automática de teoremas, facilitando el acceso interactivo a herramientas de automatización dentro de un contexto que permita la construcción de demostraciones.

Uno de los atractivos de Lean 4 frente a otros asistentes de demostración similares, es su elegante sistema de metaprogramación y su flexible notación.

La metaprogramación, consistente en manipular programas por medio de otros programas, se traduce por la correspondencia de Curry-Howard en la manipulación algorítmica de demostraciones. Por este motivo, es una técnica necesaria en cualquier asistente de demostración.

La razón por la que decimos que este sistema es elegante es porque, a diferencia de en otros sistemas similares, en Lean 4 la metaprogramación se escribe en el propio lenguaje. Definiciones, demostraciones y algoritmos que operan o construyen estas demostraciones, pueden convivir en el mismo entorno, y depender mutuamente unos de otros.

Para este propósito, Lean 4 ha sido diseñado con el objetivo de ser, no solo un asistente de demostración, sino también un lenguaje de programación funcional eficiente por sí mismo.

Debido a la elevada complejidad del lenguaje, su uso práctico, incluso para la mera lectura, puede requerir el uso de un editor adecuadamente integrado con el lenguaje. El sistema de metaprogramación de Lean dota al lenguaje de una capacidad de introspección que permite modificar de manera casi arbitraria desde el propio lenguaje la forma en la que editores de texto interaccionan con programas escritos en Lean.

En este capítulo, describiremos brevemente algunas de las características de Lean que juegan un papel fundamental en su aplicación a la demostración asistida de teoremas.

3.1 Tácticas

Las demostraciones que hemos mostrado hasta ahora en los ejemplos han sido todas muy sencillas. Sin embargo, la realidad es muy distinta. Debido a la formalidad que requiere el CIC, demostraciones intuitivamente simples, pueden traducirse en expresiones muy complejas.

Por ejemplo, aunque la demostración de que sumar `0` por la derecha es igual que no hacer nada es trivial por definición, la demostración de que lo mismo ocurre al sumar `0` por la izquierda requiere utilizar la función recursora de `Nat` para razonar por inducción, y utilizar `congrArg` .

```
theorem Nat.add_zero : ∀a : ℕ, a + 0 = a := rfl
theorem Nat.add_succ : ∀a b : ℕ, a + Nat.succ b = Nat.succ (a + b) := rfl
theorem Nat.zero_add : ∀a : ℕ, 0 + a = a
  | 0   => rfl
  | a+1 => congrArg Nat.succ (Nat.zero_add a)
theorem succ_add : ∀a b : ℕ, (Nat.succ a) + b = Nat.succ (a + b)
  | _, 0      => rfl
  | a, Nat.succ b => congrArg Nat.succ (Nat.succ_add a b)
```

Esta complejidad no deja de crecer si consideramos, por ejemplo, la demostración de que la suma de números naturales es conmutativa, en la que necesitamos realizar una cadena de substituciones dentro del paso de inducción:

```
theorem add_comm : ∀a b : ℕ, a + b = b + a :=
  λa b ↦ Nat.recOn b
    (Eq.symm (Nat.zero_add a))
    (λb' ih ↦
      Nat.add_succ a b'
      ▸ Nat.succ_add b' a
      ▸ congrArg Nat.succ ih)
```

Este problema solo crece a medida que añadimos más resultados y tipos diferentes. El principal problema que supone tratar con demostraciones formadas por expresiones tan complejas no es la dificultad de escribirlas, sino la que supone simplemente entenderlas.

Para simplificar ambas tareas, Lean hace lo mismo que cualquier otro asistente de demostración, permite utilizar un segundo lenguaje de programación para construir estas expresiones indirectamente, con la casualidad de que este segundo lenguaje no es sino el propio Lean.

En Lean, la palabra `by` permite utilizar una serie de instrucciones de alto nivel, denominadas *tácticas*, para construir una expresión del tipo que sea necesario. Dentro de un bloque de tácticas, el lenguaje utiliza una notación totalmente distinta de la de las expresiones, que permite a las tácticas definir su propia sintáxis.

```
theorem add_comm : ∀a b : ℕ, a + b = b + a := by
  intro a b
  induction b with
  | zero      => rw [Nat.add_zero, Nat.zero_add]
  | succ b' ih => rw [Nat.succ_add, Nat.add_succ, ih]
```

Esta demostración de `add_comm` expresa de más intuitivamente como pensaría un matemático para probar el resultado. Dados `a` y `b` números naturales, se aplica inducción sobre `b`. El caso cero se resuelve substituyendo las igualdades `Nat.add_zero` y `Nat.zero_add`, mientras que el paso de inducción se resuelve substituyendo `Nat.succ_add`, `Nat.add_succ` y la hipótesis de inducción.

Estas tácticas no son más que funciones de Lean con un tipo concreto, que permite modelar acciones sobre el estado de una demostración incompleta, aunque no veremos como se escriben este tipo de funciones. Al ser programas arbitrarios, las tácticas permiten implementar diversas técnicas de demostración automática directamente en Lean.

```
example (k m n o : ℕ) : ((k + m) + n) + o = ((o + n) + m) + k := by ring
```

Por ejemplo, la táctica `ring`, permite resolver igualdades en tipos dotados de una estructura de (semi)anillo conmutativo, reduciendo cada lado a una forma normal, recordando las transformaciones realizadas en cada lado y aplicándolas en el orden correcto para construir la demostración.

En Lean es fácil utilizar tácticas y expresiones conjuntamente. El prefijo `by` puede usarse en lugar de cualquier expresión para construirla insitu usando tácticas, y, análogamente, dentro de un bloque de tácticas, podemos utilizar la táctica `exact` para resolver una meta con una expresión explícita, o la táctica `have` para introducir una nueva hipótesis explícitamente.

Si bien hay muchos resultados sencillos que pueden demostrarse con una sola táctica, en la práctica las demostraciones se escriben combinando múltiples tácticas, que describen los *pasos* que realizaríamos informalmente para escribir la demostración. Aun así, estos pasos suelen tener que ser mucho más detallados que los que se escribirían habitualmente en una demostración.

Cada bloque de tácticas opera sobre un contexto que describe las hipótesis disponibles y las metas pendientes. Desde el punto de vista del usuario, cada táctica puede modificar crear y eliminar, tanto hipótesis como metas. Por este motivo, los editores de texto integrados con el lenguaje permiten inspeccionar en cada posición de una demostración por tácticas el contexto de la demostración.

Por otro lado, las tácticas tienen la responsabilidad de ser capaces de generar expresiones del tipo correcto para cerrar la demostración. El tipo de la expresión generada es comprobado de manera independiente por el lenguaje, de manera que aunque una táctica produzca una expresión incorrecta, usar tácticas no puede comprometer la consistencia de las demostraciones.

Por otro lado, muchas tácticas son extensibles. Por ejemplo, el simplificador, `simp`, reduce metas o hipótesis a una forma normal utilizando todos los resultados anotados con el atributo `@[simp]`. En la práctica, esto lleva a escribir multitud de *lemas de simplificación* que describen cual debería ser la forma normal del simplificador para cada definición, si bien la demostración de estos lemas suele ser trivial.

3.2 Clases de Tipos

En el capítulo anterior, vimos que el CIC admite polimorfismo a nivel de universos. Por la correspondencia de Curry-Howard el polimorfismo se puede interpretar como la capacidad de un lenguaje para describir teoremas genéricos, aplicables a más de un tipo.

Sin embargo, este polimorfismo no es selectivo sobre sus tipos. Las definiciones que permite hacer son siempre genéricas sobre todos los tipos de uno o varios universos. Si estuviéramos formalizando un teorema sobre anillos o grupos, nos gustaría que fuera aplicable solo a tipos con una estructura algebraica compatible.

La respuesta que se da a este problema en lenguajes de programación funcional, introducida por primera vez en Haskell, utiliza un paradigma conocido como *clases de tipos* [3, §4]. Este mecanismo no se define como parte del sistema de tipos, sino del proceso de elaboración.

Las clases de tipos describen lo que en la lógica de primer orden sería una teoría, con un conjunto de constantes u operaciones, y una serie de axiomas. Cuando un resultado tiene como parámetro una expresión de una clase de tipos, puede utilizar las constantes y axiomas de la teoría, de manera que es aplicable a todos los modelos de la teoría.

Por otro lado, las instancias de una clase, que en Lean no son más que expresiones cuyo tipo coincide con la clase, describen modelos concretos de la teoría.

Por estos motivos, las clases de tipos se utilizan en Lean para formalizar, por ejemplo, las estructuras algebraicas y sus resultados.

Dado que los modelos suelen describir operaciones sobre otros tipos, existe un mecanismo que permite inferir implícitamente qué modelo utilizar cuando un resultado lo necesita.

Para distinguir los parámetros que deben ser inferidos de esta forma, Lean utiliza corchetes alrededor de la descripción de un parámetro, asumiendo que su tipo es una clase.

```
class Add (α : Type _) where
  add (a b : α) : α
#check (Add.add : {α : Type u} → [self : Add α] → α → α → α)
notation a " + " b => Add.add a b

instance AddNat : Add ℕ := { add := Nat.add }
#eval 3 + 2  -- `Add.add 3 2` usa `AddNat` como instancia
```

El comando `class` declara un tipo inductivo con un único constructor, donde las constantes y axiomas de la clase son parámetros que necesitan proporcionar los modelos. Por otro lado, el comando `instance` define una instancia de una clase, y la anota en una tabla usada por el sistema de inferencia para encontrar instancias.

La verdadera potencia de este tipo de polimorfismo es la capacidad de declarar instancias que dependen de otras instancias, inferidas recursivamente. Esto permite, por ejemplo, extender una clase con otra que añade más axiomas,

registrando automáticamente una instancia que permite tratar la clase extendida como la más sencilla.

```
class AddSemigroup (G : Type u) extends Add G where
  add_assoc : ∀a b c : G, a + b + c = a + (b + c)
```

Este tipo de polimorfismo asume que, en general, solo existe una instancia de una clase para cada tipo en el que tenga sentido, y en caso contrario puede dar lugar a situaciones muy confusas.

A pesar de estas posibles complicaciones, el polimorfismo basado en clases de tipos es utilizado extensivamente en Lean. Por ejemplo, todos los operadores comunes se definen utilizando clases de tipos, lo que permite que el significado de un operador dependa del tipo de los operandos, como hemos mostrado con `Add`.

3.3 Axiomas y Computación

Lean extiende el CIC con axiomas adicionales, que permiten formalizar razonamientos clásicos, más allá de la lógica intuicionista. En esta sección describiremos como hace el capítulo 12 de *Theorem Proving in Lean 4* [1] algunos de estos axiomas y cómo afectan al modelo de computación de Lean 4.

Las limitaciones sobre las reglas de eliminación de los tipos inductivos proposicionales en Lean, que hemos mencionado en el capítulo anterior, están motivadas por un axioma fundamental de Lean, la extensionalidad proposicional.

```
inductive Iff : (a : Prop) → (b : Prop) → Prop where
  | intro : (mp : a → b) → (mpr : b → a) → Iff a b
notation a " ↔ " b => Iff a b

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

En Lean, el comando `axiom` afirma la existencia de una constante de un tipo, sin asignarle un valor. El axioma `propext` afirma que si podemos probar una proposición a partir de otra y viceversa, ambas son iguales, y, por tanto, gracias a la regla de eliminación de la igualdad, podemos sustituir una por la otra en cualquier contexto.

Este axioma describe un modelo de computación del CIC, en el que las demostraciones concretas de los resultados son irrelevantes. Al ser equivalentes, durante la evaluación, estas pueden ser ignoradas, solo es relevante saber que existe una.

Esto permite usar las proposiciones para describir el comportamiento de otras definiciones con una interpretación computacional, de manera que la comprobación de tipos sea capaz verificar que estas propiedades se cumplen, pero, tras la compilación, el lenguaje olvida por completo todas las proposiciones.

Este es el motivo por el que las reglas de eliminación de tipos inductivos proposicionales solo pueden ser utilizadas para construir otras proposiciones, que en tiempo de evaluación tampoco existirían.

Bajo esta interpretación, es posible definir axiomas clásicos de la lógica proposicional, como la exclusión del tercero, ya que, a pesar de que afirman la existencia de una expresión sin darla, esta expresión es una proposición, que nunca es necesario utilizar en tiempo de evaluación.

```
axiom Classical.em (p : Prop) : p ∨ ¬p

theorem not_not_elim (p : Prop) : ¬¬p → p := by
  intro h
  apply Or.elim (Classical.em p) <;> (intro; trivial)
```

En esta demostración, el operador `<;>` utiliza `(intro; trivial)` para resolver los dos casos generados por `apply`. El razonamiento por casos que aparece en esta demostración es tan común que existe una táctica para dividir una meta proposicional en casos, **`by_cases`**.

La extensión del sistema de tipos de Lean con axiomas, permite extender la correspondencia de Curry-Howard a la lógica clásica, sin perder la interpretación computacional en casos como este.

Otro axioma clásico utilizado en Lean es la extensionalidad funcional, que permite intercambiar dos funciones a partir de una demostración de que coinciden en todo su dominio.

```
axiom funext {α : Sort u} {β : α → Sort v} {f g : (x : α) → β x}
    (h : ∀ x, f x = g x) : f = g
```

Sin embargo, existe un axioma clásico que no admite interpretación computacional, el axioma de elección. En Lean, este axioma se define a partir de la clase `Nonempty`, que afirma la existencia de un elemento de un tipo.

```
class Nonempty : (α : Type u) → Prop where
  | intro (val : α) : Nonempty α

axiom Classical.choice : Nonempty α → α
```

Dado que el parámetro `val` no es una proposición y tampoco es un índice, la regla de eliminación de `Nonempty` solo permite construir proposiciones. Mientras queramos probar proposiciones, no importa cual fuera el elemento de `α` que hubiéramos utilizado, ya que tampoco importará qué demostración concreta estamos escribiendo. Sin embargo, el compilador de Lean es incapaz de transformar una expresión que usa 'Classical.choice' en un programa evaluable, dado que en tiempo de evaluación no puede utilizar la demostración que se usó originalmente para probar `Nonempty α`.

Por este motivo, en Lean, las definiciones que dependen de axiomas no computables como `Classical.choice` deben tener el prefijo **`noncomputable`**, o aparecer en una sección no computable, delimitada con el comando **`section`**.

3.4 Recursión

Dado que en Lean las funciones pueden tener una interpretación lógica, es crucial tener la garantía de que toda definición recursiva termina, es decir, de que

las demostraciones son realmente finitas. Si utilizáramos `sorry` para pedir al lenguaje que admitiera la terminación de una función cuando no debería, es sencillo dar una demostración infinita de `False`.

```
def unsound (x : ℕ) : False :=
  unsound (x + 1)
decreasing_by sorry

#check (unsound 0 : False)
```

En Lean, el compilador es responsable de traducir todas las definiciones recursivas en aplicaciones de las funciones recursoras de tipos inductivos. Sin embargo, como veremos, no siempre es capaz de hacer esta traducción, algo que nos afectará en nuestra formalización del algoritmo de división.

3.4.1 Recursión estructural

El caso más sencillo de recursión es la recursión estructural, en la que los parámetros pasados recursivamente a la iteración siguiente han sido extraídos como parámetros de un constructor inductivo de los parámetros iniciales.

```
def add : ℕ → ℕ → ℕ
| a, Nat.zero   => a
| a, Nat.succ b => Nat.succ (add a b)

noncomputable def add' (a b : ℕ) : ℕ := Nat.recOn b
  (a)
  (λb prev => Nat.succ prev)
```

La razón por la que la segunda definición, no es computable es porque, en realidad, en el caso de los números naturales, el compilador utiliza una función recursora más eficiente, `Nat.brecOn`, análoga al principio de inducción fuerte, y no es capaz de transformar expresiones que utilizan `Nat.recOn` directamente en un programa ejecutable, pero `recOn` ilustra mejor la sencillez de la traducción.

Gracias al axioma de extensionalidad funcional, podemos probar por inducción sobre `b` que ambas definiciones son iguales, y, si anotamos este resultado con el atributo `csimp`, el compilador será capaz de evaluar `add'` usando `add`.

```
@[csimp]
theorem add'_eq_add : add' = add := by
  ext a b
  unfold add' add
  induction b with
  | zero       => simp
  | succ b' ih => unfold add; simp [ih]

#eval add' 1 1 -- 2
```

Esto permite definir en el lenguaje alternativas computables a definiciones que no lo son, siempre y cuando se pueda demostrar su equivalencia. Si utilizáramos `#print add'_eq_add`, observaríamos que esta demostración no es más que otra (compleja) aplicación de `Nat.rec`, pues por la correspondencia de

Curry-Howard, una demostración por inducción se corresponde con una función recursiva de tipo proposicional.

3.4.2 Recursión bien fundada

Cuando una definición recursiva no es estructural, el compilador permite probar la terminación de la definición a partir de una demostración de que los parámetros pasados a la llamada recursiva decrecen con respecto a alguna relación de orden bien fundada.

Se dice que un elemento es accesible respecto a una relación si todos los elementos por debajo de él, interpretando la relación como una relación de orden, son accesibles. Esta definición hace la regla de eliminación de `Acc` análoga a un axioma de inducción fuerte.

```
inductive Acc {α : Sort u} (r : α → α → Prop) : α → Prop where
  | intro (x : α) (h : (y : α) → r y x → Acc r y) : Acc r x
```

Con esta definición, se dice que una relación está bien fundada si todos los elementos del tipo son accesibles respecto a ella.

```
inductive WellFounded {α : Sort u} (r : α → α → Prop) : Prop where
  | intro (h : ∀ a, Acc r a) : WellFounded r

class WellFoundedRelation (α : Sort u) where
  rel : α → α → Prop
  wf  : WellFounded rel
```

Cuando el compilador encuentra una definición recursiva que no usa recursión estructural, utiliza inferencia de clases para encontrar una relación bien fundada sobre el tipo de alguno de los parámetros, y, para cada parámetro, intenta probar que cada aplicación recursiva decrece con respecto de esta relación, invocando en el contexto de cada llamada recursiva la táctica `decreasing_tactic` con dicha meta.

A partir de esta demostración, el compilador utiliza la regla de eliminación de `Acc`, traduciendo finalmente la recursión en la aplicación de una función recursora.

Sin embargo, no es difícil encontrar funciones para las que no existe ninguna relación bien fundada que decrezca directamente sobre los parámetros. En estos casos, es posible proporcionar como criterio de descenso una función arbitraria sobre los parámetros que sí decrezca con respecto a alguna relación bien fundada.

```
-- En esta definición, `l` no cambia, y `start` crece en cada recursión
def sumFrom (l : List ℕ) (start : ℕ) : ℕ :=
  if h : start < l.length then l[start] + sumFrom l (start + 1) else 0
-- El algoritmo termina porque `l.length - start` decrece
-- con respecto al orden (estricto) usual sobre `ℕ`
termination_by sumFrom l start => l.length - start
```

En otros casos, es posible que la táctica `decreasing_tactic` sea incapaz de probar el decrecimiento del criterio para una llamada recursiva. Por ejemplo,

en la definición del algoritmo de división para los números naturales, en la que el criterio es el decrecimiento del dividendo con respecto al orden natural de $\mathbb{N}$, es necesario especificar una táctica alternativa para probar que `x - y < x` cuando `0 < y`.

```
theorem div_rec_lemma {x y : Nat} : 0 < y ∧ y ≤ x → x - y < x :=
  λ⟨ypos, ylex⟩ ↦ Nat.sub_lt (Nat.lt_of_lt_of_le ypos ylex) ypos

def div (x y : Nat) : Nat :=
  if 0 < y ∧ y ≤ x then (div (x - y) y) + 1 else 0
decreasing_by apply div_rec_lemma; assumption
```

Otra alternativa es definir una hipótesis auxiliar en el contexto de la llamada recursiva, ya que `decreasing_tactic` siempre intenta cerrar la meta usando `assumption`, una táctica que intentar cerrar la meta con cada hipótesis disponible.

```
def div (x y : Nat) : Nat :=
  if h : 0 < y ∧ y ≤ x then
    -- Esta hipótesis es una pista para `decreasing_tactic`
    have: x - y < x := by apply div_rec_lemma; assumption
    (div (x - y) y) + 1
  else 0
```

3.5 Mathlib

Mathlib es un proyecto iniciado en 2017 con el objetivo de crear una biblioteca unificada de resultados matemáticos formalizados en Lean 4 [4]. Es un esfuerzo comunitario y en constante crecimiento, que actualmente ha conseguido cubrir una gran parte del currículo de grado francés, cubriendo en distinta medida las principales ramas de las matemáticas [8].

En esta sección presentaremos como se traducen en Mathlib algunas de las nociones matemáticas que han sido relevantes para el desarrollo de este trabajo.

3.5.1 Estructuras algebraicas

La formalización del álgebra en Mathlib utiliza clases de tipos para representar estructuras algebraicas, con el objetivo de facilitar la declaración de resultados genéricos sobre dichas estructuras. Esta estrategia también permite modelar y beneficiarse de las relaciones entre estructuras algebraicas.

```
universe u
variable (T : Type u)

class Mul T where
  mul : T → T → T
notation a " * " b => Mul.mul a b

class One T where
  one : T
```

```
class Semigroup T extends Mul T where
  mul_assoc : ∀ a b c : T, a * b * c = a * (b * c)

class CommSemigroup T extends Semigroup T where
  mul_comm : ∀ a b : T, a * b = b * a

class MulOneClass T extends Mul T, One T where
  one_mul : ∀ a : T, 1 * a = a
  mul_one : ∀ a : T, a * 1 = a

class Monoid T extends Semigroup T, MulOneClass T
```

Para expresar un resultado aplicable a una estructura algebraica, por ejemplo, un semigrupo conmutativo, solo es necesario declarar como parámetro implícito una instancia de la clase `CommSemigroup G`, lo que permite acceder a todas las propiedades de la clase y las clases que extiende.

```
variable {S : Type u} [CommSemigroup S]
theorem mul_comm : ∀ a b : S, a * b = b * a :=
  CommSemigroup.mul_comm

example : ∀ a b c : S, a * b * c = c * b * a := by
  intro a b c
  rw [Semigroup.mul_assoc, CommSemigroup.mul_comm b c,
      CommSemigroup.mul_comm a (c * b)]
```

El comando **`variable`** facilita la escritura de múltiples resultados relativos a una misma estructura algebraica, declarando el parámetro una única vez.

La herencia entre clases permite utilizar una instancia de una subclase como alguna de sus superclases.

```
instance natCommSemigroup : CommSemigroup ℕ where
  mul := Nat.mul
  mul_assoc := Nat.mul_assoc
  mul_comm := Nat.mul_comm

#check (inferInstance : Semigroup ℕ) -- natCommSemigroup.toSemigroup
```

Sin embargo, esta idea no es tan útil si queremos tener más de una única estructura algebraica en un mismo tipo. Por ejemplo, los números naturales tienen estructura de semigrupo tanto para la suma como para el producto.

```
instance natSemigroup : Semigroup ℕ where
  mul := Nat.mul
  mul_assoc := Nat.mul_assoc
instance natAddSemigroup : Semigroup ℕ where
  mul := Nat.add
  mul_assoc := Nat.add_assoc
#eval 2 * 4 -- 6
#eval @Mul.mul ℕ natSemigroup.toMul 2 4 -- 8
```

En este ejemplo, al disponer de más de una única instancia de la clase `Semigroup ℕ`, la única forma de utilizar la primera es de forma explícita, anulando el propósito inicial del polimorfismo basado en clases de tipos.

Para resolver este problema, en Mathlib existen clases distintas para estructuras algebraicas similares, por ejemplo, por cada estructura algebraica por debajo de los grupos multiplicativos, existe una análoga para los grupos aditivos, con el mismo nombre y el prefijo `Add`.

```
class Zero T where
  zero : T
class AddSemigroup T extends Add T where
  add_assoc : ∀ a b c : T, a + b + c = a + (b + c)
class AddCommSemigroup T extends AddSemigroup T where
  add_comm : ∀ a b : T, a + b = b + a
class AddZeroClass T extends Add T, Zero T where
  zero_add : ∀ a : T, 0 + a = a
  add_zero : ∀ a : T, a + 0 = a
```

De esta manera, es posible tener múltiples estructuras similares sobre el mismo tipo, accesibles por medio de notación y nombres distintos. Para aliviar el esfuerzo de definir resultados aplicables a estructuras tanto multiplicativas como aditivas, Mathlib utiliza el atributo `to_additive`, que permite traducir automáticamente un resultado sobre estructuras multiplicativas a estructuras aditivas.

```
@[to_additive]
theorem mul_comm_3 : a * b * c = c * b * a := by
  rw [mul_assoc, mul_comm b _, mul_comm a _]
-- El atributo `to_additive` genera el siguiente resultado, `add_comm_3`
example [AddCommSemigroup S] (a b c : S) : a + b + c = c + b + a := by
  rw [add_assoc, add_comm b _, add_comm a _]
```

La flexibilidad de Lean como lenguaje metaprogramable permite que esta traducción automática de resultados esté implementada en el propio lenguaje, y, de hecho, forme parte de Mathlib.

Sin embargo, cuando en general nos interesa definir dos instancias de una misma estructura algebraica sobre un tipo, la solución requiere definir un tipo equivalente.

Por motivos de eficiencia, las definiciones de Lean tienen distintos grados de reducibilidad frente al elaborador, que de lo contrario sería intratable [9]. Esto nos permite definir tipos iguales por definición, pero que se comportan de forma distinta respecto a la inferencia de clases de tipos.

```
abbrev ℕ := Nat    -- `abbrev` define un alias reducible
def ℕ' := Nat      -- `def` es *semireducible*

#check (inferInstance : Add ℕ)
#check_failure (inferInstance : Add ℕ')
```

Esta estrategia permite definir nuevas operaciones y propiedades sobre un tipo ya existente, cuando existe una biyección natural entre dos estructuras algebraicas, y es común en Mathlib, pues tiene la ventaja adicional de que permite heredar resultados del tipo original fácilmente, sin tener que volver a demostrarlos, algo que haremos en nuestra formalización de algoritmo de división.

Por ejemplo, la definición del tipo de los R-polinomios multivariados sobre un conjunto σ de variables en Mathlib es la siguiente:

```
def MvPolynomial (σ : Type u₁) (R : Type u₂) [CommSemiring R] :=
  AddMonoidAlgebra R (σ →₀ ℕ)
```

donde `AddMonoidAlgebra` describe un álgebra monoidal aditiva sobre un semianillo, dotada del producto de convolución, y está definida por:

```
def AddMonoidAlgebra (k : Type u₁) (G : Type u₂) [Semiring k] :=
  G →₀ k
```

A su vez, `G →₀ k` es notación para el tipo de las funciones no dependientes con soporte finito entre `G` y `k`, que se definen como una estructura (un tipo inductivo con un único constructor):

```
structure Finsupp (α : Type u₁) (M : Type u₂) [Zero M] where
  support : Finset α
  toFun : α → M
  mem_support_toFun : ∀a, a ∈ support ↔ toFun a ≠ 0
notation σ " →₀ " M => Finsupp σ M
```

De manera similar a `class`, el comando `structure` define un tipo inductivo con un único constructor, especificando cada parámetro en una línea. En esta definición, `Finset α` representa un conjunto finito de elementos de `α`, que esencialmente se define como una lista módulo permutaciones.

Como vemos, la definición de `MvPolynomial` tiene múltiples capas de abstracción, lo que tiene como beneficio la posibilidad de utilizar en el desarrollo de teorías sobre polinomios multivariados cualquier resultado sobre álgebras monoidales y funciones de soporte finito. El tipo `MvPolynomial σ R` es igual por definición a `(σ →₀ ℕ) →₀ R`, el tipo de las aplicaciones con soporte finito de `σ →₀ ℕ` en `R`, los tipos de los multigrados y los coeficientes respectivamente.

De esta forma, los polinomios multivariados pueden contar con una definición diferente de la suma y el producto respecto a la de las funciones con soporte finito, si bien en este caso, la definición de la suma coincide.

3.5.2 Órdenes

Otra de las nociones matemáticas que define Mathlib son los distintos tipos de órdenes, que se definen también usando clases de tipos.

La jerarquía que forman las clases de órdenes en Mathlib tiene 4 niveles:

1. Las clases `LE` y LT dan significado a la notación `≤` y `<` sobre un tipo.
2. La clase `Preorder` añade las propiedades de reflexividad y transitividad a `≤`, y define `<` a partir de `≤`.
3. La clase `PartialOrder` añade la propiedad antisimétrica a `≤`.
4. La clase `LinearOrder` añade la propiedad de que el orden sea total, y, por conveniencia, decidible.

Por otro lado, la propiedad de que un orden sea buen orden se define a partir de la definición de relaciones bien fundadas, que vimos en la sección 3.4.

Capítulo 4

División Multivariada

En este trabajo, hemos implementado el algoritmo de división multivariada en Lean 4, aprovechando las capacidades del lenguaje para demostrar formalmente la corrección y terminación del algoritmo. El algoritmo de división que vamos a formalizar es el Algoritmo 21.11 de [12].

Algoritmo 1 División con resto multivariada

Entrada: $f, f_1, ..., f_s \in \mathbb{K}[\mathbf{x}]$

Salida: $q_1, ..., q_s, r \in \mathbb{K}[\mathbf{x}]$ tales que $f = q_1 f_1 + \cdots + q_s f_s + r$, y ningún término de r es divisible por ningún término principal de $f_1, ..., f_s$

$r \leftarrow 0$
$p \leftarrow f$
$q_1, ..., q_s \leftarrow 0, ..., 0$
while $p \neq 0$ **do**
    if $\exists i \in \{1, \dots, s\} : \text{LT}(f_i) \mid \text{LT}(p)$ **then**
        $i \leftarrow \min\{i \in \{1, \dots, s\} : \text{LT}(f_i) \mid \text{LT}(p)\}$
        $q_i \leftarrow q_i + \frac{\text{LT}(p)}{\text{LT}(f_i)}$
        $p \leftarrow p - \frac{\text{LT}(p)}{\text{LT}(f_i)} f_i$
    else
        $r \leftarrow r + \text{LT}(p)$
        $p \leftarrow p - \text{LT}(p)$

Como mostraremos a lo largo de esta sección, uno de los problemas que aparecen en la práctica al escribir demostraciones completamente formales, es la necesidad de introducir multitud de resultados intermedios.

En las secciones 4.1 y 4.2 mostraremos las definiciones previas que hemos necesitado introducir. En la Sección 4.3, explicaremos el orden que hemos seguido para demostrar aquellos resultados que hemos necesitado y no forman parte de Mathlib. Finalmente, en el resto de las secciones explicaremos otros problemas que hemos encontrado durante el proceso, como la forma de abordar

el 0 como divisor, distintas formas de probar la corrección, y los cambios que han sido necesarios para probar la terminación.

Como se mencionó en la Sección 3.5, los polinomios multivariados se definen en Mathlib como un `R`-álgebra monoidal sobre el conjunto de multigrados `σ →₀ ℕ`, que a su vez puede verse como el conjunto de funciones con soporte finito entre multigrados y coeficientes, dotado del producto de convolución.

```
example: MvPolynomial σ R = (σ →₀ ℕ) →₀ R := rfl
```

Desafortunadamente, esta definición, y en particular, la de `Finsupp`, el tipo de las funciones con soporte finito, está definido en Mathlib dentro de una sección no computable, para evitar tener como hipótesis adicionales la decidibilidad de la igualdad en el tipo del dominio. Por este motivo, todas nuestras definiciones serán necesariamente no computables, con la ventaja de que podremos utilizar razonamientos clásicos. Como hemos visto en la Sección 3.4, esto no impide definir separadamente una forma de evaluar nuestras definiciones, algo que no haremos.

En nuestro caso, dado que, para definir el algoritmo de división necesitaremos trabajar sobre un cuerpo, en los resultados y definiciones previas nos hemos limitado a considerar polinomios para los que `R` es al menos un anillo conmutativo, luego en general podemos asumir las siguientes variables:

```
variable {σ: Type u₁} {R: Type u₂} [CommRing R] (p q: MvPolynomial σ R)
```

Una de las definiciones que necesitaremos para formalizar el algoritmo, es el concepto de monomio. En Mathlib, existe una definición para los monomios, pero es una función que construye polinomios con un único término, a partir de un multigrado y un coeficiente:

```
def monomial (s : σ →₀ ℕ) : R →ₗ[R] MvPolynomial σ R := lsingle s
```

En esta definición, `A →ₗ[R] B` denota una aplicación R-lineal de A en B.

Dado que en nuestro caso vamos a necesitar razonar sobre relaciones y operaciones entre monomios, ha sido necesario definir un tipo que describa los monomios.

Al igual que la definición de `MvPolynomial`, nuestra definición es un alias de otro tipo, el los multigrados que intervienen en la definición de `MvPolynomial`, es decir, funciones de soporte finito sobre el conjunto de variables con imagen en `ℕ`, que determinan el grado en cada variable.

```
def Monomial (σ : Type u₁) := (σ →₀ ℕ)
```

Análogamente se ha definido un tipo para describir los términos, como pares de coeficientes y monomios.

```
structure Term (σ : Type u₁) (R : Type u₂) [CommRing R] where
  coeff : R
  monomial : Monomial σ
```

Utilizando la definición de Mathlib para polinomios monomiales, es posible definir la forma natural de interpretar un monomio como un polinomio mónico.

```
instance monomial_coe : Coe (Monomial σ) (MvPolynomial σ R) where
  coe m := monomial m 1
```

Una consecuencia importante de esta forma de definir los monomios es que el polinomio `0` no tiene monomios, algo evidente si consideramos el conjunto de monomios de un polinomio.

```
def monomials (p : MvPolynomial σ R) : Finset (Monomial σ) :=
  p.support
```

En esta definición, la notación `p.support` se traduce en `Finsupp.support p`, ya que ni `MvPolynomial` ni `AddMonoidAlgebra` definen una constante `support`. De esta misma manera, también definimos el conjunto de términos y coeficientes, ambos conjuntos finitos, utilizando `Finset.image` y `MvPolynomial.coeff`.

```
def coeffs (p : MvPolynomial σ R) : Finset R :=
  p.support.image λm => p.coeff m
def terms (p : MvPolynomial σ R) : Finset (Term σ R) :=
  p.support.image λm => ⟨p.coeff m, m⟩
```

También podemos definir una estructura de monoide multiplicativo conmutativo sobre el tipo de los monoides, que podemos heredar de la estructura de monoide aditivo sobre las funciones con soporte finito `σ →₀ ℕ`, utilizando `Multiplicative`, una definición de Mathlib que permite traducir estructuras aditivas en multiplicativas usando el sistema de tipos, capaz de levantar estructuras como `AddCommMonoid` a `CommMonoid`.

```
instance monomialCommMonoid : CommMonoid (Monomial σ) :=
  @Multiplicative.commMonoid (Monomial σ) (@Finsupp.addCommMonoid σ ℕ _)
```

Esta instancia define la multiplicación entre monomios con propiedades asociativa y conmutativa, y con un elemento neutro, el monomio de grado 0, es decir, `1`.

Un convenio implícito en Mathlib cuando se definen operaciones sobre un tipo a partir de otro, es declarar un lema que permita reescribir fácilmente esta definición en una demostración, usando el sufijo `_def`, aún si la demostración es trivial.

```
lemma monomial_mul_def (a b : Monomial σ)
    : a * b = a.toFinsupp + b.toFinsupp := rfl
```

4.1 Órdenes monomiales

El algoritmo de división multivariada se define en términos de un orden monomial, que permite ordenar los monomios del dividendo y los divisores para elegir por cuál dividir en cada paso. Por tanto, necesitamos definir este concepto.

Como se menciona en la sección 3.5.2, los órdenes se definen en Mathlib utilizando clases de tipos. De esta forma, es posible dotar a un tipo de un orden natural, definiendo una instancia adecuada.

Un orden monomial es buen un orden sobre $\mathbb{N}^n$, i.e., sobre `σ →₀ ℕ` (asumiendo que `σ` tiene cardinal finito) invariante por traslación, es decir, tal que para todos $a, b, c \in \mathbb{N}^n$, si $a \leq b$, entonces $a + c \leq b + c$, una propiedad necesaria para que el orden respete el producto de monomios.

En nuestro caso, hemos relajado la hipótesis de finitud sobre `σ`, pues la definición puede hacerse en el caso más general.

Por otro lado, dado que el tipo de los polinomios es idéntico al de las funciones de soporte finito, definir un orden monomial como una extensión de la definición de orden total, es decir, `LinearOrder` involucra algunas dificultades cuando en las demostraciones necesitamos tratar a los polinomios como funciones de soporte finito. Por este motivo hemos preferido definir la clase sin extender directamente a `LinearOrder`, sino definiendo la hipótesis de ser un orden lineal como parámetro.

```
class MonomialOrder where
  linear_order : LinearOrder (σ →₀ ℕ)
  le_add : ∀ {a b c : σ →₀ ℕ},
      linear_order.le a b → linear_order.le (a + c) (b + c)
  well_founded_lt : IsWellFounded (σ →₀ ℕ) linear_order.lt
```

Esto permite evitar la confusión entre distintas definiciones de orden, así como introducir una notación diferente para los órdenes monomiales, utilizando los símbolos `≼` y `≺`:

```
class Prec (α : Type u) where
  prec : α → α → Prop
class PrecEq (α : Type u) where
  precEq : α → α → Prop

infix:25 " ≺ " => Prec.prec
infix:25 " ≼ " => PrecEq.precEq

instance monomialOrderPrecEq [o : MonomialOrder σ]
    : PrecEq (Monomial σ) where
  precEq a b := o.le a.toFinsupp b.toFinsupp
```

Por otro lado, cuando estemos en el contexto de un orden monomial, nos gustaría que el conjunto de los multigrados heredara su orden del orden monomial, pues de lo contrario es fácil encontrar situaciones en las que distintas relaciones de orden en un teorema no coinciden.

Para que este orden sea encontrado antes que cualquier otra definición de orden sobre `σ →₀ ℕ` en el proceso de inferencia, lo declaramos con más prioridad de la normal, restringiendo su visibilidad con el atributo `scoped`, de manera que este comportamiento solo cobre efecto dentro del espacio de nombres `MvPolynomial`.

```
@[default_instance 100]
scoped instance monomialOrderFinsuppLinearOrder [o : MonomialOrder σ]
    : LinearOrder (σ →₀ ℕ) :=
  o.linear_order
```

Debido a la naturaleza del algoritmo de inferencia de clases, también es necesario declarar atajos para las superclases, de manera que, si un resultado necesita una instancia de `PartialOrder` en el contexto de un orden monomial siempre encuentre esta.

También podemos definir a partir de un orden monomial un orden sobre los propios polinomios, que nos permitiría ordenar los divisores en el algoritmo de división, si quisiéramos definirlo sobre conjuntos no ordenados en vez de listas de polinomios.

```
instance polyOrderFromMonomialOrder [MonomialOrder σ] [LinearOrder R]
    : LinearOrder (MvPolynomial σ R) :=
  Finsupp.Lex.linearOrder
```

4.2 Multigrado

A partir de un orden monomial podemos definir el *multigrado* de un polinomio, así como su término, monomio y coeficiente principal.

Sin embargo, al formalizar estas definiciones encontramos un problema. Todas ellas se pueden reducir a la del monomio principal, que se define como el máximo respecto al orden monomial del conjunto de monomios de un polinomio. No obstante, este conjunto puede ser vacío, es decir, necesitamos decidir como definir el monomio principal y el multigrado del polinomio `0`.

Afortunadamente, Mathlib define una solución general para la definición del máximo y mínimo de un conjunto vacío, los tipos `WithBot` y `WithTop`, que extienden un tipo con un elemento ínfimo y supremo respectivamente, ambos aliases de `Option`.

```
def WithBot (α : Type u) :=
  Option α
```

Esta definición va acompañada de otras que permiten denotar el ínfimo del tipo como $\bot$, y heredan el orden del tipo original, incluyendo $\bot$ como ínfimo. `Option` a su vez es un tipo genérico que describe la posibilidad de que un valor sea ausente.

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

Esto nos permite definir el multigrado del polinomio `0` como $\bot$, eliminando cualquier posible ambigüedad o arbitrariedad en su definición.

```
-- A partir de ahora, asumimos la existencia de un órden monomial
variable [MonomialOrder σ]
def lead_monomial (p : MvPolynomial σ R) : WithBot (Monomial σ) :=
  p.monomials.max
def multi_deg (p : MvPolynomial σ R) : WithBot (σ →₀ ℕ) :=
  p.lead_monomial
```

En este caso, la función `.max` es `Finset.max`, que devuelve el máximo de un conjunto respecto a un orden, o `⊥` si el conjunto es vacío.

Por conveniencia, también se define una variante que acepta una prueba de que el polinomio es no nulo, y devuelve directamente un monomio. Esta práctica es bastante común en Mathlib, y, de hecho, para hacer esta definición podemos usar una variante análoga de `Finset.max`, que recibe una demostración de que el conjunto es no vacío, construida a partir de la hipótesis `p ≠ 0` con un lema auxiliar.

```
lemma monomials_empty_iff : Finset.Nonempty (p.monomials) ↔ p ≠ 0 := by
  sorry -- Omitido
def lead_monomial' (h₀ : p ≠ 0) : Monomial σ :=
  p.monomials.max' ((monomials_empty_iff p).mpr h₀)
lemma lead_monomial'_def (h : p ≠ 0)
  : p.lead_monomial = ↑(p.lead_monomial' h) := by sorry -- Omitido
```

El símbolo `↑` realiza explícitamente una coerción a `WithBot (Monomial σ)`, aplicando `WithBot.some`. Acompañando a estas definiciones, también se han definido lemas de simplificación, que permiten al simplificador reducir expresiones en muchas de las demostraciones que haremos.

```
@[simp]
lemma lead_monomial_zero : lead_monomial (0 : MvPolynomial σ R) = ⊥ :=
  rfl
@[simp]
lemma lead_monomial'_monomial (s : Monomial σ) (c : R)
    (hr : monomial s c ≠ 0) (hc : c ≠ 0)
    : (monomial s c).lead_monomial' hr = s := by
  simp [lead_monomial', hc]
```

Esta práctica es común en Lean, y todas las definiciones realizadas en este trabajo están acompañadas de diferentes lemas de simplificación que no enunciaremos en adelante por brevedad.

En el caso del coeficiente principal, no es necesario introducir un ínfimo externo al tipo, pues el `0` ya sirve para este papel.

```
def lead_coeff : R :=
  match p.lead_monomial with
  | some m => p.coeff m
  | none   => 0

def lead_term : WithBot (Term σ R) :=
  p.lead_monomial.map λm => ⟨p.coeff m, m⟩
```

4.3 Lemas intermedios

Para poder probar la corrección del algoritmo de división multivariada, ha sido necesario primero demostrar algunas propiedades del multigrado, que no forman parte de Mathlib. En particular, para probar que el multigrado del dividendo decrece en cada iteración, nos interesa probar que el multigrado de una resta decrece cuando sus términos principales se cancelan.

```
theorem multi_deg_sub_lt (h₀ : p ≠ 0) (ht : p.lead_term = q.lead_term)
    : (p - q).multi_deg < max p.multi_deg q.multi_deg := sorry
```

Para probar esto, primero necesitamos la desigualdad no estricta.

```
theorem multi_deg_add_le :
    (p + q).multi_deg ≤ max p.multi_deg q.multi_deg := sorry
lemma multi_deg_sub_le :
    (p - q).multi_deg ≤ max p.multi_deg q.multi_deg := sorry
```

Estas demostraciones se pueden hacer más sencillas probando primero algunas propiedades sobre el conjunto de monomios.

```
lemma monomials_add [DecidableEq (σ →₀ ℕ)] (p q : MvPolynomial σ R) :
    (p + q).monomials ⊆ p.monomials ∪ q.monomials := by
  rw [monomials, support]
  apply Finsupp.support_add

lemma monomials_neg : (-p).monomials = p.monomials := by
  rw [monomials, support, Neg.neg]
  exact Finsupp.support_mapRange_of_injective
    (ι := (σ →₀ ℕ)) (M := R) (N := R)
    (e := Neg.neg) (by simp_arith) (p : (σ →₀ ℕ) →₀ R) (by
      intro a b h
      rw [←neg_neg a, ←neg_neg b, h])
```

Esta demostración utiliza el lema `Finsupp.support_mapRange_of_injective`, que afirma que el soporte de la composición de una función con soporte finito y una función inyectiva que respeta el `0`, es el soporte de la primera función, y sirve de ejemplo de como en Lean podemos construir demostraciones usando tácticas no solo en el contexto de una declaración, sino en cualquier lugar donde se espera una expresión, utilizando la palabra `by`.

En este caso, la demostración de que la negación de polinomios es inyectiva se ha realizado utilizando varias reescrituras en el último parámetro del lema, mientras que la demostración de que la negación del `0` es `0` la ha podido realizar completamente la táctica `simp_arith`, que simplifica expresiones aritméticas.

En este ejemplo, también ha sido necesario especificar algunos parámetros de tipo implícitos por nombre (`ι`, `M` y `N`), pues no ha sido posible inferirlos automáticamente. En estas situaciones, es preferible especificar parámetros por nombre, utilizando `(· := ·)`, antes que utilizar la notación `@`, para pasar explícitamente todos los argumentos, ya que, usando `@`, la demostración es sensible al orden de los parámetros implícitos del lema, que podría cambiar con relativa facilidad.

Sin embargo, cuando es necesario especificar un parámetro implícito de clase no hay otra opción, ya que, desafortunadamente, no existe una forma de pasar un argumento de clase *por tipo*, y sus nombres autogenerados suelen ser inaccesibles. Esto ocurre en la demostración de `multi_deg_add_le`, que fue omitida anteriormente.

```
theorem multi_deg_add_le :
    (p + q).multi_deg ≤ max p.multi_deg q.multi_deg := by
```

```
  unfold multi_deg lead_monomial
  simp [Finset.max_union]
  apply Finset.max_mono
  exact @monomials_add _ _ _ instDecidableEq p q
```

A partir de estas propiedades sobre el conjunto de monomios, es posible probar los lemas que necesitábamos.

```
lemma lead_monomial_neg : (-p).lead_monomial = p.lead_monomial := by
  simp [lead_monomial, monomials_neg]
lemma multi_deg_neg : (-p).multi_deg = p.multi_deg := by
  simp [multi_deg, lead_monomial_neg]
lemma multi_deg_sub_le :
    (p - q).multi_deg ≤ max p.multi_deg q.multi_deg := by
  rw [sub_eq_add_neg, ←multi_deg_neg q]
  apply multi_deg_add_le
```

Sin embargo, la demostración de `multi_deg_sub_lt`, es más larga, ya que requiere deducir la igualdad de los monomios principales y la cancelación del coeficiente principal, así como un lema auxiliar, `coeff_zero_ne_lead`, que establece que un término no puede ser principal si su coeficiente es `0`. Por este motivo, presentamos solo un esquema de la demostración omitiendo algunos pasos con `sorry`.

```
lemma multi_deg_sub_lt (h₀ : p ≠ 0) (ht : p.lead_term = q.lead_term) :
    (p - q).multi_deg < max p.multi_deg q.multi_deg := by
  apply lt_of_le_of_ne
  . apply multi_deg_sub_le
  . have h₀q : q ≠ 0 := by sorry
    have ht' : lead_term' p h₀ = lead_term' q h₀q := by sorry
    have hm : lead_monomial' p h₀ = lead_monomial' q h₀q := by sorry
    have hc : lead_coeff p = lead_coeff q := by sorry
    let m := lead_monomial' p h₀
    have h : (p - q).coeff m = 0 := by sorry
    have h': lead_monomial (p-q) ≠ m := coeff_zero_ne_lead (p - q) m h
    simp only [multi_deg]
    rw [lead_monomial'_def p h₀, lead_monomial'_def q h₀q, hm]
    simpa [←hm] using h'
```

A partir de esta demostración, probamos también un lema similar, que necesitaremos aplicar en el caso de la división en el que el monomio principal del dividendo ya no es divisible por ningún divisor, y es necesario añadirlo al resto:

```
lemma multi_deg_sub_lead_monomial (h₀ : p ≠ 0)
    : (p - monomial (p.lead_monomial' h₀) p.lead_coeff).multi_deg
      < p.multi_deg := sorry -- Omitido
```

El lema auxiliar `coeff_zero_ne_lead` afirma que, si un coeficiente es cero, no puede ser el del monomio principal, y se prueba por reducción al absurdo.

```
lemma coeff_zero_ne_lead (p : MvPolynomial σ R) (m : Monomial σ) :
    p.coeff m = 0 → p.lead_monomial ≠ m := by
  intro hc
  intro hm -- Suponemos `p.lead_monomial = m`
  have h := Finset.mem_of_max hm
```

```
  rw [monomials, support, Finsupp.mem_support_toFun] at h
  -- `hc: coeff m p = 0` y `h: p m ≠ 0` se contradicen
  contradiction
```

Un segundo lema importante para probar la corrección del algoritmo de división es el que nos permite afirmar que el multigrado del producto es la suma de multigrados, para el que necesitamos que nuestro anillo sea un dominio de integridad.

En Mathlib, la condición de ser dominio de integridad se define como la combinación de las clases `CommRing R` e `IsDomain R`. Aunque podríamos utilizar la clase `Field R`, ya que el algoritmo de división eventualmente requerirá que `R` sea un cuerpo, hemos preferido formular estos lemas en su máxima generalidad posible.

La demostración de esta propiedad es compleja de formalizar, pues la naturaleza del producto por convolución de los polinomios puede resultar difícil de abordar, y requiere llevar de un lado para otro las hipótesis de no nulidad de los polinomios, ya que la propiedad solo es cierta si ninguno de los polinomios es `0`. Esta propiedad se podría extender al polinomio `0`, si definiéramos la suma de un multigrado con `⊥`, el grado de `0`, como `⊥`, algo que no hemos hecho.

El resultado que queremos demostrar es `lead_term_mul`, que depende de `lead_monomial_mul` y `lead_coeff_mul`.

```
theorem lead_monomial_mul (h₀p : p ≠ 0) (h₀q : q ≠ 0) (h₀pq : p * q ≠ 0) :
  (p * q).lead_monomial' h₀pq =
    p.lead_monomial' h₀p * q.lead_monomial' h₀q := sorry
theorem lead_coeff_mul :
  (p * q).lead_coeff = p.lead_coeff * q.lead_coeff := sorry
theorem lead_term_mul (h₀p : p ≠ 0) (h₀q : q ≠ 0) (h₀pq : p * q ≠ 0) :
    (p * q).lead_term' h₀pq = ⟨p.lead_coeff * q.lead_coeff,
      p.lead_monomial' h₀p * q.lead_monomial' h₀q⟩ := by
  rw [lead_term'_def]
  congr
  . apply lead_coeff_mul
  . apply lead_monomial_mul
```

Para probar `lead_monomial_mul`, probamos primero una versión más débil de `lead_coeff_mul`, que afirma que el coeficiente del producto asociado al producto de los monomios principales de los factores es el producto de los coeficientes principales de los factores. Este resultado es cierto aún si no asumimos que `R` es un dominio de integridad, solo que en este caso, el producto de coeficientes podría ser `0`.

En esta demostración, utilizamos un lema existente de Mathlib, que traduce el coeficiente de un producto en una suma de productos sobre la *antidiagonal* asociada al multigrado en el que evaluamos, una propiedad del producto por convolución.

```
theorem MvPolynomial.coeff_mul (p q : MvPolynomial σ R) (n : σ →₀ ℕ) :
    coeff n (p * q) =
      ∑x in Finsupp.antidiagonal n, coeff x.1 p * coeff x.2 q :=
  AddMonoidAlgebra.mul_apply_antidiagonal p q _ _ mem_antidiagonal
def Finsupp.antidiagonal (f : α →₀ ℕ) : Finset ((α →₀ ℕ) × (α →₀ ℕ)) :=
  sorry -- Irrelevante
```

La definición de `Finsupp.antidiagonal` es complicada, pues utiliza definiciones de otros tipos, que esencialmente iteran por las combinaciones alternas de elementos que producen la misma suma, por lo que es más sencillo trabajar con el resultado que afirma precisamente esta propiedad:

```
theorem Finsupp.mem_antidiagonal {f : α →₀ ℕ} {p : (α →₀ ℕ) × (α →₀ ℕ)} :
    p ∈ Finsupp.antidiagonal f ↔ p.1 + p.2 = f := by
  sorry -- Irrelevante
```

Utilizando este resultado, nuestra meta se reduce a probar que el único elemento de la antidiagonal que no se anula es precisamente el asociado al producto de los coeficientes principales. Sin embargo, formalizar esta idea aplicando las propiedades de `Finset.sum` es laborioso, pues requiere realizar múltiples reescrituras localizadas de un término dentro de la suma, con la táctica `conv`.

Por brevedad, omitimos algunas secciones.

```
lemma coeff_lead_monomial_mul (h₀p : p ≠ 0) (h₀q : q ≠ 0) :
    (p * q).coeff (p.lead_monomial' h₀p * q.lead_monomial' h₀q) =
      p.lead_coeff * q.lead_coeff := by
  rw [coeff_mul, monomial_mul_def]
  let mp := (lead_monomial' p h₀p).toFinsupp
  let mq := (lead_monomial' q h₀q).toFinsupp

  unfold Monomial at *
  let m := mp + mq
  have h : ∀ x, x ∈ Finsupp.antidiagonal m → x.fst + x.snd = m :=
    λ_ => Finsupp.mem_antidiagonal.mp
  rw [←Finset.sum_subtype_of_mem _ h]

  conv_lhs =>
    arg 2
    intro x
    tactic =>
      have h_i : p.coeff x.val.fst * q.coeff x.val.snd =
          if x = ⟨(mp, mq), rfl⟩ then p.coeff mp * q.coeff mq else 0 := by
        sorry -- Omitido por brevedad
      rw [h_i]
  sorry -- Omitido por brevedad
```

La demostración del lema intermedio `h_i` se hace por casos, utilizando un lema auxiliar que nos dice que el coeficiente de monomios mayores que el término principal es `0`.

```
lemma coeff_gt_lead_monomial_eq_zero (h₀ : p ≠ 0)
    (h : p.lead_monomial' h₀ < m) : p.coeff m = 0 := by
  sorry -- Omitido
```

Finalmente, la demostración de `lead_monomial_mul` utiliza este lema más débil, `coeff_lead_monomial_mul`, en un cuerpo, donde el coeficiente asociado al producto de monomios principales no se anula, y coincide con el máximo monomio de `p * q`. Sin embargo, para probar que este máximo coincide, debemos probar el siguiente lema,

```
lemma monomials_mul [i : DecidableEq σ] (p q : MvPolynomial σ R) :
    monomials (p * q) ⊆ Finset.biUnion (monomials p)
```

```
    λa => Finset.biUnion (monomials q) λb => {a * b} := by
  conv in {_ * _} => rw [monomial_mul_def]
  unfold monomials Monomial Monomial.toFinsupp
  convert support_mul p q
```

donde `Finset.biUnion` es la unión de las imágenes de una función aplicada a un conjunto finito, y, en este caso, denota que el conjunto de monomios de `p * q` es un subconjunto del conjunto de productos dos a dos de monomios de `p` y `q`. Esta demostración se hereda de la análoga ya demostrada en `AddMonoidAlgebra`, `support_mul`.

Aún así, la demostración de `lead_monomial_mul` es demasiado extensa como para incluirla aquí, pues demostrar que el máximo de todos los productos dos a dos de monomios es precisamente el producto de los monomios principales, utilizando las propiedades de `Finset.biUnion`, requiere múltiples pasos y la aplicación de otro lema auxiliar para demostrar que el producto de monomios es monótono.

A partir de `lead_monomial_mul`, sí que es sencillo probar el lema equivalente para los coeficientes principales, que, por como es la definición de `lead_coeff`, es necesariamente una demostración por casos.

```
theorem lead_coeff_mul :
    (p * q).lead_coeff = p.lead_coeff * q.lead_coeff := by
  by_cases h₀p : p = 0
  . have h₀pq : p * q = 0 := by simp [h₀p]
    have h_cp₀ : p.lead_coeff = 0 := by simp [h₀p]
    have h_cpq₀ : (p * q).lead_coeff = 0 := by simp [h₀pq]
    simp [h_cp₀, h_cpq₀]
  . by_cases h₀q : q = 0
    . sorry -- Igual que el caso `p = 0`, intercambiando `p` por `q`
    . have h₀pq : p * q ≠ 0 := ne_zero_of_mul_ne_zero h₀p h₀q
      rw [←lead_coeff_def (p * q) h₀pq, lead_monomial_mul h₀p h₀q h₀pq]
      apply coeff_lead_monomial_mul
```

En esta demostración, `ne_zero_of_mul_ne_zero` es un lema que prueba que, si R es dominio de integridad, el producto de dos polinomios no nulos es no nulo, razonando por reducción al absurdo sobre su coeficiente principal, y aplicando `coeff_lead_monomial_mul`,

Esto nos permite, aunque no lo necesitaremos directamente, probar el resultado análogo a `multi_deg_add_le` para el producto.

```
theorem multi_deg_add_of_mul [Field R] (h₀p : p ≠ 0) (h₀q : q ≠ 0)
    : (p * q).multi_deg = p.multi_deg + q.multi_deg := by
  have h₀pq := ne_zero_of_mul_ne_zero h₀p h₀q
  unfold multi_deg
  rw [lead_monomial'_def (p * q) h₀pq, lead_monomial'_def p h₀p,
      lead_monomial'_def q h₀q, ←@WithBot.coe_add (σ →₀ ℕ)]
  congr
  apply lead_monomial_mul h₀p h₀q
```

4.4 El algoritmo de división multivariada

Habiendo definido el multigrado y monomio principal, podemos finalmente definir el algoritmo de división multivariada sobre `MvPolynomial σ F`, donde `F` es un cuerpo.

Para definir este algoritmo es necesaria una definición más, la divisibilidad entre monomios. En Mathlib, la relación de divisibilidad utiliza polimorfismo de clases para dar significado al operador binario `|` por medio de la clase `Dvd`. En nuestro caso, la divisibilidad entre monomios se traduce en que el grado en cada variable este acotado por el correspondiente de otro monomio.

```
instance monomialDvd : Dvd (Monomial σ) where
  dvd m n := ∀i, m i ≤ n i
```

Con esta definición, podemos definir el criterio de divisibilidad que necesitamos entre polinomios.

```
def can_divide_lead (p d : MvPolynomial σ F)
    (h₀p : p ≠ 0) (h₀d : d ≠ 0) :=
  d.lead_monomial' h₀d | p.lead_monomial' h₀p
```

Sin embargo, en esta definición podemos empezar a apreciar un problema. La divisibilidad entre polinomios no está definida cuando el divisor es `0`, y, naturalmente, tampoco la división.

4.4.1 Nulidad

Cuando un algoritmo no es aplicable a todos los valores de un tipo existen esencialmente tres formas de abordar su formalización en un sistema de tipos.

1. Ignorando el problema, es decir, extendiendo el algoritmo a todo el dominio de una forma razonable.
2. Extendiendo el codominio, como hicimos con `lead_monomial`, extendiendo el tipo del resultado con un elemento minimal.
3. Restringiendo el dominio, es decir, añadiendo hipótesis adicionales.

La primera opción es la más sencilla. Por ejemplo, los algoritmos de división se definen habitualmente en Mathlib extendiendo la definición al `0` cuando el dividendo o el divisor son `0`. Esta opción tiene como ventaja la conveniencia de no necesitar trabajar constantemente con hipótesis de no nulidad de los parámetros, pero al mismo tiempo desaprovecha la capacidad expresiva del sistema de tipos, y añade un caso extra a nuestro algoritmo.

La segunda opción es tradicionalmente usada para definir el resultado de dividir por 0 en muchos lenguajes de programación, utilizando un valor no numérico como `NaN` o `∞`. En nuestro caso, definir el resultado de dividir por `0` como `∞` u otro valor no tiene mucho sentido, ni presenta ninguna ventaja como lo hacía `⊥` en el caso del monomio principal.

La última opción permite aprovechar la capacidad del lenguaje para representar proposiciones dependientes de otros valores, y nos permitirá razonar más fácilmente sobre el algoritmo en sí, al no tener que definirlo añadiendo casos carentes de significado.

No obstante, añadir parámetros adicionales es inconveniente en la práctica, sobre todo cuando estos parámetros deben ser constantemente intercambiados entre distintos algoritmos intermedios.

Dado que la condición de que un valor no sea cero es de interés relativamente común, Mathlib define una una clase que puede utilizarse para pasar implícitamente esta condición a un algoritmo.

```
class NeZero {R} [Zero R] (n : R) where
  out : n ≠ 0
```

Sin embargo, utilizar parámetros implícitos de clase conlleva todos los inconvenientes del sistema de inferencia de clases, algo que nos gustaría evitar. Otra forma alternativa de reducir el número de parámetros es utilizando un tipo para agrupar las hipótesis con los parámetros a los que se refieren.

Lean define un tipo genérico para este propósito, `Subtype`, que se corresponde con la definición clásica de un subconjunto como un predicado, si consideramos el tipo restringido como un conjunto. Por este motivo, estos tipos se denotan con la notación clásica de subconjunto:

```
structure Subtype {α : Sort u} (p : α → Prop) where
  val : α           -- Valor
  property : p val  -- Demostración de que `val` cumple `p`

-- Denotamos `Subtype (λx : α ↦ p)` por `{x : α // p}`
macro "{" $x:ident " : " $type:term " // " $p:term "}" : term =>
  `(Subtype (λ($x:ident : $type) ↦ $p))
```

Sin embargo, en nuestro caso, hemos preferido definir un alias específico para la condición de no nulidad.

```
abbrev NonZero [Zero α] := Subtype (λx : α ↦ x ≠ 0)
abbrev NonZero.ne_zero [Zero α] (s : NonZero α) : s.val ≠ 0 := s.property
```

Esto nos permite definir con facilidad *operaciones* aplicables sobre polinomios no nulos, que pueden ser usadas con la notación de propiedad (como en `p.lead_monomial`):

```
class MulPreservesNonNull [Zero α] [Mul α] where
  nz_of_mul {a b : α} : a ≠ 0 → b ≠ 0 → a * b ≠ 0

instance fieldMulPreservesNonNull : MulPreservesNonNull F where
  nz_of_mul h₀a h₀b := by simp [h₀a, h₀b]
instance fieldPolyPreservesNonNull [MonomialOrder σ] :
    MulPreservesNonNull (MvPolynomial σ F) where
  nz_of_mul {p q} h₀p h₀q := ne_zero_of_mul_ne_zero h₀p h₀q

instance mul [Zero α] [Mul α] [inst : MulPreservesNonNull α]
    : Mul (NonZero α) where
```

```
  mul a b := ⟨a.val * b.val, inst.nz_of_mul a.ne_zero b.ne_zero⟩

lemma ne_zero_mul_def [Zero α] [Mul α] [MulPreservesNonNull α]
    (a b : NonZero α) : (a * b).val = a.val * b.val := rfl
def lead_monomial (p : NonZero (MvPolynomial σ R)) [MonomialOrder σ]
    : Monomial σ :=
  (p.val).lead_monomial' p.ne_zero
```

Con estas definiciones, nuestro criterio de divisibilidad puede escribirse de manera más compacta. También aprovechamos para definir una versión que permite que el dividendo sea nulo, en cuyo caso decimos que no se puede dividir más, en contraste con la existencia o no de un cociente (el `0`).

```
variable {F : Type _} [Field F] [MonomialOrder σ]
  (p r : MvPolynomial σ F) (s d : NonZero (MvPolynomial σ F))

def can_divide_lead := d.lead_monomial ∣ s.lead_monomial
def can_divide_lead' :=
  if h : p ≠ 0 then can_divide_lead ⟨p, h⟩ d else False
```

También podemos definir el criterio que necesitamos para la corrección del algoritmo.

```
def cannot_divide :=
  ∀m ∈ p.monomials, ¬(d.lead_monomial ∣ m)
```

Con estas definiciones, en lugar de intentar construir el algoritmo entero directamente, empezamos definiendo el cociente de monomios, así el cociente parcial de polinomios, que realiza una cancelación en el dividendo.

```
def monomial_quotient (m d : Monomial σ) (h : d ∣ m) : Monomial σ :=
  m.toFinsupp - d.toFinsupp

def partial_quotient_alg (h : can_divide_lead s d)
    : NonZero (MvPolynomial σ F) :=
  ⟨monomial
    (monomial_quotient s.lead_monomial d.lead_monomial h)
    (s.lead_coeff.val / d.lead_coeff.val),
  by simp⟩
```

4.4.2 Corrección

Llegados a este punto, empezamos a necesitar probar propiedades sobre nuestros algoritmos. Por ejemplo, la propiedad fundamental del cociente parcial de polinomios que hemos definido es que permite cancelar el término principal del dividendo.

```
lemma partial_quotient_spec
    (h : can_divide_lead s d) :
    (d * partial_quotient s d h).lead_term = s.lead_term := by sorry
```

Para probar este lema, a su vez necesitamos probar la misma propiedad para el cociente de monomios.

```
lemma monomial_quotient_spec (m d : Monomial σ) (h : d ∣ m)
    : d * monomial_quotient m d h = m := by
  unfold monomial_quotient Monomial at *
  rw [monomial_mul_def, Monomial.toFinsupp, Monomial.toFinsupp]
  ext i
  rw [Finsupp.add_apply, Finsupp.tsub_apply]
  apply add_tsub_cancel_of_le (h i)
```

Sin embargo, esta forma de probar propiedades sobre nuestros algoritmos no es ideal. Por ejemplo, consideremos la función `Vector.findIdx` de la librería estándar, que encuentra el primer índice de un vector que satisface un predicado.

```
def List.findIdx (p : α → Bool) (l : List α) : Nat := go l 0 where
  go : List α → Nat → Nat -- `findIdx.go p l n = findIdx p l + n`
  | [], n => n
  | a :: l, n => bif p a then n else go l (n + 1)
def Vector.findIdx (p : α → Bool) : Option (Fin n) :=
  let i := v.toList.findIdx p
  if h : i < n then some ⟨i, h⟩ else none
```

Si queremos probar la corrección de nuestro algoritmo de división, también necesitaremos una prueba de la corrección de `findIdx`, que no existe ni en Lean ni en Mathlib:

```
lemma findIdx_apply {n : ℕ} {v : Vector α n} {p : α → Bool} {i : Fin n}
  (h : v.findIdx p = some i) : p (v.get i) := by sorry
```

Para probar este resultado, es necesario probar la propiedad principal de la función `findIdx.go`, por inducción sobre el tamaño de las listas.

```
lemma List.findIdx.go.prop_k (p : α → Bool) (k : ℕ) (n : ℕ)
    : l.length ≤ k → List.findIdx.go p l n = List.findIdx p l + n := by
  induction k generalizing l n with
  | _ => sorry -- Omitido por brevedad
```

Aún con este lema, la demostración de `findIdx_apply`, también probada por inducción, es laboriosa de formalizar, debido a la naturaleza casuística de la función `findIdx`. La demostración debe esencialmente analizar cada caso del algoritmo y probarlo por separado.

Sin embargo, esta dificultad es puramente artificial, pues en la definición de `Vector.findIdx` y `List.findIdx` casi aparecen las proposiciones que afirman su corrección, que podríamos simplemente exponer a nuestros lemas si los algoritmos devolvieran también una prueba de su corrección. Esto resulta, no obstante, en una pérdida considerable de claridad:

```
def Vector.findIdx' {n : ℕ} (v : Vector α n) (p : α → Bool)
    : Option ({i : Fin n // p (v.get i)}) :=
  let ⟨i, h₁⟩ := v.toList.findIdx' p
  if h₂ : i < n then some ⟨⟨i, h₂⟩, by
    rw [Vector.get_eq_get, List.get_map_rev p, ←List.getD_eq_get]
    simpa using h₁⟩ else none
```

Con estas ideas en mente, y dado que para probar la corrección del algoritmo de la división multivariada será necesario utilizar la corrección de distintas funciones auxiliares, como la aplicación de cada paso de la división, se ha decidido utilizar esta última estrategia para definir todos los algoritmos involucrados.

En lugar de utilizar `Subtype`, se ha definido un tipo específico para el que se han definido extensiones sintácticas con el objetivo de preservar en la medida de lo posible la claridad en la definición de algoritmos que devuelven una prueba de su corrección.

```
structure CorrectResult (α : Type _) (p : α → Prop) where
  res : α
  correctness : p res

syntax:min term " such_that " term : term
syntax:min term " such_that " term " => " term : term
syntax:min term " correct_by " tacticSeq : term

macro_rules
| `($t:term such_that $c:term) =>
  -- Por defecto llamamos `it` al parámetro
  `(CorrectResult $t λ$(mkIdent `it) ↦ $c)
| `($t:term such_that $pat:term => $c:term) =>
  `(CorrectResult $t λ$pat ↦ $c)
| `($v:term correct_by%$c $s:tacticSeq) => do
  `(CorrectResult.mk $v $(←withRef c `(by $s)))
```

Con esta notación, podemos escribir nuestro cociente de monomios como

```
def monomial_quotient (m d : Monomial σ) (h : d ∣ m) : Monomial σ
such_that d * it = m :=  -- Condición de corrección sobre `it`
  m.toFinsupp - d.toFinsupp correct_by
    sorry   -- La demostración de `monomial_quotient_spec`
```

Sin embargo, esta forma de definir `monomial_quotient` hace su uso más incómodo, pues, si bien es posible definir una coerción implícita entre `CorrectResult` y `Monomial`, depender del sistema de coerciones en Lean puede limitar las formas en las que nos podemos beneficiar de la inferencia de tipos.

Para recuperar la simplicidad de uso de nuestra función original, se ha implementado un atributo, `alg`, aplicable a funciones que usan nuestro tipo `CorrectResult`, que genera automáticamente dos definiciones auxiliares, exponiendo a otras funciones por separado nuestro algoritmo y su corrección como un lema, pudiendo registrar este último para su uso automático por el simplificador.

```
@[alg simp]
def monomial_quotient_alg (m d : Monomial σ) (h : d ∣ m) : Monomial σ
such_that d * it = m :=
  m.toFinsupp - d.toFinsupp correct_by sorry
```

A partir de esta definición, el atributo genera las definiciones:

```
def monomial_quotient (m d : Monomial σ) (h : d ∣ m) : Monomial σ :=
  (monomial_quotient_alg m d h).res
```

```
@[simp]
lemma monomial_quotient_spec (m d : Monomial σ) (h : d ∣ m)
    : d * monomial_quotient m d h = m :=
  (monomial_quotient_alg m d h).correctness
```

Para generar estas definiciones, el atributo `alg` opera con la representación interna utilizada por Lean para representar nuestras expresiones, como se describe en [10, §4], si bien en este caso las transformaciones no son tan complejas como las realizadas por el atributo `to_additive` de Mathlib, en el que se ha inspirado nuestra implementación.

Utilizando esta notación y el atributo `alg simp`, el cociente de polinomios se define como:

```
def partial_quotient_alg (h : can_divide_lead s d)
    : NonZero (MvPolynomial σ F)
such_that (d * it).lead_term = s.lead_term :=
  ⟨monomial (monomial_quotient s.lead_monomial d.lead_monomial h)
            (s.lead_coeff.val / d.lead_coeff.val),
  by simp⟩ correct_by
    ext
    all_goals simp [NonZero.lead_term]
    . simp [NonZero.lead_coeff]; rw [mul_comm]; simp_arith
    . simp [NonZero.lead_monomial]
```

En la demostración de corrección, la táctica `ext` reduce la igualdad entre términos principales a las igualdades entre coeficientes y monomios principales. Al haber registrado `monomial_quotient_spec` como lema del simplificador, este es capaz de cerrar la igualdad entre monomios principales con tan solo desdoblar el alias `NonZero.lead_monomial`.

Con estas definiciones, estamos en condición de definir una primera versión del paso que necesitamos aplicar en cada iteración del algoritmo de división, que determina un cociente y el valor restante del dividendo.

```
@[alg]
def divide_step_alg (h : can_divide_lead s d)
    : MvPolynomial σ F × NonZero (MvPolynomial σ F)
such_that (r, q) => s = r + d * q :=
  (s.val - (d * partial_quotient s d h).val, partial_quotient s d h)
correct_by simp
```

De nuevo, gracias al atributo `alg simp`, en este caso el simplificador es capaz de probar completamente la corrección de nuestro algoritmo.

Finalmente, estamos en condiciones de intentar definir el algoritmo de división. No obstante, el tipo de recursión que necesitamos en la división no es estructural, por lo que deberemos probar la terminación de nuestro algoritmo.

4.4.3 Terminación

Como discutimos en la sección 3.4, en Lean, todas las definiciones recursivas deben probar su terminación, de manera que puedan ser traducidas en la aplicación de alguna función recursora.

En nuestro caso, probar esta terminación es interesante, no solo para garantizar la consistencia lógica del sistema, sino también como cualidad deseable de nuestro algoritmo.

Para probar la terminación del algoritmo de división multivariada, el criterio que necesitamos utilizar es el descenso del multigrado del dividendo en cada iteración. Para ello, debemos mejorar nuestra definición de `divide_step`, para incluir como propiedad de corrección la caída del multigrado en cada paso.

```
def divide_step_alg (h : can_divide_lead s d)
    : MvPolynomial σ F × NonZero (MvPolynomial σ F)
such_that (r, q) => s = r + d * q ∧ r.multi_deg < s.val.multi_deg :=
  (s.val - (d * partial_quotient s d h).val, partial_quotient s d h)
correct_by
constructor
. simp
. suffices multi_deg (s.val - (d * partial_quotient s d h).val) <
      max s.val.multi_deg (d * partial_quotient s d h).val.multi_deg by
    sorry -- Omitido por brevedad
  apply multi_deg_sub_lt'
  sorry -- Omitido por brevedad
```

En esta demostración aparece finalmente el lema `multi_deg_sub_lt'`, que tanto esfuerzo supuso probar en la sección anterior. También, aunque la hemos omitido, interviene la propiedad de corrección de `quotient_alg`.

Por otro lado, para que el compilador sea capaz de probar la terminación en base al multigrado del dividendo, necesitamos definir una instancia de la clase `WellFoundedRelation` sobre `WithBot (σ →₀ ℕ)`, el tipo de los multigrados, en el contexto de un orden monomial. Esto es sencillo, porque la definición de orden monomial incluye la propiedad de que el orden sea buen orden, de manera que la relación de orden estricto está bien fundada, y solo necesitamos *levantar* esta propiedad al orden extendido por `WithBot` usando `WithBot.wellFounded_lt`.

```
instance multi_deg_well_founded [o : MonomialOrder σ]
    : WellFoundedRelation (WithBot (σ →₀ ℕ)) where
  rel := WithBot.preorder.toLT.lt
  wf := WithBot.wellFounded_lt o.well_founded_lt.wf
```

Esto nos permitiría definir una versión débil del algoritmo de división multivariada con respecto a un único divisor, se conforma con reducir solo los monomios principales del dividendo.

```
@[alg]
def divide_one_alg (p : MvPolynomial σ F) (d : NonZero (MvPolynomial σ F))
    : (MvPolynomial σ F) × (MvPolynomial σ F)
such_that (q, r) => p = d * q + r ∧ ¬can_divide_lead' r d :=
  if h₀ : p ≠ 0 then
    if h_d : can_divide_lead ⟨p, h₀⟩ d then
      let ⟨(p_s, q_s), ⟨h_s_eq, h_decreases⟩⟩ :=
        divide_step_alg ⟨p, h₀⟩ d h_d
      let ⟨⟨q, r⟩, ⟨h_r_eq, h_r_dvd⟩⟩ := divide_one_alg p_s d
      (q + q_s, r) correct_by sorry -- Omitido
    else sorry -- Omitido
  else sorry -- Omitido
termination_by divide_one_alg p d => p.multi_deg
```

Ahora, la hipótesis `h_decreases` , lado derecho de la propiedad de corrección de `divide_step_alg` tiene como tipo `multi_deg p_s < multi_deg p` , de manera que la táctica `decreasing_tactic` es capaz de aplicar h_s_2 para probar que la llamada recursiva decrece con respecto a nuestro criterio, y no necesitamos introducir ninguna hipótesis adicional ni especificar una cláusula `decreasing_by` .

Para definir el algoritmo completo de división multivariada, hemos utilizado el tipo `Vector` , que describe listas de tamaño fijo. Esto tiene la ventaja de que nos permite expresar en el sistema de tipos que el número de cocientes obtenidos por nuestro algoritmo es igual al número de divisores.

```
def Vector (α : Type u) (n : ℕ) :=
  { l : List α // l.length = n }
```

Sobre este tipo hemos necesitado definir la operación `dot` , que denota el producto escalar entre vectores, con el objetivo de escribir nuestra condición de corrección brevemente.

```
namespace Vector
variable {α: Type u} {n : ℕ} (v : Vector α n) (w : Vector α n)

def sum [Zero α] [Add α] : α :=
  v.toList.sum

def dot [Zero α] [Add α] : α :=
  v.zipWith (· * ·) w |>.sum
```

En esta definición, `zipWith` utiliza el producto que pasamos con la notación de función implícita `(· * ·)` para multiplicar dos a dos los elementos de `v` y `w` , tomando finalmente la suma del vector resultante. Para poder usar esta definición, será necesario probar también una serie de lemas sobre vectores cuya demostración omitimos, con el objetivo de poder utilizar los siguientes resultados:

```
lemma dot_set_right : v.dot (w.set i a) =
    v.dot w + -(v.get i * w.get i) + v.get i * a := by
  conv_lhs => rw [←set_get_eq v i]
  apply dot_set

lemma dot_add [Ring α] (v w₁ w₂ : Vector α n)
    : v.dot (w₁.add w₂) = v.dot w₁ + v.dot w₂ := by sorry
```

Utilizando el tipo `Vector` , también podemos definir finalmente la condición de corrección del algoritmo, a la que acompañan algunos lemas que omitimos.

```
def cannot_divide_lead_any :=
  if h : p ≠ 0 then ∀i : Fin n, ¬can_divide_lead ⟨p, h⟩ (dd.get i)
  else True
def cannot_divide_any :=
  ∀i : Fin n, cannot_divide p (dd.get i)
```

De este modo, y aplicando el lema `multi_deg_sub_lead_monomial` para dar una pista de terminación a `decreasing_tactic`, estamos finalmente en condiciones de definir el algoritmo de división multivariada:

```
@[alg]
def mv_divide_alg {n : ℕ} (p : MvPolynomial σ F)
    (d : Vector (NonZero (MvPolynomial σ F)) n)
    : Vector (MvPolynomial σ F) n × (MvPolynomial σ F)
such_that (q, r) => p = (d.map (·.val)).dot q + r
                  ∧ cannot_divide_any r d :=
  if h₀ : p ≠ 0 then
    if h_i : (∃i : Fin n, can_divide_lead ⟨p, h₀⟩ (d.get i)) then
      let i := h_i.choose
      let ⟨(p_s, q_s), ⟨h_s_eq, h_decreases⟩⟩ := divide_step_alg ⟨p, h₀⟩
        (d.get i) (h_i.choose_spec)
      let ⟨⟨q, r⟩, ⟨h_r_eq, h_r_cannot_divide⟩⟩ := mv_divide_alg p_s d
      (q.addAt i q_s, r) correct_by
        constructor
        . simp only at h_s_eq
          rw [h_s_eq, Vector.addAt, Vector.dot_set_right,
              Vector.get_map, h_r_eq]
          ring
        . exact h_r_cannot_divide
    else
      let p_lt := monomial (p.lead_monomial' h₀) p.lead_coeff
      let p' := p - p_lt
      -- Pista de terminación:
      have: p'.multi_deg < p.multi_deg := multi_deg_sub_lead_monomial p h₀
      let ⟨⟨q', r⟩, ⟨hr₁, hr₂⟩⟩ := mv_divide_alg p' d
      (q', p_lt + r) correct_by
        constructor
        . rw [←sub_eq_of_eq_add hr₁, show p' = p - p_lt by rfl]; ring
        . apply cannot_divide_any_add p_lt r d
          . apply cannot_divide_any_of_lead p d h₀
            simp [cannot_divide_lead_any, h₀, not_exists.mp h_i]
          . exact hr₂
  else (0, p) correct_by
    constructor
    . simp
    . rw [ne_eq, not_not] at h₀
      simp [cannot_divide_lead_any, h₀]
termination_by mv_divide_alg p d => p.multi_deg
```

A partir del algoritmo, también podemos probar el resultado de existencia de un cociente y un resto. La unicidad es imposible de probar, salvo para el resto en el caso en que los divisores forman una base de Gröbner [12, §21.2].

```
theorem exists_mv_quorem
    (p : MvPolynomial σ F) (dd : Vector (NonZero (MvPolynomial σ F)) n)
    : ∃ qq : Vector (MvPolynomial σ F) n, ∃ r : MvPolynomial σ F,
        p = (dd.map (·.val)).dot qq + r ∧ cannot_divide_any r dd := by
  use (mv_divide p dd).1, (mv_divide p dd).2
  exact mv_divide_spec p dd
```

Bibliografía

[1] Jeremy Avigad et al. *Theorem Proving in Lean 4*. URL: https://leanprover.github.io/theorem_proving_in_lean4/.

[2] Mario Carneiro. *The Type Theory of Lean*. Master thesis. 2019. eprint: https://github.com/digama0/lean-type-theory/releases.

[3] David Thrane Christiansen. *Functional Programming in Lean*. Microsoft Corporation. URL: https://leanprover.github.io/functional_programming_in_lean/.

[4] The mathlib Community. "The Lean Mathematical Library". In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* (2020). DOI: 10.1145/3372885.3373824.

[5] Haskell B. Curry, Robert Feys, and William Craig. *Combinatory logic*. 1958.

[6] William A. Howard. "The formulae-as-types notion of construction". In: 1969. URL: https://www.cs.cmu.edu/~crary/819-f09/Howard80.pdf.

[7] *Lean 4*. URL: https://github.com/leanprover/lean4.

[8] *Mathlib Overview - Lean Community Website*. URL: https://leanprover-community.github.io/mathlib-overview.html.

[9] Leonardo de Moura et al. "Elaboration in Dependent Type Theory". In: (May 2015). DOI: https://doi.org/10.48550/arXiv.1505.04324.

[10] Arthur Paulino et al. *Lean 4 Metaprogramming Book*. URL: https://github.com/leanprover-community/lean4-metaprogramming-book.

[11] The Coq Development Team. *The Coq Proof Assistant, version 8.7.0*. Version 8.7.0. Oct. 2017. DOI: 10.5281/zenodo.1028037. URL: https://doi.org/10.5281/zenodo.1028037.

[12] Gathen Joachim von zur and Gerhard Jürgen. *Modern Computer algebra*. Cambridge University Press, 2013.

Apéndice A: Descripción clásica de los resultados formalizados

En este apéndice recopilamos una descripción clásica de los resultados que hemos necesitado probar en Lean 4. El objetivo de este apéndice es que pueda servir para facilitar la lectura de estas definiciones escritas en Lean.

Muchos de estos resultados son observaciones que cualquier matemático consideraría triviales en otro contexto, pero necesarios para escribir nuestras demostraciones formalmente. También hay algunos que son variaciones de otros tomando hipótesis ligeramente diferentes, que por cuestiones técnicas simplifican algunas demostraciones.

Lemas sobre monomios y órdenes monomiales

En esta sección recopilamos una descripción de los resultados introducidos en el archivo `MvPolynomial/Monomial.lean`.

Lema 4.4.1 (`monomials_of_monomial`)**.** *Sea f el polinomio con un único término, de multigrado d y coeficiente c, su conjunto de monomios es $\emptyset$ si $c = 0$, y $\{d\}$ en caso contrario.*

El siguiente resultado es la desigualdad estricta de la propiedad de un órden monomial, que se ha definido a partir de un orden no estricto sobre los multigrados.

Lema 4.4.2 (`MonomialOrder.lt_add`)**.** *Sean a, b y c multigrados, y "$<$" un orden monomial, si $a < b$, entonces $a + c < b + c$.*

Lema 4.4.3 (`monomial_mul_def`)**.** *El multigrado del producto de monomios es la suma del multigrado de los factores.*

Los siguientes resultados son la traducción de la propiedad del orden monomial a los monomios.

Lema 4.4.4 (`MonomialOrder.le_mul`). *Sean a, b y c monomios, y $\preccurlyeq$ un orden monomial, si $a \preccurlyeq b$ respecto a un orden monomial, entonces $a \cdot c \preccurlyeq b \cdot c$.*

Lema 4.4.5 (`MonomialOrder.lt_mul`). *Sean a, b y c monomios, y $\preccurlyeq$ un orden monomial, si $a \prec b$ respecto a un orden monomial, entonces $a \cdot c \prec b \cdot c$.*

Lema 4.4.6 (`monomial_prec_eq_def`). *Sean a y b monomios, $a \preccurlyeq b$ si y solo si $a \leq b$ como multigrados.*

Lema 4.4.7 (`lex_monomial_order`). *Sea σ un conjunto finito sobre el que existe un orden total, entonces el orden lexicográfico sobre los multigrados asociados a σ es un orden monomial.*

Lema 4.4.8 (`lead_monomial_zero`). *El monomio principal del polinomio 0 es $\bot$.*

Lema 4.4.9 (`lead_monomial_ne_zero`). *Si p es distinto de 0, su monomio principal es distinto de $\bot$.*

Lema 4.4.10 (`monomials_empty_iff`). *El conjunto de monomios de un polinomio p es vacío si y solo si p es igual a 0.*

Lema 4.4.11 (`lead_monomial'_def`). *Las funciones* `lead_monomial p` *y* `lead_monomial' p _` *coinciden cuando p es distinto de 0.*

Lema 4.4.12 (`lead_monomial'_monomial`). *Sea f el polinomio con un único término, de multigrado d y coeficiente c, distinto de 0. Entonces el monomio principal de f tiene multigrado d.*

Lema 4.4.13 (`lead_coeff_zero`). *El coeficiente principal de 0 es 0.*

Lema 4.4.14 (`lead_coeff_ne_zero`). *Si $p \neq 0$, su coeficiente principal es distinto de 0.*

Lema 4.4.15 (`lead_coeff_monomial`). *Sea f el polinomio con un único término, de multigrado d y coeficiente c. Entonces el coeficiente principal de f es c.*

Lema 4.4.16 (`lead_term_zero`). *El término principal de 0 es $\bot$.*

Lema 4.4.17 (`lead_term_ne_zero`). *Si $p \neq 0$, su término principal es distinto de $\bot$.*

Lema 4.4.18 (`lead_term'_def`). *El término principal de un polinomio $p \neq 0$, es el término cuyo coeficiente es el coeficiente principal de p, y cuyo monomio es el monomio principal de p.*

Lema 4.4.19 (`lead_term'_coeff_def`). *El coeficiente del término principal de un polinomio $p \neq 0$, coincide con el coeficiente principal de p.*

Lema 4.4.20 (`lead_term'_monomial_def`). *El monomio del término principal de un polinomio $p \neq 0$, coincide con el monomio principal de p.*

Lema 4.4.21 (`lead_term_monomial`). *Sea $f \neq 0$ el polinomio con un único término, de multigrado d y coeficiente c. Entonces, el término principal de f tiene multigrado d y coeficiente c.*

Lema 4.4.22 (`multi_deg_bot`). *El multigrado de 0 es $\bot$.*

Lema 4.4.23 (`multi_deg_ne_bot`). *Si $p \neq 0$, su multigrado es distinto de $\bot$.*

Lema 4.4.24 (`monomials_add`). *El conjunto de monomios de la suma de dos polinomios está contenido en la unión del conjunto de monomios de ambos.*

Lema 4.4.25 (`monomials_mul`). *El conjunto de monomios del producto de dos polinomios está contenido en la unión de productos dos a dos de monomios de los factores.*

Lema 4.4.26 (`monomials_neg`). *El conjunto de monomios de $-p$ coincide con el de p.*

Lema 4.4.27 (`lead_monomial_neg`). *El monomio principal de $-p$ coincide con el de p.*

Lema 4.4.28 (`coeff_gt_lead_monomial_eq_zero`). *El coeficiente de p asociado a un monomio mayor que el monomio principal de p es 0.*

Lema 4.4.29 (`coeff_ne_zero_le_lead_monomial`). *Si el coeficiente de p asociado a un monomio m es distinto de 0, entonces m es menor o igual que el monomio principal.*

En la siguiente propiedad, la resta se hace sobre $\mathbb{N}$, donde no se puede cancelar siempre.

Lema 4.4.30 (`Finsupp.add_sub_cancel_nat`). *Si a y b son multigrados, $(a + b) - b = a$.*

Lema 4.4.31 (`coeff_lead_monomial_mul`). *El coeficiente de $p \cdot q$ asociado al producto de monomios principales de p y q es el producto de sus coeficientes principales.*

Lema 4.4.32 (`ne_zero_of_mul_ne_zero`). *Si trabajamos con polinomios sobre un cuerpo, el producto de polinomios no nulos es distinto de 0.*

Lema 4.4.33 (`monomial_mul_mono`). *El producto de monomios es monótono, es decir, $m \cdot (x \sqcup y) = (m \cdot x) \sqcup (m \cdot y)$, donde $\sqcup$ denota el supremo entre dos monomios respecto a un orden monomial (equivalente al máximo).*

Teorema 4.4.34 (`lead_monomial_mul`). *Si p y q son polinomios sobre un cuerpo, el monomio principal de $p \cdot q$ es el producto de sus monomios principales.*

Teorema 4.4.35 (`lead_coeff_mul`). *Si p y q son polinomios sobre un cuerpo, el coeficiente principal de $p \cdot q$ es el producto de sus coeficientes principales.*

Teorema 4.4.36 (`lead_term_mul`). *Si p y q son polinomios no nulos sobre un cuerpo, el término principal de $p \cdot q$ está formado por el producto de coeficientes y monomios principales de p y q.*

Lema 4.4.37 (`multi_deg_neg`). *El multigrado de $-p$ coincide con el de p.*

Lema 4.4.38 (`lead_coeff_neg`). *Si c es el coeficiente principal de p el coeficiente principal de $-p$ es $-c$.*

Lema 4.4.39 (`coeff_zero_ne_lead`). *Si el coeficiente de p asociado a un monomio es 0, este no puede ser el monomio principal de p.*

Lema 4.4.40 (`Finset.max_union`). *Sean A y B conjuntos finitos. Entonces,* $\max(A \cup B) = \max(\max(A) \cup \max(B))$.

Lema 4.4.41 (`multi_deg_eq`). *Si los términos principales de p y q coinciden, sus multigrados también.*

Teorema 4.4.42 (`multi_deg_add_le`). *El multigrado de $p + q$ es menor o igual que el máximo de los multigrados de p y q.*

Lema 4.4.43 (`multi_deg_sub_le`). *El multigrado de $p - q$ es menor o igual que el máximo de los multigrados de p y q.*

Teorema 4.4.44 (`multi_deg_sub_lt`). *Si los términos principales de p y q coinciden, el multigrado de $p - q$ es menor que el máximo de los multigrados de p y q.*

Lema 4.4.45 (`multi_deg_sub_lt_left`). *Si los términos principales de p y q coinciden, el multigrado de $p - q$ es menor que el multigrado de p.*

La siguiente es un alias de `multi_deg_sub_lt` que toma como hipótesis la no nulidad tanto de p como de q, para usar `lead_term'` en la hipótesis de igualdad entre términos principales:

Lema 4.4.46 (`multi_deg_sub_lt'`). *Si los términos principales de p y q coinciden, el multigrado de $p - q$ es menor que el máximo de los multigrados de p y q.*

Teorema 4.4.47 (`multi_deg_add_of_mul`). *Sean p y q dos polinomios distintos de 0 sobre un cuerpo, el multigrado de $p \cdot q$ es la suma de los multigrados de p y q.*

Lema 4.4.48 (`multi_deg_sub_lead_monomial`). *El resultado de restarle a un polinomio $p \neq 0$ su término principal tiene menor multigrado que p.*

Teorema 4.4.49 (`polyOrderFromMonomialOrder`). *Un orden monomial define un orden total sobre el anillo de polinomios en varias variables sobre un anillo totalmente ordenado.*

Lemas sobre listas y vectores

En esta sección recopilamos una descripción de los resultados introducidos en el archivo `Vector/Dot.lean`.

En esta sección entenderemos que un vector ded longitud n es una lista con n elementos, numerados desde 0 hasta $n-1$. Denotaremos por v_i al elemento i-ésimo de v, por $v*w$ a la concatenación de v y w, por $(a)*v$ a la concatenación de un elemento a a v, y por $v[i \leftarrow a]$ al resultado de substituir en v el elemento i-ésimo por a.

Si el tipo de los elementos de un vector es un anillo, y v y w tienen la misma longitud, denotaremos por $v + w$ a la suma, elemento a elemento de v y w, por $\sum v$ a la suma de los elementos de v, y por $v \cdot w$ al producto escalar de v y w, es decir, la suma de productos coordenada a coordenada de v y w.

Lema 4.4.50 (`zero_def`)**.** *El vector* 0 *de longitud* n *es el vector definido con* n *ceros.*

Lema 4.4.51 (`zero_ext_iff`)**.** *Un vector de longitud* n *es igual al vector* 0 *de longitud* n *si y solo si todos sus elementos son* 0*.*

Lema 4.4.52 (`zero_get`)**.** *Para todo* $i < n$*, el elemento* i*-ésimo del vector* 0 *de longitud* n *es* 0*.*

Lema 4.4.53 (`sum_cons`)**.** *Sea* v *un vector y* a *un elemento cualquiera, entonces* $\sum((a) * v) = a + \sum v$*.*

Lema 4.4.54 (`sum_append`)**.** *Sean* v *y* w *vectores,* $\sum(v * w) = \sum v + \sum w$*.*

Lema 4.4.55 (`sum_zero`)**.** *La suma de los elementos del vector* 0 *es* 0*.*

Lema 4.4.56 (`zipWith_set`)**.** *Sean* $v = (v_0, \ldots, v_{n-1})$ *y* $w = (w_0, \ldots, w_{n-1})$ *dos vectores del mismo tipo, y* f *una operación binaria sobre los tipos de* v *y* w*. Sea* Z *la operación que aplica* f *elemento a elemento sobre dos vectores, tal que* $Z(v, w) = (f(v_0, w_0), \ldots, f(v_{n-1}, w_{n-1}))$*, y sean* $i < n$*, y* a *y* b *dos elementos.*

Entonces $Z(v[i \leftarrow a], w[i \leftarrow b]) = Z(v, w)[i \leftarrow f(a, b)]$*.*

Lema 4.4.57 (`set_get_eq`)**.** *Sea* v *un vector de longitud* n*,* $i < n$ *y* a *un elemento. Entonces,* $(v[i \leftarrow a])_i = a$*.*

Lema 4.4.58 (`dot_cons`)**.** *Sean* v *y* w *dos vectores de longitud* n *con elementos de un anillo. Entonces,* $((a_1) * v) \cdot ((a_2 * w)) = a_1 \cdot a_2 + v \cdot w$*.*

Lema 4.4.59 (`dot_append`)**.** *Sean* v_1 *y* w_1 *dos vectores sobre un anillo de longitud* n*, y* v_2 *y* w_2 *de longitud* m*, todos del mismo tipo. Entonces,* $(v_1 * v_2) \cdot (w_1 * w_2) = v_1 \cdot w_1 + v_2 \cdot w_2$*.*

Lema 4.4.60 (`dot_set`)**.** *Sean* v *y* w *dos vectores de longitud* n *sobre un anillo, e* $i < n$*. Entonces* $v[i \leftarrow a_1] \cdot w[i \leftarrow a_2] = v \cdot w - v_i \cdot w_i + a_1 \cdot a_2$*.*

Lema 4.4.61 (`dot_set_right`)**.** *Sean* v *y* w *dos vectores de longitud* n *sobre un anillo, e* $i < n$*. Entonces,* $v \cdot (w[i \leftarrow a]) = v \cdot w - v_i \cdot w_i + v_i \cdot a$*.*

Lema 4.4.62 (`dot_set_left`). *Sean v y w dos vectores de longitud n sobre un anillo, e $i < n$. Entonces, $(v[i \leftarrow a]) \cdot w = v \cdot w - v_i \cdot w_i + a \cdot w_i$.*

Lema 4.4.63 (`dot_zero`). *Sea v un vector de longitud n. Entonces $v \cdot \vec{0} = \vec{0}$, donde $\vec{0}$ denota el vector 0 de longitud n.*

Lema 4.4.64 (`zero_dot`). *Sea v un vector de longitud n. Entonces $\vec{0} \cdot v = 0$, donde 0 denota el vector 0 de longitud n.*

Lema 4.4.65 (`any_nil`). *Sea p un predicado, no existe un elemento del vector vacío (de longitud 0) que satisface p.*

Lema 4.4.66 (`any_cons`). *Sea p un predicado y v un vector, al menos un elemento de $(a) * v$ satisface p si y solo si a satisface p o al menos un elemento de v satisface p.*

Lema 4.4.67 (`findIdx_nil`). *Sea p un predicado, no existe un índice i tal que el elemento i-ésimo del vector vacío satisfaga p.*

Lema 4.4.68 (`findIdx_cons`). *Sea p un predicado, v un vector y a un elemento. Entonces, el índice del primer elemento que satisface p de $(a) * v$ es: 0 si a satisface p. No existe, si a no satisface p y ningún elemento de v satisface p. $i + 1$, si a no satisface p y v_i es el primer elemento de v que satisface p.*

Lema 4.4.69 (`findIdx_none`). *Sea p un predicado y v un vector, el índice del primer elemento de v que satisface p no existe si y solo si ningún elemento de v satisface p.*

Lema 4.4.70 (`findIdx_apply`). *Sea p un predicado y v un vector. Si i es el índice del primer elemento de v que satisface p, entonces v_i satisface p.*

Lema 4.4.71 (`dot_add`). *Sean v, w y c tres vectores de longitud n sobre un anillo, entonces $v \cdot (w + c) = v \cdot w + v \cdot c$.*

Lemas sobre la división multivariada

En esta sección recopilamos una descripción de los resultados introducidos en el archivo `MvPolynomial/MvDivide.lean`.

En esta sección asumiremos que todos los polinomios son sobre un cuerpo. Cuando digamos que un polinomio es un divisor en el algoritmo de división, asumiremos que es distinto de 0.

Lema 4.4.72 (`cannot_divide_any_zero`). *Sea $\vec{d} = (d_0, \ldots, d_{n-1})$ un vector de divisores, ningún divisor de $\vec{d}$ se puede utilizar para dividir al polinomio 0 (de acuerdo a nuestro criterio, `cannot_divide_any`).*

Este lema traduce `cannot_divide_lead_any` en `cannot_divide_any` para el término principal como polinomio.

Lema 4.4.73 (`cannot_divide_any_of_lead`). *Sea $\vec{d} = (d_0, \dots, d_{n-1})$ un vector de divisores, y p un polinomio distinto de 0. Si ningún divisor de $\vec{d}$ puede utilizarse para dividir al término principal de p, ningún divisor de $\vec{d}$ puede usarse para dividir ninguno de los términos del polinomio formado únicamente por el término principal de p.*

Utilizamos este lema para probar que `p_lt + r` no se puede dividir más en la segunda rama del algoritmo.

Lema 4.4.74 (`cannot_divide_any_add`). *Sean $\vec{d} = (d_0, \dots, d_{n-1})$ un vector de divisores, y p y q dos polinomios. Si ningún divisor de $\vec{d}$ divide a ninguno de los términos de p ni de q, tampoco dividen a ninguno de los términos de $p + q$.*

Con el siguiente resultado, declaramos una instancia de `WellFoundedRelation` que permite al compilador utilizar esta relación para probar la terminación de nuestro algoritmo:

Lema 4.4.75 (`multi_deg_well_founded`). *Sea $\preccurlyeq$ un orden monomial, es posible extender este orden con un elemento ínfimo $\bot$, el multigrado del polinomio 0, de manera que el orden resultante es un buen orden, y, en particular, una relación bien fundada compatible con $\preccurlyeq$.*

Criterio de corrección de `monomial_quotient_alg` :

Lema 4.4.76 (`monomial_quotient_spec`). *Sean m y d monomios tales que d divide a m, y sea q el resultado de dividir m por d, se tiene que $m = d \cdot q$.*

Criterio de corrección de `partial_quotient_alg` :

Lema 4.4.77 (`partial_quotient_spec`). *Sean p y d polinomios con $d \neq 0$, tales que el monomio principal de d divide al monomio principal de p, y sea q el cociente parcial de dividir p entre d, es decir, el monomio añadido al cociente en un paso concreto del algoritmo de la división. Entonces, el término principal de $d \cdot q$ coincide con el de p.*

Criterio de corrección de `divide_step_alg` :

Lema 4.4.78 (`divide_step_spec`). *Sean p y d polinomios con $d \neq 0$, tales que el monomio principal de d divide al monomio principal de p, y sean q y r el cociente y el resto de aplicar un paso de la división de p entre d, entonces $p = r + d \cdot q$, y el multigrado de r es estrictamente menor que el de p.*

Terminación de `divide_one_alg` .

Lema 4.4.79 (`divide_one_alg`). *El algoritmo de división multivariada débil con un único cociente termina.*

Para comprobar que nuestros lemas van a ser suficientes, hemos definido una versión débil del algoritmo de división multivariada, en la que solo hay un único divisor, y el criterio de parada sea que el término principal del dividendo no sea divisible por el término principal del divisor. Su criterio de corrección es el siguiente:

Lema 4.4.80 (`divide_one_spec`)**.** *Sean p y d polinomios con $d \neq 0$, y sean q y r el cociente y el resto de aplicar el algoritmo de división multivariada débil a p y d. Entonces, $p = d \cdot q + r$, y el término principal de d no divide al término principal de r.*

Terminación de `mv_divide_alg` :

Teorema 4.4.81 (`mv_divide_alg`)**.** *El algoritmo de división multivariada termina.*

Criterio de corrección de `mv_divide_alg` :

Teorema 4.4.82 (`mv_divide_spec`)**.** *Sea p un polinomio y $\vec{d} = (d_0, \ldots, d_{n-1})$ un vector de divisores, y sean $\vec{q} = (q_0, \ldots, q_{n-1})$ y r, el cociente y el resto respectivamente, de aplicar el algoritmo de división multivariada a p y $\vec{d}$.*

Entonces, $p = \sum_{i=0}^{n-1} d_i \cdot q_i + r$, y ninguno de los monomios de r es divisible por ninguno de los monomios principales de $\vec{d}$.

Teorema 4.4.83 (`exists_mv_quorem`)**.** *Sea p un polinomio y $\vec{d} = (d_0, \ldots, d_{n-1})$ un vector de divisores.*

Existen $\vec{q} = (q_0, \ldots, q_{n-1})$, un vector de polinomios, y r un polinomio tales que $p = \sum_{i=0}^{n-1} d_i \cdot q_i + r$, y ninguno de los monomios de r es divisible por ninguno de los monomios principales de $\vec{d}$.