UNIVERSIDADE DA CORUÑA

FACULTADE DE INFORMÁTICA
Departamento de Computación

Bases de Gröbner: Desarrollo formal en Coq

GILBERTO PÉREZ VEGA

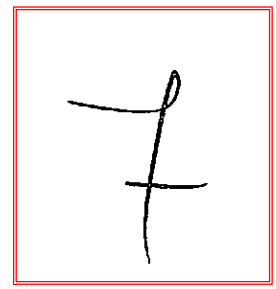
NOVIEMBRE 2004

TESIS DOCTORAL

LFCIA

UNIVERSIDADE DA CORUÑA

FACULTAD DE INFORMÁTICA
DEPARTAMENTO DE COMPUTACIÓN



Bases de Gröbner: Desarrollo formal en Coq

GILBERTO PÉREZ VEGA

TESIS DOCTORAL

UNIVERSIDADE DA CORUÑA

NOVIEMBRE 2004

D. José María Barja Pérez, Catedrático de Álgebra de la Falcultad de Informática de la Universidade da Coruña,

CERTIFICA:

Que D. Gilberto Pérez Vega, Licenciado en Ciencias Matemáticas, ha realizado en el Departamento de Computación de la Facultad de Informática de la Universidade da Coruña, bajo mi dirección, el trabajo de investigación correspondiente a su Tesis Doctoral titulado:

Bases de Gröbner: Desarrollo formal en Coq

Revisado el presente trabajo, estimo que puede ser presentado al Tribunal que ha de juzgarlo, por considerar que alcanzó los resultados con la calidad requerida. Autorizo su presentación en la Universidade da Coruña.

A Coruña, a 14 de Septiembre de 2004

Dr. José María Barja Pérez
Catedrático de Álgebra

A mis padres que siempre están;
A mis tios, Pepe y Toño, por todo;
A Mariné por su paciencia y comprensión;
A Mariña y Miguel Anxo por su ternura.

AGRADECIMIENTOS

José Luis Freire, con su siempre certera visión de las relaciones temáticas, me propuso el estudio formal de las Bases de Gröbner mediante los sistemas de ayuda a la prueba. Todo esto sucedía en Noirmoutier en las *cuartas Jornadas Francófonas de Lenguajes Aplicativos* "JFLA 94" donde él había sido invitado como primer conferenciante de habla no francesa. Después vino mi estancia en el INRIA y mi trabajo sobre el tema en la Facultad de Informática, donde él me ayudó a encontrar un poco la serenidad perdida en mis *pájaras* y, por último, me dió el *arreón* final, con la interacción de polinomios y polinomios canónicos, para llegar a la *meta*.

Quiero dar las gracias a José María Barja por su disponibilidad en todos los momentos, incluso en aquellos en los que tenía otras cosas más importantes que hacer (...), y la atención que ha puesto en mi trabajo. Mis numerosas discusiones con él siempre han encontrado comprensión y consejos. Su ayuda sobre todo en la etapa de descorazonamiento de la tesis, la redacción y mis peleas con los polinomios, ha sido irremplazable. También aprecio sus cualidades humanas como amigo, más si cabe, que como director de la tesis.

Travailler pendant trois mois au sein de l'équipe Coq a été un vrai plaisir, aussi bien du point de vue professionnel que personnel, et je remercie tous les membres de l'équipe Rocquencourt. Un merci particulier et affectueux à Benjamin Werner, il m'a donné l'idée de voir mon problème sur les structures algébriques à travers les lunettes de les setoids et il a mis les fondations de ma formalisation en Coq. Merci à Bruno Barras pour l'aide qu'il m'a donnée sur des problèmes techniques à plusiers reprises, sa disponibilité de tous les instants, et l'attention toute particulière qu'il a porté à mon travail dans mon séjour au INRIA et plus tard aussi pour courrier électronique.

Gracias a los hispano-parlantes del proyecto Coq, un gran número de talentos hoy repartidos por todo el mundo en diversos proyectos: Cristina Cornes, Cesar Muñoz, Micaela Mayero, Christian Rinderknecht, Daniel de Rauglaure, Eduardo Giménez, etc, por su acogida en mis primeros días de estancia, su compañía y discusiones en las filas del comedor y sobremesas con café, su ayuda en mi francés y en la superación de mi morriña (en particular por mi "Mariña").

Entre el sistema Linux y yo nunca ha habido una historia de amor ... a menudo mis problemas con los "makefiles de Coq" y demás instalaciones de software han sido resueltos gracias a los miembros del laboratorio LFCIA, en particular Javier Mosquera, Carlos Abalde, J.J. Quintela, Miguel Barreiro, Laura Castro y Javier París; ellos han tolerado con humor mis innumerables interrupciones con preguntas del tipo "¿cómo arreglo esto que no funciona?", "¿cómo se instala esto?" o "¡no me imprime!".

Estoy particularmente feliz de trabajar con mis compañeros del Área de Álgebra. Han sido clave para salir de mis desfallecimientos a lo largo de estos años, y les agradezco de todo corazón a Feli, Conchi, María José, Doncel y Antonio sus lecturas y correciones, así como que hayan soportado mis cambios de humor y mi despreocupación, a veces, de mis labores docentes sobre todo en la última etapa de este trabajo. También quiero destacar la labor de asesoramiento y amistad de otro miembro del Área de Álgebra, aunque no de esta universidad, Manolo Ladra.

Muchas gracias a José María Molinelli (Moli), por su amistad en situaciones difíciles y sus consejos de implementación: ¿tipos dependientes o no?, perfeccionamiento de código CAML, ...

Índice general

Capítulo 1

Introducción

> Llegará un día en que la herramienta ejecutará por sí misma y ordenadamente el trabajo que le corresponda.
>
> Aristóteles.

Este trabajo se puede encuadrar en los dominios de la computación algebraica[1] y la teoría de la prueba (deducción automática). La teoría de la prueba es el estudio de las demostraciones con la ayuda de herramientas matemáticas. Forma parte de las metamatemáticas y de la lógica, del estudio del lenguaje de las matemáticas y las leyes que permiten razonar sobre objetos matemáticos. La computación algebraica es un área reciente de la ciencia de la computación donde se desarrollan en común herramientas matemáticas y software de computación que las implementan. Los objetos básicos del álgebra computacional son los números y los polinomios. Nosotros nos interesaremos en este trabajo por los polinomios y en concreto por los polinomios de varias variables sobre un cuerpo.

1.1. Motivaciones y objetivos

La posibilidad de cometer errores en las demostraciones matemáticas es muy factible. De hecho ya en 1935 el libro de Maurice Lecat [78] recoge 135 páginas de errores cometidos, antes de 1900, por matemáticos reconocidos. Aunque en este mismo libro se reconoce con optimismo que la comunidad matemática detectó tarde o temprano estos errores, en la época de los ordenadores parece esencial que éstos deben jugar un papel importante en esta tarea. Desde hace algunos años, quizás decenas, las matemáticas forman un conjunto que sobrepasa de lejos las capacidades de un solo ser humano. La producción anual de los trabajos matemáticos representan varios miles de páginas y es necesario tener la seguridad que cada vez que se cita un teorema, está bien aplicado y se verifican

[1] A este dominio se le suele denominar también con los nombres de manipulación algebraica, cálculo formal, computación simbólica y algoritmos algebraicos.

sus hipótesis. Este trabajo parece considerable. El intento más importante de mecanización de las matemáticas es el del sistema Mizar[2] (creado por Trybulec [115] en el año 1970), en el cual este era su único objetivo. El conjunto de matemáticas formalizadas en Mizar representan 50.000 páginas que consisten tanto en lemas triviales, a la vista de un matemático, como teoremas muy complicados. Todo esto durante unos 25 años; pero como dice Trybulec, es menos que la producción de los matemáticos en el último año.

El problema de la mecanización de las matemáticas, hoy en día, es un problema de lógicos e informáticos más que un problema de las matemáticas. A pesar de todo, es necesario decir que ningún resultado de la lógica ha conseguido, hasta el momento, desanimar a los matemáticos. Incluso los resultados de incompletitud de Gödel no han permitido mostrar ejemplos de proposiciones matemáticas importantes e indemostrables.

Nuestro objetivo es estudiar los métodos formales de verificación de la corrección de algoritmos. El ejemplo escogido, el estudio de las bases de Gröbner de ideales de anillos de polinomios, nos parece adecuado para demostrar la potencia de un sistema de ayuda a la prueba. Por otra parte, los tipos de datos que se precisan, polinomios en varias variables sobre un cuerpo, no son elementales y requieren una definición explícita, incluyendo las operaciones algebraicas básicas.

Esta tesis se inscribe dentro del marco de las interacciones entre los métodos formales y la computación algebraica. Podemos citar en esta dirección, por ejemplo, los trabajos [23, 94, 107, 84, 63, 37, 52, 118, 51]. En particular, los trabajos [9, 10, 86] implementan los polinomios en los sistemas Coq y ACL2, mientras que [42, 112, 103] utilizan la teoría de las bases de Gröbner orientada hacia la prueba en Coq del Algoritmo de Buchberger, presentando resultados sobre el anillo de polinomios de forma axiomática. Nosotros formalizamos totalmente dicha teoría, utilizando como únicos axiomas los de la formalización de cuerpo.

Implementamos el conjunto de los polinomios como un *Setoid*[3], tipo base que parametriza todo nuestro desarrollo posterior. Para ello definimos los polinomios con respecto a un cuerpo base, definiendo las operaciones y una relación de equivalencia que representa la igualdad matemática de polinomios. Respecto a la formalización de polinomios elegimos dos niveles de aproximación:

- Definimos polinomios como listas de monomios, posiblemente no ordenados, implementando sus operaciones algebraicas. Con esto conseguimos definiciones claras y sencillas, matemáticamente hablando, aunque ineficientes desde el punto de vista calculatorio.

- Más tarde, formalizamos los polinomios canónicos, así como nuevas versiones de las operaciones sobre estos polinomios, mostrando que son matemáticamente equivalentes a las anteriores. Por razones de eficiencia, las

[2] Este año se celebra un workshop para recordar los 30 años de existencia del sistema Mizar, MKM 2004 "30 years of Mizar".

[3] Un *Setoid* es un triple compuesto de un tipo S, una relación R sobre S, y una prueba de que R es una relación de equivalencia sobre S.

pruebas constructivas se hacen exclusivamente sobre polinomios canónicos.

Hace falta remarcar que cuando realizamos el cálculo de una base de Gröbner con ayuda de alguno de los sistemas comerciales existentes (Maple, Mathematica, Axiom, Reduce, ... etc), ¿cómo podemos estar seguros de que este resultado es correcto si un simple cambio de orden en la lista de las variables ó en la relación de orden elegida, produce cambios notables en el resultado? Además, si lo hacemos con dos sistemas de Cálculo Simbólico diferentes, probablemente obtengamos un resultado distinto, aunque teóricamente equivalente.

La motivación central en este trabajo ha sido la de formalizar en el sistema **Coq** todos los conceptos necesarios, desde los más básicos (cuerpo, términos, polinomios, ideales, etc), para demostrar el teorema de Buchberger pieza básica en el algoritmo para el cálculo de las bases de Gröbner. Asimismo, se proporcionará una manera de extraer programas automáticamente a partir de especificaciones. Esto sólo es posible en un contexto donde convivan tanto las construcciones de nuevos objetos (parte conjuntista) como las propiedades de estos objetos (parte proposicional).

Uno de estos contextos es **Coq**, una implementación del Cálculo de Construcciones inductivas (CIC) de Gérard Huet, hecha en el INRIA [73]. Este sistema es un probador automático de teoremas dirigido a objectivos (*goal-directed*) y conducido por tácticas (*tactic-driven*) en el estilo de LCF [61], que permite la verificación formal de programas y, asímismo, su generación automática.

Mostraremos la potencia del extractor de programas en **Coq**, y su utilización para definir programas directamente desde las especificaciones.

También esperamos que esta reflexión sobre la herramienta Coq sirva para despertar el interés por los probadores automáticos y mostrar, con un ejemplo no trivial, que es posible, con las tecnologías actuales, conjugar prueba y cálculo formal. Asimismo, también pretendemos resaltar la importancia que tienen en informática las pruebas constructivas y la conexión (isomorfismo de Curry-Howard) entre prueba y programa, como herramienta matemática de certificación de software.

1.2. La mecanización de las matemáticas

La idea de la mecanización de las matemáticas existe desde mucho tiempo atrás; reconocida su utilidad, sólo desde hace unos veinte años parece realizable.

Como un ejemplo de la importancia de las demostraciones en matemáticas, reproducimos a continuación un extracto de la introducción de N hecha por Bourbaki:

Despuis les Grecs, qui dit mathématique dit démonstrations; certains doutent même qu'il se trouve, en dehors des mathématiques, des démonstrations au sens précis et rigoureux que ce mot a reçu des Grecs et qu'on entend lui donner ici. On a le droit de dire que ce sens n'a pas varié, car ce qui était une démonstration por Euclide en est toujours une à nos yeux; et, aux époques où la notion

a menacé de s'en perdre et où de ce fait la mathématique s'est trouvée en danger, c'est chez les Grecs qu'on en a recherché les modèles. Mais à ce vénérable héritage sont venues s'ajouter depuis un siècle d'importantes conquêtes.

Suele citarse a los lógicos Boole, Frege y Peano como los pioneros en la búsqueda de marcos formales correctos para hacer razonamientos matemáticos. La "teoría de conjuntos" de Zermelo y la "teoría de tipos" de Russell y Whitehead nacieron del esfuerzo de hallar un sistema de notaciones riguroso para fundamentar las matemáticas. Hilbert va más allá, con su deseo de conseguir el razonamiento correcto como consecuencia del cálculo. La llegada de los ordenadores dan un soplo de aire fresco a los proyectos de mecanizar el razonamiento para ciertas partes de la matemática y efectuar su demostración de manera interactiva.

En 1954, aunque publicada en 1957, apareció la primera prueba matemática realizada en un programa de ordenador: escrita por M. Davis en el "Institute of Advanced Studies", USA, probaba un teorema de teoría de números en la aritmética de Presburger. A partir de aquí, se sucedieron los intentos de mecanización como se puede ver en [109].

En los años 70, Wu Wenjun[4], dándose cuenta de la potencia de los ordenadores, consideraba a las matemáticas mecanizadas, combinación de las matemáticas clásicas y de la informática, como una de las principales orientaciones del futuro desarrollo de las matemáticas. Así como la revolución industrial había instaurado la mecanización del trabajo manual, la mecanización de las matemáticas instauraría la mecanización del trabajo intelectual.

1.3. Demostración automática

Un programa de demostración automática es un programa que acepta como entrada una proposición y diversos axiomas y devuelve una demostración de dicha proposición bajo los axiomas introducidos.

Los pioneros que concibieron los primeros programas de demostración automática [20, 25] a finales de los cincuenta, tenian proyectos muy ambiciosos. Algunos de ellos preveian, que en diez años como máximo, los programas de demostración automática reemplazarían a los matemáticos y resolverían problemas abiertos.

Estos proyectos están hoy en día abandonados, al menos de momento. Por una parte, se ha visto que la inteligencia no se reduce a la capacidad de razonar formalmente. Por otra parte, si algunos programas de demostración automática son, en teoría, capaces de demostrar teoremas difíciles, el tiempo que requieren para hacerlo es desorbitado con lo cual estos programas no tienen interés práctico.

Aunque la demostración automática no consiguió convertir los ordenadores en inteligentes, ni resolver problemas matemáticos difíciles, si consiguió despertar el interés en otros campos y encontrar aplicaciones, rebajando sus preten-

[4] Matemático chino, presidente del Congreso Internacional de matemáticas 2002 y Premio Nacional de Ciencia en el año 2000, cuando contaba 80 años.

siones iniciales. Y así, los sistemas de demostración automática derivaron en sistemas interactivos de prueba, lo que hoy llamamos "sistemas de ayuda a la demostración" de los cuales el sistema Coq es un ejemplo. Hay que decir que estos sistemas de ayuda a la demostración utilizan módulos de demostración automática.

1.4. Ayuda a la demostración

El estudio de los sistemas de ayuda a la demostración [111], cada día más utilizados en numerosas áreas (cálculo formal, validación de programas, programación lógica, etc.), es un dominio aplicado de la informática que emplea los métodos algorítmicos de la lógica para la resolución de ciertos problemas ligados a las relaciones de deducción.

Ya hemos comentado que la demostración automática nació historicamente de las tentativas de mecanizar las matemáticas, y por consiguiente de automatizar la "producción" de teoremas. Los primeros trabajos, como tal área de la ciencia, se desarrollaron a partir del descubrimiento por Alan Robinson del principio de resolución que es la base de los conceptos y herramiemtas de la programación lógica y se ha convertido hoy en día en uno de los principales paradigmas de programación. Los años 80 consagraron la automatización del razonamiento ecuacional, permitiendo la manipulación mecánica de las especificaciones algebraicas.

1.4.1. Corrección de los programas informáticos

Todo el aparato lógico [77, 79, 92, 8, 89] destinado a estudiar el lenguaje formal de las matemáticas, permite estudiar otras formalizaciones de lenguaje: los lenguajes de programación. Así vemos que, en efecto, el problema de la demostración automática está estrechamente ligado a la corrección de programas. Es este un problema cada día más importante debido a la aparición de numerosos sistemas críticos que tienen el riesgo de tener consecuencias trágicas, tales como la aviación, el frenado de los coches, el control de los reactores de una central nuclear, aplicaciones en medicina, lanzamiento de satélites, seguridad de tarjetas bancarias, etc. La idea es traducir las propiedades de los programas en una fórmula lógica (formalización) de la cual se va a intentar buscar un prueba. Si se puede encontrar una prueba de la fórmula, entonces tenemos asegurado que el programa posee la propiedad esperada, suponiendo que la propiedad fué correctamente codificada. El conjunto de estas propiedades forma lo que se llama "especificación del programa".

En los años 70, un modo de probar un programa era la utilización del **método de Hoare** [46, 59]. Para un lenguaje dado, las fórmulas de Hoare son de la forma $\{P\}S\{Q\}$, donde S es la especificación del programa y P, y Q expresiones que traducen las propiedades de las variables de S. A P se le denomina *precondición* y a Q *poscondición*. El problema de esta aproximación es que las fórmulas de Hoare no aportan más que la corrección parcial del programa. Para probar la

corrección total, debemos probar que termina. Otro problema añadido es que el usuario debe eligir las "buenas" fórmulas P y Q, pues no hay medio automático de derivarlas. Si se escribe la demostración en un papel, se obtiene una prueba con un grado de precisión parecido al de una demostración matemática. Es en esto donde los métodos formales son útiles permitiendo escribir las fórmulas P y Q como términos de un lenguaje lógico.

Así, a partir de la utilización de la lógica, aparecen un cierto número de sistemas, denominados sistemas de ayuda a la prueba, destinados a probar propiedades de los programas. Estos sistemas se pueden clasificar según varios criterios: la lógica subyacente a cada sistema, el grado de automatización, el nivel de seguridad intrínseca, el lenguaje de implementación, la naturaleza imperativa o funcional, etc. Claramente, estos sistemas aportan un plus con relación a los métodos como el de Hoare, entre otros: un nivel de seguridad superior, y sobre todo, la capitalización de los errores y de la experiencia. En efecto, si se descubre un error en uno de estos sistemas, todos los usuarios posteriores se beneficiarán de ello, lo que contribuye a una mayor seguridad en aplicaciones posteriores.

1.4.2. Herramientas de verificación formal

Hacer de la programación una disciplina formal, con un método sistemático, requiere integrar las actividades de programar, especificar y probar en el ciclo del desarrollo lógico. Los investigadores en informática tienen el desafío de proponer herramientas lógicas (adaptadas a las actividades de programación, entornos de prueba, nuevos lenguajes de programación, etc.) que puedan ayudar a los ingenieros informáticos a comprobar rigurosamente la corrección de su programa respecto a la especificación.

Varios demostradores automáticos y entornos interactivos de prueba se desarrollaron en los últimos tiempos. Entre los primeros se encuentra **NQTHM** [24] y sistemas más recientes como **ACL2**[5] [75] y **PVS** [93], que combina potentes herramientas automáticas con entornos interactivos. Entre los asistententes interactivos podemos citar **AUTOMATH** [28] y **LCF** [38, 100] en los años 70; mas recientemente, **Alf** [82], **Coq** [73], **Isabelle** [101], **HOL** [60], **Lego** [81, 80]. En estos sistemas el usuario puede guiar la construcción de la prueba indicando el lema que se debe aplicar y definir la estrategia (por casos, por recurrencia, etc.) de prueba que se quiere aplicar.

Estos sistemas implementan diferentes formalismos lógicos. El programador puede elegir entonces el cuadro lógico donde razonar sobre los programas. Hay una gran variedad; por ejemplo el cálculo proposicional, la lógica de primer orden, de segundo orden, de orden superior, los sistemas de tipos, etc. Es importante resaltar que la mayoría de estos sistemas de prueba permiten definiciones inductivas y aportan herramientas para razonar por casos y por recurrencia[6].

Sin embargo, el desafío de conseguir sistemas de prueba fácilmente utilizables por la comunidad matemática está aún lejos de ser conseguido. El objetivo de

[5] Sistema de razonamiento automático que proviene de NQTHM, el demostrador de teoremas de Boyer y Moore.

[6] El pionero en la automatización de la recurrencia fué el sistema NQTHM.

formalizar se paga con el nivel de detalle con el cual el usuario debe modelizar su demostración en el sistema de prueba. La formalización de una prueba exige la justificación de cada etapa del razonamiento, incluso las que son triviales (por ejemplo, $0 = -0$). El usuario experimenta el fenómeno de que, demostraciones sencillas con papel y lápiz, se corresponden, a menudo, con pruebas largas y con mucho detalle en los sistemas de prueba.

Un buen sistema de prueba debe tener fundamentalmente las dos propiedades siguientes:

- Buenas propiedades de la teoría subyacente. Por ejemplo, en relación con el poder expresivo, poder escribir tantos algoritmos como sea posible así como métodos de cálculo.
- Herramientas de ayuda a la construcción de pruebas que sean fáciles de utilizar, tanto las que conciernen al lenguaje de especificación del sistema para definir objetos, como las tácticas automáticas para ayudar al usuario a hacer la demostración.

1.4.3. Estado del arte de los diversos sistemas de prueba

Después de haber comentado la necesidad de contar con herramientas para mecanizar las matemáticas y para probar que los programas informáticos verifican ciertas propiedades, intentamos presentar, sin pretender ser exhaustivos, la panoplia de los diferentes sistemas existentes así como un pequeño comentario y referencias históricas sobre algunos de ellos.

NQTHM: Boyer y Moore [26] abordaron en 1971 la prueba de programas en Lisp mediante una aproximación recursiva primitiva. La primera versión del sistema trataba con ecuaciones recursivas primitivas expresadas en lenguaje Lisp. Más tarde este sistema evolucionó. La introducción de una noción de tipos inductivos permitió en este sistema desarrollar pruebas inductivas uniformes, procediendo por recurrencia sobre el tamaño de la estructura de los objetos definidos. Más adelante, el sistema fué enriquecido al poder manipular cuantificadores y el añadido de un cálculo de predicados.

Este sistema se ha beneficiado de un desarrollo continuo durante 30 años, tanto como herramienta teórica como en las pruebas; lo que contribuyó a la elaboración de bibliotecas (teorema fundamental de la aritmética, teorema de Wilson, etc).

La particularidad de este sistema reside en el hecho de que es fundamentalmente no tipado.

OTTER: (Organized Techniques for Theorem-proving and Effective Research) es un demostrador automático de teoremas para la lógica de primer orden con igualdad, que utiliza el lenguaje de cláusulas, desarrollado por W. McCune [85] en 1988.

Nuprl: Se gestó en los años 70 y se utilizó para formalizar, de forma constructiva, las matemáticas utilizando la teoría de tipos de Martin Löf. Su aplicación principal fué el desarrollo del análisis constructivo de Max Forester [50]

HOL: Este sistema es un descendiente directo de LCF[7] y su implementación se debe a Robin Milner [61]. En el asistente de prueba HOL (Higher-Order Logic), los términos pertenecen a un λ-cálculo tipado "a lo Curry" con constantes y polimorfismo. Hay dos tipos atómicos *bool* e *ind* que definen respectivamente el conjunto de proposiciones y el de conjuntos (se corresponden con *Prop* y *Set* en Coq). Las fórmulas primitivas se construyen a partir de la implicación, la igualdad polimórfica y del operador de Hilbert. Los demás operadores lógicos se definen por igualdades.

A diferencia de Coq la inducción no está integrada en el sistema; las definiciones inductivas estan automatizadas con la ayuda de puntos fijos. En HOL están disponibles un conjunto de herramientas para generar automaticamente pruebas por recurrencia y por inversión.

Este sistema posee una amplia comunidad de usuarios de la cual surgen un gran desarrollo de pruebas. Puede obtenerse por **ftp** en *ted.cs.uidaho.edu* en el directorio ftp/pub/hol.

PVS: El sistema PVS (Prototipe Verification System) se compone de un lenguaje de especificación y un sistema interactivo de ayuda a la prueba. El lenguaje de especificación se basa en la lógica clásica de orden superior, enriquecido por una definición de subtipado. Más concretamente, un subtipo está descrito por un conjunto de elementos que verifican un cierto predicado: si T es un tipo y P un predicado sobre T, $P : T \rightarrow bool$, el tipo de todos los elementos de T que verifican P es $\{x : T | P(x)\}$. Esta flexibilidad del sistema de tipos se hace en detrimento de la decibilidad del tipado.

Este sistema está mucho más automatizado que Coq o HOL, sin embargo ofrece al usuario más control sobre las pruebas que otros como NQTHM u OTTER. Como dice el tutorial de PVS: *una prueba debe ser un objeto que sea humanamente legible sin demasiado esfuerzo, pero tal que esté subordinada a las reglas de la comunidad matemática.*

Isabelle: Fué desarrollado inicialmente por Larry Paulson en la Universidad de Cambridge, y ahora codirigido también desde Munich por un equipo encabezado por Tobias Nipkow. Su principal uso es la formalización de demostraciones matemáticas (incluye muchas teoría matemáticas formalmente verificadas) y la verificación de la corrección de software. Isabelle es un sistema de prueba al estilo LCF utilizando como herramienta para demostrar teoremas la *unificacion de orden superior*; incorpora herramientas para automatizar algunas partes del proceso de prueba (por ejemplo, las propiedades aritméticas lineales se demuestran automaticamente).

A destacar su flexibilidad; de hecho es un probador genérico que permite utilizar lógicas diferentes. Más información acerca de Isabelle, incluyendo tanto software como un tutorial, se puede obtener en *http : //isabelle.in.tum.de/*.

Alf: Agrega un mecanismo de filtrado primitivo a la teoría de tipos, que implementa por la unificación de orden superior. La teoría resultante es más potente que la utilizada por la reglas de eliminación de Coq. Está poco extendido y no

[7] Logic for computable functions.

es interactivo.

Automath: Es el más antiguo de los sistemas de prueba que realizaba demostraciones matemáticas no triviales. Se implementó a partir de la tesis de Bruijn, el cual afirmaba que "este lenguaje es suficiente para implementar gran parte de las matemáticas". Fué poco utilizado.

Lego: Fué desarrollado en Edimburgo por Rod Burstall y Randy Pollack. Se utiliza para formalizar matemáticas así como para la especificación y verificación de programas. Utiliza la teoría de tipos separando, al estilo de Coq, tipos de datos y proposiciones lógicas. Implementa tipos inductivos y predicados. Se puede ver más en *http://www.dcs.ed.ac.uk/home/lego/*.

ACL2: Son las siglas de *A Computational Logic for Applicative Common Lisp*. El sistema *ACL2* es, al mismo tiempo, un lenguaje de programación, una lógica para razonar sobre los programas construidos en ese lenguaje y un demostrador que automatiza, en cierta medida, el razonamiento en lógica. Este sistema es un descendiente directo, como muchos otros, del demostrador de Boyer y Moore, NQTHM, del que hereda parte de su filosofía y su lógica para poder razonar sobre un subconjunto aplicativo de Common Lisp. Utiliza una lógica de primer orden con igualdad sin cuantificadores.

1.4.4. ¿Por qué verificar en el sistema Coq?

Las principales razones por las que hemos elegido el sistema Coq para nuestras formalizaciones y pruebas, entre todos los sistemas citados anteriormente, son las siguientes:

- El aspecto más atrayente del sistema Coq es la posibilidad de extraer un programa ejecutable de las demostraciones constructivas. Es conocido que las pruebas en lógica intuicionista permiten, mediante el isomorfismo de Curry-Howard, deducir algoritmos correspondientes a las propiedades probadas. Se puede utilizar la información calculatoria proveniente de las pruebas efectuadas. En efecto, una función, representada por un predicado P, cuya proposición $\forall x\ \exists y\ (P\ x\ y)$ se demuestra de manera intuicionista, es calculable.

- Tiene una comunidad de usuarios muy extendida e implementa mejoras constantes del sistema. En estos momentos está en la versión 8.1.

- Está adaptado para elaborar pruebas complicadas al estilo **LCF**. Basado en una teoría matemática reciente y compleja, como es el Cálculo de Construcciones [40, 39], pone a nuestra disposición herramientas abstractas muy potentes (por ejemplo, la implementación de tipos dependientes) adaptadas a la manera de razonar en matemáticas.

- Coq es un lenguaje mientras que otros, como HOL e Isabelle, pueden considerarse como bibliotecas.

- La sintaxis abstracta de Coq, mediante una función de normalización de tipos, permite "pattern matching" sobre los tipos inductivos mediante los constructores.
- Control sobre el tipado, no permitiendo el tipado incorrecto.
- Flexibilidad para formalizar y probar propiedades sobre los tipos definidos, así como la posibilidad de trabajar sobre funciones totales.

No obstante, entre los problemas que hemos detectado en la utilización de Coq destacamos los siguientes:

- Problemas con la utilización de la reescritura; por ejemplo el lenguaje de tácticas de HOL está mucho más adaptado al uso de la reescritura de manera intensiva.
- Falta de tácticas de automatización. Muchas veces, pruebas triviales necesitan pruebas interactivas muy largas y pesadas.
- Se echan en falta tácticas para el desdoblamiento automático de constantes.
- Limitaciones en el trabajo con los números reales y racionales. No existen, exceptuando las tácticas **Ring**[8] y **Field**[9], tácticas especializadas que simplifiquen las pruebas en este dominio; existen lineas de investigación abiertas sobre este tema, [23, 84], así como algunas implementaciones.
- El lenguaje (de alto nivel) de especificación **Gallina**, mediante el cual el sistema Coq representa las pruebas realizadas utilizando las tácticas, es difícil y engorroso de leer.
- Los mensajes de error, ver manual [73], no son claros y por lo tanto no siempre es fácil detectar dichos errores.

1.5. Plan de la tesis

En el capítulo 2, abordamos, de forma ajustada a lo que vamos a necesitar, las nociones más practicas del sistema **Coq**. Éstas van a ser necesarias para comprender la formalización de la teoría matemática desarrollada en los capítulos siguientes.

El capítulo 3 está dedicado a la formalización de los términos de n variables, así como las operaciones más usuales. Se implementa el orden lexicográfico, profundizando en la prueba de la noetherianidad de dicho orden.

En el capítulo 4 se describe la axiomatización de la estructura de cuerpo abstracto y se verifican algunas de sus propiedades.

[8] Desarrollada en OCaml por S. Boutin y P. Loiseleur.
[9] Desarrollada por M. Mayero y D. Delahaye.

Utilizando lo descrito en los capítulos 2 y 3, se introduce en el capítulo 5 la noción de monomio, comprobando algunas de sus características.

En los capítulos 6 y 7 nos adentramos en el corazón del trabajo, los polinomios; formalizamos la igualdad de polinomios, distinta de la del sistema; describimos sus operaciones; terminamos implementando un orden de polinomios, definido a partir del orden lexicográfico de términos, demostrando que es noetheriano.

Después, en el capítulo 8, abordamos el concepto de ideal de polinomios, que jugará un papel determinante en el desarrollo de los capítulos siguientes. Nos interesamos por las propiedades más importantes de esta estructura relacionadas con las operaciones de polinomios.

El objetivo del capítulo 9 es generalizar el algoritmo de la división de un polinomio en varias variables por una lista de polinomios en varias variables. Una vez implementada dicha división, que llamaremos reducción, estudiamos la relación entre congruencia y reducción así como la noetherianidad de la reducción. Calculamos la forma normal módulo un conjunto de polinomios, terminando con el estudio de las propiedades que relacionan los grados de los polinomios iniciales y resultantes de dicha reducción.

En el capítulo 10 introducimos el concepto de Bases de Gröbner; estudiamos y probamos su equivalencia con otras caracterizaciones alternativas que serán útiles para la construcción de dichas bases. En particular, la que utiliza el Algoritmo de Bucheberger para resolver el problema de las "situaciones críticas" y que es el objetivo final de este trabajo. Para resolver estas equivalencias damos una versión del Lema de Newman adaptada a nuestra formalización, así como la definición y propiedades de la confluencia y confluencia local de la reducción de polinomios

En el primer apéndice comentamos de forma muy sucinta el procedimiento de extracción del sistema Coq y mostramos el código extraido, sin optimizaciones, a partir de pruebas calculatorias realizadas a lo largo de este trabajo. Asimismo, en el segundo apéndice se adjuntan dos módulos, uno de ellos es de utilidad general sobre listas y permite razonar por recurrencia sobre la longitud de una lista; el segundo, contiene la prueba en Coq, de que la propiedad irreflexiva es una propiedad inherente a cualquier relación bien fundada.

1.6. Como leer esta tesis

Esta tesis se ha escrito de manera secuencial, cada capítulo ligado sobre todo al precedente. También se ha procurado que sea autocontenida; para ello, hemos hecho una introducción al sistema utilizado y cada formalización se ha acompañado de la noción matemática que representa. Asimismo, la mayoría de las pruebas realizadas están comentadas, intentando conjugar la prueba abstracta con la seguida en el sistema de ayuda a la prueba.

Las pruebas que comentamos siguen el guión realizado en Coq, lo que nos permite abordar las matemáticas bajo un ángulo de "métodos formales". Algunas de estas pruebas pueden parecer triviales desde un punto de vista matemático,

pero queremos presentar las pruebas de la manera más "formal posible". Decir que muchas veces los problemas encontrados en las pruebas formales nos sugirieron caminos matemáticos distintos de los inicialmente pensados y otras muchas veces nos obligaron a volver atrás[10] y probar resultados que no habíamos tenido en cuenta.

Aunque algunas de las demostraciones realizadas en Coq podrían acortarse; preferimos no hacerlo por razones de legibilidad y "acercamiento" a la metodología del sistema.

Los símbolos y tipos predefinidos en el sistema, por defecto, así como los conceptos matemáticos los hemos escrito en negrita; las construcciones específicas de la tesis se han puesto en cursiva.

La mayor parte del desarrollo hecho en Coq (versión V 7.3.1) está presentado en este trabajo y su funcionamiento disponible una vez cargado el sistema. Todas las definiciones, lemas y teoremas que figuran en *verbatim* han sido probadas formalmente en Coq y están disponibles en un repositorio de código [2]. Antes de cada una de las formalizaciones y demostraciones hemos escrito su notación usual en matemáticas; nos parece más accesible tanto a los no iniciados en Coq, como a los no familiarizados con esta teoría matemática.

Los nombres de las definiciones y de los lemas se refieren normalmente a la propiedades que representan; en casos concretos a las personas que los sugirieron o a los nombres de las pruebas donde son utilizados.

[10] Por ejemplo, la mayoría de los resultados de la sección 6.8.

Capítulo 2

Una breve introducción a la utilización de Coq

> The theory of recursive functions provides a notion of constructivity within the framework of classical mathematics. Roughly speaking, it is concerned with the constructive character of results rather than proofs, while the latter are considered in intuitionistc mathematics.
>
> Kreisel[1]

En este capítulo haremos una pequeña presentación del sistema que vió la luz en 1984 bajo el impulso de Gérard Huet y Thierry Coquand, **Coq**, describiendo algunas de sus características. El objetivo no es presentar el sistema en todos sus detalles. Nos ceñiremos a dar una descripción que sea suficiente para facilitar la comprensión de los fragmemtos de código que presentamos en los capítulos siguientes. Comenzando por esbozar la lógica de este sistema, describiremos brevemente las nociones más básicas del mismo así como los mecanismos de definición, prueba y axiomatización de teorías que necesitamos para nuestro posterior desarrollo. Más detalles de Coq se pueden ver en [73, 43, 94, 97, 18, 72].

2.1. La lógica

Todo sistema de pruebas formales reposa sobre un sistema lógico concreto. El sistema **Coq** [1] utiliza la lógica intuicionista; otros como **PVS** y **HOL** se basan en la lógica clásica.

La lógica **intuicionista**, ver [83], es constructiva [16, 113, 114], es decir que probar una proposición de la forma *existe* x *tal que* ..., significa que se sabe construir tal x. De este hecho se deduce que, a diferencia de la lógica clásica, el axioma del tercio excluso[2] no es válido. Por ejemplo la interpretación en

[1] G. Kreisel en Proceedings of the Colloquium on Constructivity in Mathematics at Amsterdam, 1957.

[2] Para toda proposición P, la proposición "P *o no* P" es verdadera.

matematicas intuicionistas de la connectiva ($\neg$) es la siguiente: $\neg P$ es válida sí, y sólo sí, tenemos un método en el cual, supuesta una prueba constructiva de P, entonces al aplicar este método a P nos lleva a una contradicción ($P \rightarrow False$).

Coq es un probador basado sobre la teoría de tipos e implementa el Cálculo de Construcciones Inductivo (CCI) [41, 48, 97], una teoría de tipos que resulta de la combinación de la teoría intuicionista de tipos de Martin-Löf y del λ-cálculo polimorfico de Girard F_ω.

El cálculo de construcciones es un λ-cálculo tipado de orden superior concebido por T. Coquand y G. Huet en 1.985. Formalmente, un λ-cálculo tipado [71, 7] define un lenguaje de términos y tipos, así como un conjunto de reglas que describen como se derivan sentencias de tipo de la forma $\Gamma \vdash M : \alpha$ (que se lee "M es un término de tipo α en el contexto Γ"). Referimos a [39] para un estudio detallado de las reglas de inferencia del cálculo de construcciones.

En este cálculo, los tipos son también términos. Su gran potencia de expresión permite definir tanto estructuras de datos como proposiciones lógicas.

La lógica constructiva, en la cual se apoya Coq, permite hacer una analogía entre pruebas y programas llamada **isomorfismo de Curry-Howard** (o de Curry-De Bruijn-Howard en [49]). Este resultado de 1969 [67], da sentido a la utilización de la teoría de tipos en el dominio de la modelización, tanto en informática como en matemáticas.

El isomorfismo de Curry-Howard permite, por la semántica de Heyting-Kolmogorov[3], identificar el tipo de una prueba con la fórmula que ella demuestra, identificando los principales conectores lógicos con los operadores apropiados del lenguaje de programación. Una proposición se puede ver como el tipo de su prueba[4]; las proposiciones no demostrables son tipos vacios. Gracias a esta particularidad Coq posee un "principio de extracción", que no es posible con la lógica clásica y que nos permite aislar el contenido calculatorio de la formalización de una prueba. El principio de extracción de Coq consiste en la obtención de un programa *OCaml* a partir del término de prueba; se puede ver más en [95, 96].

El lenguaje de especificación que utiliza Coq se denomina **Gallina**. Este lenguaje es el que nos permite introducir objetos y probar propiedades. En particular permite definir: tipos de datos estructurados, funciones recursivas sobre la estructura de los datos, fórmulas del cálculo de predicados de orden superior, relaciones especificadas inductivamente por un conjunto de propiedades de clausura, etc. Para profundizar en este lenguaje ver [41, 70].

2.1.1. Las Clases de Coq

Al tratarse de una lógica de orden superior tipada, vamos a considerar los objetos agrupados en lo que llamamos tipo. Estos objetos se representan por λ-términos. Por estar en una lógica de orden superior, los tipos son ellos mismos

[3] El primer sistema formal de la matemática intuicionista fué presentado en 1.930 por Heyting.

[4] Una prueba de $\forall x.P(x)$ es una función que a cada término t le asocia una prueba de $P(t)$. Y una prueba de $\exists x.P(x)$ es un par formado por un término t y una prueba de $P(t)$.

objetos de los que se puede hablar.

En el cálculo de construcciones, se llama **clase** al tipo de un tipo (considerado como término). Una clase es siempre una constante.

El formalismo intuicionista de Coq acompañado del principio de extracción, requiere la introducción de una clase particular, la clase **Set**.

La clase *Set*

La clase **Set** ha sido definida para ser el tipo de las especificaciones. Esto incluye los programas y conjuntos usuales tales como los booleanos (*bool*), enteros (*nat*), listas (*list*) etc. **Set** es una clase impredicativa, es decir que $(A : Set)A \rightarrow A$ es de tipo **Set**. Desde el punto de vista de la extracción de programas, **Set** es el tipo de las proposiciones con contenido constructivo.

La clase **Set** tiene eliminación fuerte, lo que significa que es posible recuperar el testigo de un teorema de existencia definido con un valor en **Set**.

La clase *Prop*

La meta de una lógica no es únicamente definir objetos, sino sobre todo enunciar propiedades de esos objetos. Es por esto por lo que existen objetos que se asimilan de alguna manera a las frases de un lenguaje natural: son las proposiciones. El tipo de las proposiciones es **Prop**.

Prop es el universo de las proposiciones lógicas; esto incluye **True** y **False**. Un objeto de tipo **Prop** se llama proposición. **Prop** es igualmente impredicativo, es decir que $(P : Prop)P \rightarrow P$ es de tipo **Prop**.

Al contrario que en **Set**, en **Prop** no hay eliminación fuerte, es decir que no se puede construir objetos **Set** a partir de una proposición lógica. En la práctica, eso significa que es imposible, por ejemplo, recuperar el testigo de un teorema de existencia cuando se trata de una propiedad que no sea una proposición con contenido constructivo. Es decir que una propiedad que contenga información calculatoria, tipo **Set**, no puede ser probada a partir de una proposición de tipo **Prop**.

Ejemplo 2.1.1.

```
  P : Prop
  H : P\/~P
  ============================
   {P}+{~P}

ss < Elim H.
Error: Cannot find the elimination combinator or_rec
 The elimination of the inductive definition or on sort
 Set is probably not allowed
```

Las clases anteriores son términos del cálculo de construcciones y deben a su vez poseer un tipo. Pero este tipo, todavia un término, va a tener un nuevo tipo, etc.

La clase *Type*

Puesto que decir "que un tipo A es él mismo de tipo A" produce incoherencia [56], existe en **Coq** una jerarquía infinita de clases llamadas universos denotada por **Type**. Esta familia está formada por clases $Type(i)$, para todo i en $\mathbb{N}$, y se caracteriza por las relaciones siguientes:

$$Set \ : \ Type(i) \quad (\forall i \in \mathbb{N})$$

$$Type(i) \ : \ Type(j) \quad (\text{si } i < j \quad \forall i, j \in \mathbb{N})$$

Por ejemplo, para todo natural i se tiene la propiedad siguiente: $Type(i)$ es de tipo $Type(i+1)$[5]. Este método permite paliar el problema de la paradoja de Girard [57]. De este hecho, se deduce fácilmente que la clase **Type** es predicativa, es decir, que: $(T \ : Type(i))\ S \ \rightarrow \ S$ es de tipo $Type(i+1)$; más en [18].

2.1.2. Reglas de conversión

El Cálculo de construcciones tiene un mecanismo de conversión que permite decidir si dos términos son intensionalmente iguales (convertibles).

Existen varias reglas de reducción en **Coq**, resumimos a continuación las más utilizadas. Si t y u son términos y v una variable de tipo τ; entonces $t\{v/u\}$ denota la sustitución de las apariciones libres de v por u en el término t.

β-reducción: permite transformar un β-radical[6], es decir un término de la forma $([v : \tau]t\ u)$ en el término $t\{v/u\}$.

Por ejemplo, para los términos $t = A \rightarrow A$ y $u = nat \rightarrow nat$, el término $t\{A/u\}$ es: $(nat \rightarrow nat) \rightarrow (nat \rightarrow nat)$.

δ-reducción: sea un entorno E, un contexto Γ, un término t, e identificadores e y c definidos por t, respectivamente, en el entorno y en el contexto en curso. Entonces la δ-reducción expande los identificadores en el término t. Está reducción concierne tanto a las variables del contexto, como a las constantes del entorno global. La notación usual es de la forma siguiente:

$$E[\Gamma] \vdash c \rhd_\delta \ t \quad \text{si} \quad (c := t : T) \in \Gamma$$

$$E[\Gamma] \vdash e \rhd_\delta \ t \quad \text{si} \quad (e := t : T) \in E$$

ι-reducción: está especificamente ligada a los objetos inductivos, siendo la responsable de ciertas simplificaciones ligadas a los esquemas recursivos, por ejemplo las reducciones de $(plus\ 0\ n)$ a n. Un ι-redex es un término de la forma: $< P > \ Cases\ (c_{pi}\ q_1 \ldots q_r\ a_1 \ldots a_m)\ of\ f_1 \ldots f_l\ end$ con c_{pi} el i-ésimo constructor del tipo inductivo I con r parámetros. La ι-reducción de este término es $(f_i\ a_1 \ldots a_m)$, es decir:

$$< P > Cases\ (c_{pi}\ q_1 \ldots q_r\ a_1 \ldots a_m)\ of\ f_1 \ldots f_l\ end \rhd_\iota \ (f_i\ a_1 \ldots a_m)$$

[5] En Coq, los índices de los universos no son visibles para el usuario. Se abrevian en $Type$.

[6] β-redex en inglés.

ζ-reducción: consiste en la eliminación de las ligaduras locales; más concretamente, reemplaza una expresión de la forma $[v := u]t$ por $t\{v/u\}$.

El sistema Coq suministra herramientas, como **Eval**, **Compute** o **Cbv**, para realizar estas conversiones, ver [73].

```
Coq < Eval Compute in (plus (S (S 0)) (S (S (S 0)))).
     = (S (S (S (S (S 0)))))
     : nat
```

2.2. Vocabulario y notaciones

En **Coq** el desarrollo de las pruebas se hace de modo interactivo utilizando órdenes llamadas tácticas. El sistema utiliza un razonamiento "hacia atras". La conclusión es la **meta** y las premisas necesarias según la táctica son las **submetas**. Cuando aplicamos una táctica, el sistema reemplaza la meta por las submetas generadas. El sistema en cada etapa muestra las submetas que quedan por probar.

En la tabla siguiente se muestra la correspondencia entre la sintaxis usual de matemáticas y la de Coq.

Noción	Denominación matemática	Notación de Coq
Falso	$\bot$	False
Verdad	$\top$	True
Negación	$\neg$	$\sim$
Implicación	$\Rightarrow$	$\rightarrow$
Equivalencia	$\Leftrightarrow$	$\leftrightarrow$
Conjunción	$\wedge$	$\bigwedge$
Disyunción(Prop)	$\vee$	$\bigvee$
Suma(Set)	$\vee$	$\{\ldots\} + \{\ldots\}$
Cuantificación universal	$\forall x \in A$	$(x : A)$
Cuantificación existencial(Prop)	$\exists x \in A \ldots$	$(Ex\,[x : A]\ \ldots)$
Cuantificación existencial(Set)	$\exists x \in A$	$\{x : A \mid \ldots\}$
Lambda abstracción	$\lambda x : A\,.\,t$	$[x : A]\,t$
Producto no dependiente	$A \rightarrow B$	$A \rightarrow B$
Producto dependiente	$\prod x \in A.U$	$(x : A)U$
Aplicación	$f\,(x)$	$(f\ x)$

Hay que hacer notar que en **Coq** sólo dos conectores son primitivos: la implicación y la cuantificación universal; los demás se definen inductivamente.

2.3. Mecanismos de definición

En esta sección se presenta la sintaxis del sistema que permite introducir objetos y probar propiedades. Hay dos maneras de hacerlo:

- **Por axiomatización**: se declaran las variables correspondientes a los objetos sobre los cuales se quiere razonar, y sus propiedades se expresan como axiomas.

- **Con definiciones**: se forman funciones con la λ-abstración. Además se dispone en Coq de tipos inductivos, que permiten introducir nuevos tipos de manera análoga a los tipos concretos de **Caml Light**.

2.3.1. Axiomatización

Para esto disponemos fundamentalmente de dos órdenes **Variable** y **Axiom**:

Variable $x : T$. introduce una variable de tipo T.
Axiom $x : T$. introduce en el contexto el axioma T cuya prueba es x.

```
Variable K:Set.
Variable eqK: K->K->Prop.

Axiom eqK_refl: (reflexive K eqK).
```

En Coq si R es una relación en el conjunto X, $(reflexive\ X\ R) = (x : X)(R\ x\ x)$ predica el carácter reflexivo de R. La axiomatización anterior en Coq implementa un término eqK_refl que representa una prueba del carácter reflexivo de la relación eqK definida sobre K.

Otras órdenes análogas a los anteriores son **Hipothesis**, **Parameter**, etc. Se puede ver su funcionamiento en [18].

2.3.2. Definición

Las definiciones consisten en dar un nombre a un objeto construido a partir de los objetos del entorno en curso.

Definition $x : T := t$. define la constante x de tipo T como t. Siempre es posible reemplazar la constante x por su definición t.

```
Definition term:= (list nat).
```

El comando **Local** es análogo al anterior y suele utilizarse para variables locales.

Mediante la orden **Check** podemos comprobar el tipo de los nuevos objetos (todas las frases en Coq se terminan por un punto).

```
Coq < Check eqK_refl.
eqK_refl
     : (reflexive K eqK)

Coq < Check term.
term
     : Set
```

2.3.3. Teoremas

Al igual que en la terminología de matemáticas, el sistema Coq utiliza la notación de lemas y teoremas. Y nos permite utilizar las herramientas (las tácticas) del sistema para encontrar pruebas.

Lemma $x : T.$ y **Theorem** $x : T.$ declaran x como un "candidato" a prueba de un teorema T.

```
Lemma plusK_comp_l: (k,k':K)(eqK k k')->(y:K)(eqK (plusK y k) (plusK y k')).
```

Los resultados locales, es decir los lemas que sólo interesan en una sección, utilizan la sintaxis de **Remark** y **Fact**.

Debido a que utilizaremos continuamente el modo interactivo de Coq (toplevel) para probar teoremas y lemas, vamos a hacer una presentación general de éste. Las hipótesis aparecen encima de una linea discontinua de doble trazo y la propiedad a probar (llamada meta) debajo. Si hay varias metas aparecen debajo de la inicial. Sólo son visibles permanentemente las hipótesis de la meta en curso, las demás no son visibles, salvo que las necesite el usuario (comandos **Show** o **Focus**). Mostramos un ejemplo donde quedan dos metas por probar:

```
2 subgoals

  n : nat
  H : (lt (0) n)
  ============================
   (S (pred (1)))=(1)

subgoal 2 is:
 (m:nat)(le (1) m)->(S (pred m))=m->(S (pred (S m)))=(S m)

< Show 2.
subgoal 2 is:

  n : nat
  H : (lt 0 n)
  ============================
   (m:nat)(le (S 0) m)->(S (pred m))=m->(S (pred (S m)))=(S m)
```

2.3.4. Modularidad

Coq permite una forma de modularidad agrupando las definiciones de objetos en el seno de secciones. Las secciones definen un mecanismo de bloques parecido al de la mayoría de los lenguajes de programación, permitiendo la declaración y definición de variables locales y controlando su alcance.

En **Coq**, las secciones tienen un nombre, y las órdenes de principio y fin de sección son respectivamente *Section nombre* y *End nombre*. Cuando se termina una sección, Coq generaliza (o abstrae) los objetos que se han definido vis a vis con los parámetros de la sección.

2.4. Tipos inductivos

Los matemáticos, lógicos e informáticos utilizan a menudo las definiciones inductivas; son herramientas esenciales para definir la sintaxis de los lenguajes formales. En **Coq**, existe un medio de introducir dichas definiciones sin recurrir a la axiomatización. Como la teoría subyacente a la definiciones inductivas [44,

97, 3, 14] es compleja, preferimos dar una presentación más informal, basándonos en ejemplos prácticos.

2.4.1. Tipos de datos

La sintaxis de las definiciones de tipos de datos en **Coq** se hace de manera similar a la manera de definir tipos concretos de CAML. Para definir un tipo inductivo, se da una lista de constructores con su tipo. Así, el ejemplo más inmediato de un tipo inductivo es la definición del conjunto de los números naturales:

```
Inductive nat : Set :=  O : nat | S : nat->nat.
```

Este tipo inductivo tiene por nombre **nat** y posee dos constructores **O** (la constante cero) de tipo **nat** y **S** (la función sucesor) de tipo $nat \rightarrow nat$. Se utiliza la orden **Inductive** para definir un tipo inductivo.

La principal ventaja de los tipos inductivos, con respecto a los tipos concretos de CAML, es que Coq asocia automaticamente a cada tipo inductivo "un principio de inducción" para el tipo de las proposiciones, y un "principio de recursión" para el tipo de las especificaciones y para el tipo **Type**.

Los principios de inducción y recursión para **nat** son:

```
nat_rect
     : (P:(nat->Type))(P (0))->((n:nat)(P n)->(P (S n)))->(n:nat)(P n).

nat_ind
     : (P:(nat->Prop))(P (0))->((n:nat)(P n)->(P (S n)))->(n:nat)(P n).

nat_rec
     : (P:(nat->Set))(P (0))->((n:nat)(P n)->(P (S n)))->(n:nat)(P n).
```

La única diferencia entre los tres principios reside en el tipo del predicado P.

Interpretamos el tipo *nat_ind*: la expresión anterior se lee " *cualquiera que sea la propiedad P sobre los naturales (P es una función de nat en Prop), se prueba $(n : nat)(P\ n)$ (en matemáticas: $\forall n.\ P\ (n)$) obteniendo un objeto de tipo $(P\ 0)$ y un objeto de tipo $((n : nat)(P\ n) \rightarrow (P\ (S\ n)))$ (en matemáticas: $\forall n.\ P\ (n) \Rightarrow P(n+1)$)*": se reconoce aquí el principio de demostración por inducción. El hecho de que Coq haya generado a partir de la definición de los números naturales un objeto que corresponde a la técnica de prueba por inducción sobre los enteros es muy interesante, pues tenemos "gratuitamente" un método para probar propiedades dependiendo de un entero.

Además de los constructores, una definición inductiva hace intervenir *Cases*, un destructor cuyo tipo es una proposición testigo de la minimalidad del punto fijo[7]. Esto tiene dos consecuencias:

[7]En matemáticas, muchos objetos se definen como el menor punto fijo de una función creciente y continua. En Coq, el medio para designar tales tipos son los tipos inductivos.

- Aparece la noción de representación canónica: los términos cerrados de éste tipo se forman por medio de los constructores. Por ejemplo definiendo el tipo inductivo booleano:

```
Inductive bool  : Set :=  true : bool | false : bool.
```

el destructor permite hacer razonamientos por casos sobre el valor de un booleano. Por ejemplo, nos permite definir la conjunción sobre los booleanos:

```
Definition andbool: bool->bool->bool :=
   [x,y:bool]<bool>Cases x of
      (* true *)   y
      (* false *)  false
      end.
```

- El destructor permite a la vez definir objetos y hacer demostraciones por recurrencia o por casos. En este caso mediante el destructor se puede definir la adición sobre los naturales, gracias a **Fixpoint**, que es un operador de recurrencia estructural:

```
Fixpoint plus [n:nat] : nat->nat :=
     [m:nat]Cases n of
       (* 0 *)       0 => m
       (* (S p) *) | (S p) => (S (plus p m))
            end.
```

Nota: El destructor se denota por **Cases**, da un término para cada constructor, los cuales deben estar en el mismo orden que los constructores.

También se pueden definir tipos polimórficos, como las listas de elementos de un tipo dado A:

```
Inductive list [A : Set]  : Set :=
     nil : (list A) | cons : A->(list A)->(list A).
```

2.4.2. Predicados inductivos

Gracias a las definiciones inductivas, se pueden definir predicados sobre los objetos que manipulamos (de manera análoga a Prolog). Así el conjunto de los números pares se puede definir como el conjunto S más pequeño de naturales conteniendo al 0, y tal que para todo número $n \in S$, $n+2$ también pertenece a S:

```
Inductive par: nat->Prop :=
     0_par : (par 0)
   | SS_par: (n:nat)(par n)->(par (S (S n))).
```

Los dos constructores de *par* son la traducción al lenguaje Coq de los axiomas matemáticos:

$$\begin{cases} 0 \text{ es par}, \\ \forall n.\ n\ par \Rightarrow (S\ (S\ n))\ es\ par \end{cases}$$

Con esta definición un objeto de tipo $(par\ n)$ nos asegura la existencia de un entero k tal que $n = (S\ (S\ k))$.

2.5. Desarrollo de la prueba: Tácticas

Como ya hemos dicho anteriormente el sistema Coq es un asistente de prueba: su papel no es "encontrar" pruebas de manera automática, sino la de construir un término a partir de una especificación, es decir un tipo; en este sentido, el usuario de Coq dispone de un cierto número de herramientas, llamadas *tácticas*. La tácticas son la traducción directa de las reglas de formación de los términos del cálculo de construcciones, o en otros casos la implementación de ciertas propiedades pertenecientes a esta teoría. Se emplean en un escenario que se denomina "*goal directed*".

Una táctica es una función que, aplicada a una submeta $\Gamma \vdash^? T$ una de dos:

- O tiene éxito y produce las dos informaciones siguientes:
  - Una sucesión finita de nuevas submetas $\Gamma_1 \vdash^? T_1, \ldots \Gamma_i \vdash^? T_i$
  - Una función que permite, a partir de eventuales soluciones $t_1, \ldots, t_i$ de las nuevas submetas, construir una solución para $\Gamma \vdash^? T$.
- O fracasa y deja la submeta igual.

Una táctica permite reemplazar una meta por una sucesión de submetas, de las cuales se espera sean más fáciles de resolver. Cuando después de la aplicación de una sucesión de tácticas llegamos a una meta que no se sabe resolver, es posible volver atrás, los pasos que queramos, mediante el comando **Undo**. También se puede volver al comienzo de la prueba con la orden **Restart**; y también abandonar la prueba ejecutando **Abort**.

Una prueba se termina cuando el conjunto de metas a resolver es vacio. Esta situación la señala el sistema con **Subtree proved!**. Para guardar la prueba se utiliza **Save**.

Omitiremos dar la lista completa de tácticas; simplemente citaremos la aplicación general de las que se van a necesitar a lo largo de la formalización posterior; otras se comentarán durante el transcurso de las pruebas. Se puede ver más información sobre las tácticas en [73, 18].

Tácticas de introducción

- **Intro**: aplicada a una meta de la forma $(x : N)\ M$ (o $N \rightarrow M$), introduce la hipótesis $x : N$ en el contexto local y cambia la meta por **M**. Esta táctica

corresponde a la regla de deducción natural $\frac{\Gamma[P] \vdash Q}{\Gamma \vdash P \to Q}$, donde Γ denota un conjunto de proposiciones que son implicitamente las hipótesis (contexto local de hipótesis).

- **Generalize**: introduce en la meta una abstracción; tiene el efecto inverso de la táctica **Intro**.
- **Clear**: borra hipótesis en el contexto local de la meta en curso. Es de mucha utilidad en pruebas largas donde se acumulan hipótesis inútiles.
- **Cut**: introduce, en el contexto de hipótesis, resultados que a su vez tendrán que probarse en otra submeta.

Tácticas de resolución

- **Apply**: reduce la meta a probar otras más elementales por la aplicación de un axioma o un resultado ya probado. Se aplica a una hipótesis del contexto de prueba, o a una constante del entorno. Su sintaxis es (*Apply t*); funciona si los tipos de cabeza de t coinciden con la meta en curso, dando error en otro caso. Esta táctica permite aplicar *modus ponens*: si tenemos un teorema de la forma $A \Rightarrow B$ y queremos probar B, aplicamos este teorema para reducir la meta en curso a la prueba de A. Resaltar que Coq utiliza la unificación para evitar tener que especificar todos los argumentos cuando se aplica esta táctica. Algunas variantes son **LApply** y **EApply**, puede verse su funcionamiento en [73].
- **Assumption**: busca en el contexto local un hipótesis idéntica a la meta.
- **Split**: transforma la meta $R \wedge S$ en dos metas R y S; esta táctica corresponde a la regla de deducción natural $\frac{\Gamma \vdash R; \Gamma \vdash S}{\Gamma \vdash R \wedge S}$.
- **Left**: transforma la meta $R \vee S$ en R.
- **Right**: transforma la meta $R \vee S$ en S.
- **Exists**: es la táctica usual para utilizar en una meta formada por cuantificación; se puede aplicar sobre todas las definiciones inductivas con un sólo constructor. Debemos dar un argumento testigo de la aplicación del constructor.

Tácticas de eliminación

- **Elim H**: aplica el principio de eliminación, que consiste en aplicar los esquemas de inducción anteriormente citados, asociado a la definición inductiva de **H** en la meta en curso (obviamente **H** debe ser un término definido inductivamente). En realidad, muchas veces la utilización de **Elim** puede interpretarse como una llamada a la táctica **Apply**, simplemente más cómoda. Se utilizarán a lo largo de nuestro desarrollo algunas variantes como **ElimType**.

- **Cases m**: reemplaza todas las apariciones de m en la meta, por los diferentes casos posibles que puede tomar un elemento de este tipo. Es una táctica más "primitiva" que **Elim**.

- **Induction n**: introduce n en el contexto y aplica el principio de eliminación asociado al tipo de n. Su comportamiento es parecido al de **Elim**.

- **Discriminate**: permite concluir una prueba a partir de una igualdad (=) que resulta falsa en alguna de las hipótesis de la meta.

- **Injection**: utiliza que los constructores de un tipo inductivo se asemejan a funciones inyectivas. Por ejemplo, de una igualdad de la forma $(c\ x_1 \ldots x_k) = (c\ y_1 \ldots y_k)$, esta táctica deduce las siguientes igualdades $x_1 = y_1, \ldots, x_k = y_k$.

- **Inversion**: Sirve para "extraer información" de los tipos inductivos. Esta táctica inspecciona los constructores del tipo inductivo al que se aplica, intenta aplicarlos y, en otro caso, los muestra para ser resueltos interactivamente por el usuario. Se utiliza constantemente a lo largo de nuestro desarrollo, así como sus variantes **Simple Inversion** e **Inversion clear**.

- **Destruct H**: Es equivalente a la composición de tácticas (también llamada tacticales) *Intros Until H*; *Case H*.

Tácticas de conversión

- **Simpl**: aplica las reglas de $\beta\iota$−reducción expandiendo las constantes que sean transparentes.

- **Unfold nom**: aplica la regla de δ−reducción a la meta, reemplazando todas las apariciones de *nom* por su definición. Otra táctica de funcionamiento análogo a esta es **Red**.

- **Pattern term**: El argumento *term* debe ser un subtérmino libre de la meta en curso. **Pattern** ejecuta la β-expansión de la meta T; reemplazando todas las "ocurrencias" de *term* en T por una nueva variable y abstrayendo dicha variable.

Tácticas de reescritura

Son tácticas especializadas en la igualdad del sistema.

- **Rewrite H**: si H es una hipótesis del tipo $(x_1 : T_1; \ldots; x_n : T_n)a = b$, aplicada la táctica a una meta M, genera una nueva submeta reemplazando todas las apariciones de a en M por b.

Otras tácticas de reescritura son: **Replace, Reflexivity, Symmetry** y **Transitivity.**

Un ejemplo de prueba

Mostramos el uso de las tácticas en Coq con un ejemplo simple. Se define el término (*paridad n*) un predicado *paridad* como un tipo inductivo con dos constructores, llamados respectivamente *par* e *impar*, según que n sea múltiplo de dos o no:

```
Inductive paridad: nat->Prop:=
    par:     (n:nat)(paridad (mult (S (S 0)) n))
  | impar:   (n:nat)(paridad (S (mult (S (S 0)) n))).
```

Queremos definir una función que a un entero n le asocie un objeto que tenga tipo (*paridad n*). Si se sabe construir una función que asocie tal objeto a cada n, tendremos probado que (*paridad n*) está definido para todo n; es decir "existe una inyección de *nat* en *paridad*".

```
Coq < Lemma paridad_nat: (n:nat)(paridad n).
1 subgoal

  ============================
   (n:nat)(paridad n)
```

Coq responde con la descripción de la meta: y tenemos que probar la "fórmula" $(n : nat)(paridad\ n)$. Disponemos de un método para probar propiedades sobre los enteros, el dado por el tipo *nat_ind* (pág 20). Además, como ya hemos indicado, *nat_ind* es generado de manera automática a partir de la definición inductiva de los enteros. Se puede aplicar directamente con la ayuda de la táctica **Induction:**

```
paridad_nat < Induction n.
2 subgoals

  n : nat
  ============================
   (paridad 0)

subgoal 2 is:
 (n0:nat)(paridad n0)->(paridad (S n0))
```

Nota: (*Induction n*) tiene el mismo efecto que aplicar (*Intro n*; *Elim n*), es decir, introducir n de tipo *nat* en el contexto[8] y después aplicar el principio *nat_ind* mediante la táctica **Elim**.

Vemos que (*Induction n*) aplica literalmente el principio de inducción. Ahora tenemos que probar dos metas que se corresponden con la propiedad en el caso cero y la validez de la propiedad en el caso del sucesor. Técnicamente, construimos una prueba de (*paridad n*) aplicando el término de tipo *nat_ind*

[8] Se trata de un contexto "local": el contexto global incluye todas las definiciones y pruebas conocidas por Coq hasta el momento de realizar la prueba.

a los argumentos cuyos tipos son $(paridad\ 0)$ y $(n0 : nat)(paridad\ n0) \rightarrow (paridad\ (S\ n0))$.

Caso base: $(paridad\ 0)$ se prueba reemplazando 0 por $(mult\ (S\ (S\ 0))\ 0)$ (para obtener un término que se corresponda con el constructor *par*).

```
paridad_nat < Replace 0 with (mult (S (S 0)) 0); [Apply par | Auto].
1 subgoal

  n : nat
  ============================
   (n0:nat)(paridad n0)->(paridad (S n0))
```

Utilizamos los "constructores de tácticas" ; y [|] para componer y distribuir, respectivamente, la prueba. En este caso, la táctica *Replace ... with* genera dos submetas, correspondientes a la meta inicial en la cual se ha efectuado el reemplazamiento y la meta que consiste en probar la igualdad entre los dos términos que intervienen en el reemplazamiento.

La primera submeta se resuelve con la ayuda del constructor *par*; se utiliza la táctica **Apply**. Tenemos que probar $(paridad\ (mult\ (S\ (S\ 0))\ 0))$ y disponemos de un objeto de tipo $(n : nat)(paridad\ (mult\ (S\ (S\ 0))\ n))$ (el constructor *par*), así vemos que es suficiente aplicar *par* sustituyendo n por 0. En este caso se puede comprobar, como hemos dicho anteriormente, que **Apply** utiliza la unificación "adivinando" la sustitución adecuada para obtener el tipo esperado.

La segunda meta expresa una propiedad elemental de la multiplicación de enteros que se resuelve automáticamente con la táctica **Auto**.

Paso inductivo: Tenemos que probar la segunda meta generada anteriormente por la táctica **Induction**:

```
1 subgoal

  n : nat
  ============================
   (n0:nat)(paridad n0)->(paridad (S n0))
```

La n del contexto no interviene en la meta (el sistema reescribe automáticamente la n en la conclusión); se puede eliminar mediante la táctica **Clear** que, aplica la regla de *debilitamiento* del contexto.

```
paridad_nat < Clear n.
1 subgoal

  ============================
   (n:nat)(paridad n)->(paridad (S n))
```

Introducimos la hipótesis en el contexto mediante la táctica **Intros**, una variante de **Intro**.

```
paridad_nat < Intros.
1 subgoal
```

```
  n : nat
  H : (paridad n)
  ============================
   (paridad (S n))
```

Tenemos en el contexto dos objetos: un natural n, y una prueba H (una hipótesis) de $(paridad\ n)$. Utilizamos H para probar $(paridad\ (S\ n))$ mediante el razonamiento por casos: si H está construido con el constructor *par*, utilizaremos el constructor *impar* para probar la meta, y reciprocamente.

```
paridad_nat < Case H.
2 subgoals

  n : nat
  H : (paridad n)
  ============================
   (n0:nat)(paridad (S (mult (S (S 0)) n0)))

subgoal 2 is:
 (n0:nat)(paridad (S (S (mult (S (S 0)) n0))))
```

En este caso vemos que Coq, en el análisis de casos reemplaza n por su valor en el seno de la meta. Las dos submetas se prueban fácilmente, utilizando los constructores *impar* para la primera y *par* para la segunda:

```
paridad_nat < Intro; Apply impar.
1 subgoal

  n : nat
  H : (paridad n)
  ============================
   (n0:nat)(paridad (S (S (mult (S (S 0)) n0))))


paridad_nat < Intro; Replace (S (S (mult (S (S 0)) n0))) with
                (mult (S (S 0)) (S n0)); [Apply par | Simpl; Auto].
Subtree proved!
```

Hemos terminado la prueba, en otras palabras; hemos construido un término de tipo $(n : nat)(paridad\ n)$. Podemos guardarla, y asociar así el término que acabamos de construir al tipo en cuestión.

```
paridad_nat < Save.
Induction n.
Replace 0 with (mult (S (S 0)) 0); [ Apply par | Auto ].
Clear n.
Intros.
Case H.
Intro; Apply impar.
Intro; Replace (S (S (mult (S (S 0)) n0)))
 with (mult (S (S 0)) (S n0)); [ Apply par | Simpl; Auto ].
paridad_nat is defined
```

Coq nos permite mediante el comando **Print** ver la estructura del término construido.

```
Coq < Print paridad_nat.
paridad_nat =
[n:nat]
 (nat_ind [n0:nat](paridad n0)
   (eq_ind nat (mult (S (S 0)) 0) [n0:nat](paridad n0) (par 0) 0
     (refl_equal nat 0))
   [n0:nat; H:(paridad n0)]
    <[n1:nat](paridad (S n1))>
      Cases H of
        (par n1) => (impar n1)
      | (impar n1) =>
         (eq_ind nat (mult (S (S 0)) (S n1)) [n2:nat](paridad n2)
           (par (S n1)) (S (S (mult (S (S 0)) n1)))
           (f_equal nat nat S (plus n1 (S (plus n1 0)))
             (S (plus n1 (plus n1 0)))
             (sym_eq nat (S (plus n1 (plus n1 0)))
               (plus n1 (S (plus n1 0))) (plus_n_Sm n1 (plus n1 0)))))
      end n)
     : (n:nat)(paridad n)
```

En esta definición se pueden ver las diversas étapas de nuestro desarrollo (algunos trozos de prueba como *eq_ind* y *plus_n_Sm* los generan las llamada a **Auto**). No hay duda, como ya hemos indicado, de lo engorroso de su lectura.

En Coq la igualdad (proposicional) está definida como la relación reflexiva más pequeña:

```
Inductive eq [A: Set; x : A]: A->Prop:=  refl_equal: x=x.
```

La notación = es una abreviatura sintáctica de $(eq\ A\ x\ x)$ para el tipo inductivo *eq* (igualdad de Leibniz). La igualdad entre términos (convertibilidad) se manipula en Coq con la ayuda de la propiedad *eq_ind*, aunque el usuario lo hace directamente utilizando diversas tácticas como **Replace** y **Rewrite**. Examinando el tipo de *eq_ind* podemos comprender el funcionamiento de éstas tácticas.

```
Coq < Check eq_ind.
eq_ind
     : (A:Set; x:A; P:(A->Prop))(P x)->(y:A)x=y->(P y)
```

La idea es la de reemplazar la prueba de una meta de la forma $(P\ y)$ por la prueba de $(P\ x)$, con la condición de construir una prueba de $x = y$.

Para terminar esta breve presentación mencionar **CtCoq**, ver [17], que es un interface de ayuda al desarrollo de la prueba, basado en la idea de *proof by pointing* [19].

Capítulo 3

Términos

La primera parte de este capítulo está dedicada a definir y formalizar en Coq la noción de término (esta noción no está universalmente aceptada), que será clave para la posterior definición de polinomio en varias variables. A continuación definimos el producto de términos y probamos que con dicha operación es un **monoide conmutativo**. Formalizamos un orden estricto sobre dicho monoide; acabamos dando la relación de divisibilidad de términos y el cálculo del mínimo común múltiplo de dos términos que serán básicos, respectivamente, para la definición de **reducción** de polinomios y el cálculo de los **S-polinomios**.

3.1. Definiciones y notaciones básicas

Comenzamos introduciendo la noción término. La definición usual de término es la siguiente:

Definición 3.1.1. *Un* **término (power products)** *en* $x_1, \ldots, x_n$ *es un producto de la forma:*

$$x_1^{\alpha_1} x_2^{\alpha_2} \ldots x_n^{\alpha_n}$$

en donde los exponentes $\alpha_1, \ldots, \alpha_n$ *son números enteros no negativos.*

Se puede simplificar esta notación tomando $\alpha = (\alpha_1, \ldots, \alpha_n)$, es decir una n-upla de enteros no negativos. Así tendríamos:

$$x^\alpha := x_1^{\alpha_1} x_2^{\alpha_2} \ldots x_n^{\alpha_n}$$

Cuando $\alpha = (0, \ldots, 0)$, denotamos $x^\alpha = 1$.

Aprovechando esta notación formalizamos en **Coq** la definición de término como una lista de naturales; para ello utilizamos la definición inductiva de natural **nat** así como la definición de listas **list**, comentadas en la sección anterior (páginas 20 y 21, respectivamente), y ambas cargadas por defecto una vez que abrimos una sesión en **Coq**.

Traducción en **Coq**:

```
Definition term:= (list nat).
```

Antes de describir el álgebra de los términos, introducimos la longitud de un término que será el número de variables que posee. A lo largo de nuestro desarrollo trabajaremos normalmente con términos de n variables; es por ello que implementamos en **Coq** los términos de n variables *full* y para ello utilizamos otra función ya cargada por defecto en el sistema, **length**.

Expresión en **Coq**:

```
Fixpoint length [l:(list nat)] : nat := Cases l of
       nil => (0)
     | (cons _ m) => (S (length m))
      end.
```

Nota: Cuando se define un objeto en Coq mediante **Cases**, la idea de base para el destructor es tener un elemento m de tipo inductivo I y verificar una propiedad $(P\ m)$, que, en general, depende de m. Para esto, es suficiente verificar la propiedad para cada $m = (c_i\ u_1, \ldots, u_{pi})$ de cada constructor c_i de I.

Traducción en **Coq**:

```
Definition full:= [n:nat][t:term]((length  t) = n).
```

La notación que utilizaremos para el conjunto de los términos de **n** variables, es decir de longitud **n**, es

$$T^n := \{x_1^{\alpha_1} x_2^{\alpha_2} \ldots x_n^{\alpha_n} | \alpha_i \in N, i = 1, \ldots, n\}$$

Empleando las inclusiones obvias, $T^0 \subset T^1 \subset T^2 \subset T^3 \subset \ldots$, se tiene el conjunto de los términos $T := \bigcup_n T^n$.

Para la **igualdad de términos** utilizaremos la igualdad del sistema (=), es decir, la igualdad polimórfica estándar de Leibniz definida en el módulo *Logic_Type*, ya comentada en el capítulo anterior. La ventaja que tiene utilizar esta igualdad es que podemos utilizar directamente las tácticas de reescritura del sistema referidas a dicha igualdad, como **Rewrite, Replace, Reflexivity, Symmetry, ...**, (ver [73]).

3.2. Producto de términos

Antes de introducir la multiplicación de términos damos algunos resultados, relativos al número de variables de los términos, necesarios para simplificar demostraciones posteriores.

Uno de estos resultados es que *si el término* $t = (\alpha_1, \ldots, \alpha_n)$ *tiene n variables, entonces el término* $t' = (\alpha_2, \ldots, \alpha_n)$ *tiene* $(n-1)$ *variables y, por supuesto, el término* $t'' = (\alpha_1, \ldots, \alpha_{n+1})$ *tiene* $(n+1)$ *variables, donde* α_{n+1} *es cualquier entero no negativo.*

Prueba: Hacemos explícita la definición de *full* mediante la orden **Unfold**, reescribimos n y por último aplicamos β-reducción sobre **length** mediante la

táctica **Simpl** obteniendo, como meta, fáciles igualdades aritméticas que el sistema resuelve automáticamente.

Traducción en **Coq**:

```
Lemma full_S: (n,n1:nat)(t:term)(full n (cons n1 t))->(full (pred n) t).

Unfold full; Intros.
Rewrite <- H; Simpl; Auto.

Lemma full_pred1: (n:nat)(t:term)(full n t)->(n1:nat)(full (S n) (cons n1 t)).

Unfold full; Intros.
Rewrite <- H; Simpl; Auto.
□
```

Identificamos el concepto "término 1" mediante un predicado, al estilo de **Prolog**, y se construye el término 1 de n variables, $x_1^0 x_2^0 \ldots x_n^0$, con la función *n_term_0*.

Nota: Definimos el predicado *null_term* mediante el comando **Inductive** de forma similar a la manera de hacerlo en **Caml Light**. Para definir la función *n_term_0* sobre los naturales utilizamos **Fixpoint**[1]. La descripción de la función recursiva, mediante este operador, se organiza siguiendo la estructura del tipo inductivo, en este caso **nat**. Como tiene dos constructores en el tipo inductivo, tenemos dos casos; el segundo constructor tiene un argumento en el mismo tipo inductivo, pudiéndose hacer llamadas recursivas sobre dicho argumento. Como veremos más adelante la funciones recursivas pueden tener más de un argumento.

Traducción en **Coq**:

```
Inductive null_term: term -> Prop:= null_term_nil: (null_term (nil ?))
           | null_term_cons: (t:term)(null_term t)->(null_term (cons 0 t)).


Fixpoint n_term_0 [n:nat]: term:= Cases n of
      0     => (nil ?)
    | (S n) => (cons 0 (n_term_0 n))
   end.
```

Los lemas siguientes comprueban que $x_1^0 x_2^0 \ldots x_n^0$ *es un término* 1, *que tiene n variables y que el único término* 1 *de n variables es precisamente* $x_1^0 x_2^0 \ldots x_n^0$.
Prueba: El primer resultado se demuestra por inducción sobre n (*Induction n*) pues de la definición de *n_term_0* obtenemos como metas los constructores de *null_term*. El segundo se hace asimismo por inducción sobre n, ya que de la definición de *full* obtenemos dos igualdades triviales. En el tercero se comienza aplicando recurrencia sobre la definición de *null_term*; así obtenemos dos metas correspondientes a los constructores de *null_term*, resueltas sustituyendo las definiciones de *full* por su valor.

Traducción en **Coq**:

```
Lemma null_term_term_0: (n:nat)(null_term (n_term_0 n)).
```

[1]Operador de recurrencia estructural

```
Induction n; Intros; Simpl.
Apply null_term_nil.
Apply null_term_cons; Trivial.


Theorem full_0 : (n:nat)(full n (n_term_0 n)).

Induction n; Intros; Simpl.
Unfold full; Simpl; Trivial.
Unfold full; Simpl.
Replace (length (n_term_0 n0)) with n0; Trivial.


Lemma nterm_null: (t:term)(null_term t)->(n:nat)(full n t)->t=(n_term_0 n).

Induction 1; Intros.
Inversion_clear H0; Simpl; Trivial.
Inversion H2; Simpl.
Pattern 1 t0.
Rewrite -> (H1 (length t0)); Auto.
□
```

Definimos el producto de términos, como el término que se obtiene al sumar los exponentes correspondientes a cada variable.

Nota: Para facilitar las pruebas formalizamos la definición sobre términos genéricos en cuanto al número de variables, más adelante, cuando sea necesario, fijaremos este número de variables con el predicado *full*.

Traducción en **Coq**:

```
Fixpoint term_mult [t1,t2:term]: term:= Cases t1 t2 of
        nil x  => x
      | x nil  => x
      | (cons n1 t1') (cons n2 t2')  =>
              (cons (plus n1 n2) (term_mult t1' t2'))
      end.
```

A continuación probamos propiedades algebraicas sobre el producto de términos, así como lemas básicos que necesitaremos en otras secciones.

Teorema 3.2.1. *El producto de términos es conmutativo.*

$$\forall t, t' \in T;\ t.t' = t'.t$$

Prueba: Se hace por inducción sobre los términos y aplicando resultados aritméticos simples.

Traducción en **Coq**:

```
Theorem term_mult_conmt: (t,t':term)((term_mult t t') = (term_mult t' t)).
□
```

Teorema 3.2.2. *El producto de términos es asociativo.*

$$\forall t, t_1, t_2 \in T;\ (t.t_1).t_2 = t.(t_1.t_2)$$

Prueba: Análoga a la hecha para la conmutatividad de la multiplicación de términos.

Traducción en **Coq**:

```
Theorem term_mult_assoc: (t,t1,t2:term)
   ((term_mult (term_mult t t1) t2) = (term_mult t (term_mult t1 t2))).
□
```

Teorema 3.2.3. *El producto de términos es compatible con la igualdad.*

$$\forall t, t_1, t_2 \in T;\ t_1 = t_2 \Rightarrow t_1.t = t_2.t$$

Prueba: Dados los axiomas de la igualdad de Leibnitz (ver [73]) disponibles en el sistema, el resultado se obtiene mediante reescritura.

Traducción en **Coq**:

```
Lemma term_mult_r: (t,t1,t2:term)(t1 = t2)->
                     ((term_mult t1 t) = (term_mult t2 t)).
□
```

Teorema 3.2.4.

$$\forall t, t_1, t_2 \in T;\ (t.t_1).t_2 = t_1.(t.t_2)$$

Prueba: Utilizando dos veces la reescritura (**Rewrite**), los lemas de conmutatividad y asociatividad que acabamos de demostrar, producen el resultado.

$$(t.t_1).t_2 = (t_1.t).t_2 = t_1.(t.t_2)$$

Nota: También se podría utilizar en esta prueba la inducción estructural sobre términos; la primera forma de demostración tiene la ventaja de que en caso de cambiar la estructura de los términos esta prueba sigue siendo válida, mientras que en el segundo caso habría que rehacerla.

Traducción en **Coq**:

```
Lemma term_mult_perm: (t,t1,t2:term)
   ((term_mult (term_mult t t1) t2) = (term_mult t1 (term_mult t t2))).

Intros.
Replace (term_mult (term_mult t t1) t2) with
   (term_mult (term_mult t1 t) t2); Auto.
□
```

Teorema 3.2.5. *El término $x_1^0 x_2^0 \ldots x_n^0$ es el elemento neutro de la multiplicación de términos de n variables.*

$$\forall t \in T^n;\ t.(x_1^0 x_2^0 \ldots x_n^0) = t$$

Prueba: Se realiza por inducción sobre el término t, aplicando la β-reducción en cada uno de los casos a la definición de *term_mult*. En la segunda submeta, debemos aplicar la hipótesis de inducción, utilizar la hipótesis de que t tiene n variables, además de resultados sobre el tipo **nat**.

Traducción en **Coq**:

```
Theorem term_mult_full_nulo: (t:term)(n:nat)(full n t)->
   ((term_mult t (n_term_0 n)) = t).

Induction t; Intros.
Inversion H; Auto.
Inversion H0; Simpl.
Replace (plus a 0) with a; Auto.
Replace (term_mult l (n_term_0 (length l))) with l; Auto.
Symmetry;Apply H; Auto.
□
```

Así hemos comprobado que el conjunto T^n con la operación producto es un monoide conmutativo.

Un resultado técnico, muy útil para simplificar pruebas de lemas posteriores, es que *si multiplicamos dos términos de n variables el resultado es otro término de n variables.*

Prueba: La prueba utiliza recurrencia estructural sobre el primer argumento t_1 y, en cada uno de los dos casos, de nuevo recurrencia sobre el segundo argumento t_2; las submetas generadas se resuelven utilizando las tácticas de reescritura.

Traducción en **Coq**:

```
Lemma term_mult_full: (t1:term)(t2:term)(n:nat)(full n t1)->(full n t2)->
      (full n (term_mult t1 t2)).

Unfold full.
Induction t1; Simpl; Intros.
Elim H0.
Case t2; Simpl; Auto.
Generalize H1; Case t2; Simpl; Intros; Trivial.
Rewrite -> (H l0 (length l0)); Trivial.
Cut (S (length l))=(S (length l0)).
Auto.
Rewrite -> H2; Auto.
□
```

3.3. Decidibilidad y algunas propiedades

Los resultados siguientes son elementales, pero el sistema no puede deducirlos automáticamente y serán necesarios en pruebas posteriores. Muchos de estos lemas se pueden declarar como **indicaciones** para **Coq** mediante el comando **Hints**; es decir, cuando el usuario lo pida mediante **Auto**[2], el sistema intentará resolver la meta en curso aplicando estos lemas. Otra táctica de automatización es **Tauto**, (ver [73]).

Esta herramienta es indispensable en un sistema de prueba automática debido a que libera al usuario de largas búsquedas que pueden hacer tediosas las pruebas y hacer perder el hilo del razonamiento seguido. Por ejemplo, si se ve que la meta en curso puede ser resuelta aplicando la conmutatividad del producto de términos, no es necesario buscar en el entorno el nombre exacto del

[2]Para que un teorema sea utilizado por Auto, es necesario que todas las variables cuantificadas universalmente de este teorema esten presentes en la cabeza del teorema.

teorema, el sistema lo hace automáticamente. Evidentemente, no todos los teoremas pueden darse como **indicaciones**, pues la búsqueda por parte del sistema podría alargarse en demasía.

```
Lemma eqterm_cons1: (t1,t2:term)(n1:nat)(t1 = t2)->
        ((cons n1 t1) = (cons n1 t2)).

Intros; Rewrite -> H; Auto.
Save.
Hint Resolve eqterm_cons1.


Lemma eqexp_cons: (t:term)(n1,n2:nat)(n1 = n2)->((cons n1 t) = (cons n2 t)).

Intros; Rewrite -> H; Auto.
Save.
Hint Resolve eqexp_cons.
□
```

Nota: En la biblioteca *Polylist.v* de Coq se prueba la decidibilidad de la igualdad de listas sobre todo $A : Set$, denotada por *list_eq_dec*. No obstante por razones de autocontenido y legibilidad damos una prueba de dicha decidibilidad directamente a partir de nuestra formalización.

Teorema 3.3.1. *La igualdad de términos es decidible.*

$$\forall t_1, t_2 \in T;\ \{(t_1 = t_2)\} \vee \{(t_1 \neq t_2)\}$$

Prueba: El objetivo de la demostración es la decidibilidad de la igualdad de términos. La prueba es simple debido a que *term* es de tipo inductivo. Es suficiente utilizar la recurrencia sobre los dos argumentos t_1 y t_2; obtenemos cuatro metas que demostramos mediante la discriminación de los diferentes constructores de los términos, salvo en la última meta:

```
{(cons a u)=(cons b v)}+{~((cons a u)=(cons b v))}
```

en la cual necesitamos utilizar la decidibilidad de la relación de igualdad en enteros **eq_nat_dec**, ya probada en las bibliotecas de **Coq**.

Traducción en **Coq**:

```
Lemma eq_tm_dec : (t1,t2:term){t1=t2}+{~(t1=t2)}.

Induction t1.
Induction t2.
Left; Trivial.
Intros a t22 HR; Right; Discriminate.
Intros a u hru ; Induction t2.
Right; Discriminate.
Intros b v hrv; Elim (eq_nat_dec a b); Intros He.
Elim (hru v); Intros Ht.
Left; Rewrite He; Rewrite Ht; Auto.
Right; Unfold not; Intros HH; Apply Ht; Injection HH; Trivial.
Right; Unfold not; Intros HH; Apply He; Injection HH; Trivial.
□
```

Nota: Este es el primer resultado calculatorio, es decir, del cual se puede extraer un programa de la prueba realizada, que aparece en nuestro desarrollo.

La prueba de un lema con contenido calculatorio tiene la misma estructura que el algoritmo que realiza ese lema.

El programa extraído, en **OCaml** y sin optimización "a mano", correspondiente al algoritmo seguido en la demostración es:

```
type 'a list =
  | Nil
  | Cons of 'a * 'a list

type nat =
  | O
  | S of nat

let rec eq_nat_dec n m =
  match n with
    | O -> (match m with
              | O -> true
              | S nO -> false)
    | S nO -> (match m with
                 | O -> false
                 | S n1 -> eq_nat_dec nO n1)

let rec eq_tm_dec l t2 =
  match l with
    | Nil -> (match t2 with
                | Nil -> true
                | Cons (a, l0) -> false)
    | Cons (a, l0) ->
        (match t2 with
           | Nil -> false
           | Cons (a0, l1) ->
               (match eq_nat_dec a a0 with
                  | true -> eq_tm_dec l0 l1
                  | false -> false))
```

Teorema 3.3.2. *Los elementos de T^n son simplificables para el producto.*

$$\forall t, t', t_0 \in T^n;\ t_0.t = t_0.t' \Rightarrow t = t'$$

Prueba: Esta demostración se hace por inducción estructural sobre los términos (mediante las órdenes **Induction** y **Destruct**). La primera meta obtenida es trivial, la resuelve el sistema por la reflexividad de la igualdad. La segunda y tercera meta se resuelven mediante la introducción de la negación, utilizando la táctica **Inversion** sobre las hipótesis referidas al número de variables (*full*); dado que entonces en el contexto de hipótesis aparece **False**, se sigue la validez de la conclusión. En la última meta hacemos uso de la hipótesis de recursión y de propiedades aritméticas; también se utiliza la táctica **Injection**[3], esto es, que los constructores de tipos inductivos son funciones inyectivas. Dicho de otro modo, si c es un constructor de un tipo inductivo, y $(c\ t_1)$ y $(c\ t_2)$ son dos objetos iguales, lo son también t_1 y t_2.

Traducción en **Coq**:

[3]Esta táctica nos muestra que los tipos inductivos gozan de propiedades muy fuertes.

```
Lemma term_mult_can_full: (t,t',t0:term)(n:nat)(full n t)->(full n t')->
   (term_mult t0 t)=(term_mult t0 t')-> t=t'.

Induction t; Destruct t'; Intros; Auto.
Inversion H; Inversion H0.
Rewrite <- H3 in H2; Inversion H2.
Inversion H1; Inversion H0.
Rewrite <- H4 in H3; Inversion H3.
Generalize H2;Elim t0; Intros; Auto.
Simpl in H4;Injection H4; Intros.
Replace a with n.
Replace l0 with l; Trivial.
Apply H with t':=l0 t0:=l1 n:=(pred n0); Trivial.
Inversion H0; Auto.
Inversion H1; Auto.
Apply simpl_plus_l with n:=a0; Auto.
□
```

Teorema 3.3.3.

$$\forall t, t' \in T^n;\ \forall t_0 \in T;\ t \neq t' \Rightarrow t_0.t \neq t_0.t'$$

Prueba: Se demuestra utilizando el teorema anterior *term_mult_can_full*, pues es su contrarrecíproco.

Traducción en **Coq**:

```
Lemma term_mult_n_eq: (t,t',t0:term)(n:nat)(full n t)->(full n t')->
   (~(t=t'))->(~((term_mult t0 t)=(term_mult t0 t'))).
□
```

3.4. Orden sobre términos

En esta sección, presentamos los órdenes más conocidos y utilizados sobre términos y formalizamos en **Coq** el que se va a utilizar en nuestro desarrollo. Para más detalles sobre estos órdenes, se puede consultar [45, 54, 76, 88, 15, 5].

Hay muchas maneras de ordenar T, pero para nuestro desarrollo este orden debe verificar ciertas propiedades, que veremos a continuación.

Definición 3.4.1. *Un orden* **estricto** $<_T$ *sobre* T *es una relación binaria que verifica las propiedades:*

- *Transitiva:*

$$\forall t_1, t_2, t_3 \in T;\ (t_1 <_T t_2 \wedge t_2 <_T t_3) \Rightarrow t_1 <_T t_3$$

- *Irreflexiva:*

$$\forall t \in T;\ \neg(t <_T t)$$

Definición 3.4.2. *Un orden* $<_T$ *sobre* T *se llama* **total** *si:*

$$\forall t_1, t_2 \in T; (t_1 <_T t_2) \vee (t_2 <_T t_1) \vee (t_1 = t_2)$$

Definición 3.4.3. *Un orden $<_T$ sobre T se dice que es* **bien fundado** *(well-founded) si no existe una cadena infinita decreciente de términos, es decir:*

$$\{\forall t_k, t_j \in T;\ (k < j \Rightarrow t_j <_T t_k)\} \Rightarrow \exists k_1 \in N;\ (\forall j \in N;\ (j > k_1 \Rightarrow t_{k_1} = t_j))$$

Definición 3.4.4. *Entendemos por orden* **admisible**, *un orden $<_T$ sobre T que verifica las dos propiedades siguientes:*

- *Todo término 1 es minimal para el orden:*

$$\forall t \in T; t \neq (x_1^0 x_2^0 \ldots x_n^0) \Rightarrow (x_1^0 x_2^0 \ldots x_n^0) <_T t$$

- *Propiedad de monotonía: el orden es compatible con el producto de términos:*

$$\forall t, t_1, t_2 \in T;\ (t_1 <_T t_2 \Rightarrow (t_1.t) <_T (t_2.t))$$

A cualquier orden que satisfaga la segunda condición, pero no la primera, se le denomina orden **semiadmisible**.

Hay muchos ejemplos de órdenes admisibles, pero los más comunes son: el orden lexicográfico, el orden lexicográfico graduado y orden lexicográfico graduado inverso.

Normalmente a los órdenes admisibles suele llamárseles, en la literatura matemática, **órdenes de términos** (term orderings), como haremos nosotros de ahora en adelante.

Aunque implementaremos y trabajaremos con el orden lexicográfico, describimos a continuación los tres órdenes de términos antes citados.

Definición 3.4.5. *Se define el* **orden lexicográfico** $<_L$ *sobre T, eligiendo primero un orden sobre las variables $\ldots <_v x_n <_v x_{n-1} <_v \cdots <_v x_2 <_v x_1$. Entonces para*

$$\alpha = (\alpha_1, \ldots, \alpha_n), \beta = (\beta_1, \ldots, \beta_n) \in N^n$$

se dirá,

$$x^\alpha <_L x^\beta :\overset{def}{\Longleftrightarrow} \left\{ \begin{array}{l} \textit{las primeras, de izquierda a derecha, coordenadas diferentes,} \\ \alpha_i \textit{ y } \beta_i, \textit{ de } \alpha \textit{ y } \beta, \textit{ verifican que } \alpha_i < \beta_i. \end{array} \right.$$

Es decir:

$$x^\alpha <_L x^\beta :\overset{def}{\Longleftrightarrow} \exists l \leq n,\ \alpha_l < \beta_l \wedge (\forall k < l,\ \alpha_k = \beta_k)$$

Traducción en **Coq**:

```
Inductive Ttm : term -> term -> Prop :=
  Ttm_car : (n1,n2:nat)(u1,u2:term)(lt n1 n2)->(length u1)=(length u2)->
            (Ttm (cons n1 u1)(cons n2 u2))
| Ttm_cdr :  (n:nat)(u1,u2:term)(Ttm u1 u2)->(Ttm (cons n u1)(cons n u2)).
```

Nota: En la formalización del orden *Ttm* comparamos únicamente términos del mismo número de variables, en otro caso existiria una cadena de términos decreciente infinita.

$$\begin{array}{cccc} 1 & & & \\ 0 & 1 & & \\ 0 & 0 & 1 & \\ 0 & 0 & 0 & 1 \\ & \ldots\ldots & & \end{array}$$

Para el orden lexicográfico, x_1^n siempre es mayor que x_2^m, para todo $n, m \in N$.
En el caso de cuatro variables T^4 con $(z < y < x < w)$, tenemos:

$$\begin{array}{llllllllllllll} 1 & <_L & z & <_L & z^2 & <_L & \ldots & & & & & & & \\ & <_L & y & <_L & y.z & <_L & \ldots & <_L & y^2 & \ldots & & & & \\ & <_L & x & <_L & x.z & <_L & \ldots & <_L & x.y & <_L & \ldots & x^2 & \ldots & \\ & <_L & w & <_L & w.z & <_L & \ldots & <_L & w.y & <_L & \ldots & w.x & <_L & \ldots & w^2 \\ & <_L & \ldots & & & & & & & & & & & \end{array}$$

En ejemplos de un número pequeño de variables, como el anterior, utilizamos normalmente letras w, x, y o z en vez de variables subindicadas y suponemos el orden alfabético de las variables. Es importante resaltar que es necesario especificar el orden de las variables. Existen tantos órdenes lexicográficos, como formas de ordenar las variables; en el caso general de n variables, son $n!$.

Trabajaremos con el orden lexicográfico pues nuestro objetivo es utilizar Bases de Gröbner, y en su cálculo este orden tiene ventajas, (como puede verse en [4, 45]), con respecto a otros órdenes de términos.

Definición 3.4.6. *El* **orden graduado lexicográfico** $<_{gL}$ *sobre* T^n, *una vez fijado un orden sobre las variables* $x_n <_v x_{n-1} <_v \ldots <_v x_2 <_v x_1$, *se define de la forma siguiente: Para*

$$\alpha = (\alpha_1, \ldots, \alpha_n), \beta = (\beta_1, \ldots, \beta_n) \in N^n$$

diremos:

$$x^\alpha <_{gL} x^\beta :\overset{def}{\Longleftrightarrow} \begin{cases} \sum_{i=1}^n \alpha_i < \sum_{i=1}^n \beta_i \\ o \\ \sum_{i=1}^n \alpha_i = \sum_{i=1}^n \beta_i \ y \ x^\alpha <_L x^\beta \end{cases}$$

En este orden, primero se ordena por el grado total (suma de los exponentes de las variables) y, en caso de igualdad, se recurre al orden lexicográfico.

Por ejemplo:

$$1 < y < x < y^2 < x.y < x^2 < y^3 < x.y^2 < x^2.y < x^3 < \ldots$$

Otro orden de términos, algo menos intuitivo, es el orden graduado lexicográfico inverso. Aunque este orden es menos utilizado, se ha comprobado que para algunos tipos de computacion es más eficiente que los anteriores.

Definición 3.4.7. *Para definir el* **orden graduado lexicográfico inverso** $<_{gLi}$ *sobre* T^n, *se elige un orden sobre las variables* $x_n <_v x_{n-1} <_v \ldots <_v x_2 <_v x_1$, *y para*

$$\alpha = (\alpha_1, \ldots, \alpha_n), \beta = (\beta_1, \ldots, \beta_n) \in N^n$$

se dirá

$$x^{\alpha} <_{gLi} x^{\beta} :\overset{def}{\Longleftrightarrow} \begin{cases} \sum_{i=1}^{n} \alpha_i < \sum_{i=1}^{n} \beta_i \\ o \\ \sum_{i=1}^{n} \alpha_i = \sum_{i=1}^{n} \beta_i \text{ y las primeras coordenadas,} \\ \text{de izquierda a derecha, diferentes, } \alpha_i \text{ y } \beta_i, \text{ de } \alpha \text{ y } \beta, \\ \text{verifican que } \alpha_i > \beta_i \end{cases}$$

Vemos que al igual que el orden graduado lexicográfico, primero se ordena por el grado total, pero la manera de romper la igualdad es diferente. En el caso de dos variables ambos órdenes son iguales; sin embargo, esto ya no ocurre en el caso de tres variables, pues:

$$x.y^3 <_{gL} x^2.y.z$$

y, sin embargo:

$$x^2.y.z <_{gLi} x.y^3$$

Cada uno de los órdenes de términos descritos tienen diferentes propiedades y la elección del orden a utilizar dependerá del problema que se quiera tratar.

Implementado el orden lexicográfico, vamos a ver a continuación que este orden verifica las propiedades deseadas para un orden de términos. En primer lugar demostramos un lema auxiliar sobre el número de variables, que hemos llamado la longitud del término, dada la forma en que se han implementado estos.

Teorema 3.4.1. *El orden lexicográfico es contractivo, es decir*

$$\forall t, t' \in T;\ t <_L t' \Rightarrow \textit{la longitud de } t \textit{ es menor o igual que la de } t'$$

Prueba: Se hace por recursión sobre la definición inductiva del orden lexicográfico Ttm. Obtenemos dos metas, correspondientes a los dos constructores de Ttm, y en cada una de ellas, mediante el comando **Simpl** hacemos la β-reducción de la definición **length**, obteniendo como submetas dos resultados aritméticos cuyas pruebas aparecen en las bibliotecas cargadas al inicio de la sesión.

Traducción en **Coq**:

```
Lemma Ttm_shorter: (x,y:term)(Ttm x y)->(le (length x) (length y)).

Induction 1; Intros; Simpl.
Elim H1; Auto.
Auto with v62.
□
```

Antes de comenzar con la demostración de que el orden lexicográfico tiene la propiedad de bien fundado, damos un metateorema, sugerido por B. Barras [12], que permite razonar por recurrencia sobre la longitud de una lista.

Metateorema de la Inducción Completa para listas.

Sea P una propiedad sobre listas, se verifica:

$$\{\forall n : nat\ [\forall w : list,\ ((long(w)) < n) \Rightarrow P(w)] \quad \wedge$$
$$(\forall v : list\ ((long(v)) = n) \Rightarrow P(v))\} \quad \Rightarrow \quad \forall a : list,\ P(a)$$

Prueba: Mediante el comando **Cut** introducimos la hipótesis de que para cada natural n, las listas de longitud menor que n verifican la propiedad P. Tenemos entonces dos metas, la primera, que es el resultado global, se sigue trivialmente de la hipótesis añadida, pues toda lista w es de longitud menor que $(long(w)+1)$. La segunda, que es dicho resultado, se demuestra por inducción en la cota de la longitud de las listas. El caso de cota cero es inmediato por introducción de la negación $((v_0 : nat),\ v_0 < 0)$ en el contexto de hipótesis; el paso de inducción se sigue de las hipótesis de recurrencia. Al final queda por demostrar una propiedad aritmética sobre la relación $<$ (lt) de enteros, que se resuelve automáticamente mediante la táctica **Omega**[4], ver [105].

Traducción en **Coq**:

```
Lemma list_indlg: (A:Set)(P:(list A)->Prop)
   ((n:nat)((v:(list A))(lt (length v) n)->(P v))->
    (v:(list A))(length v)=n->(P v))->
       (v:(list A))(P v).

Intros A P H v.
Cut (n:nat; v:(list A))(lt (length v) n)->(P v).
Intros H0.
Apply H0 with (S (length v)); Auto.
Induction n.
Intros v0 H0.
Inversion H0.
Intros n0 H0 v0 H1.
Apply H with (length v0); Intros; Trivial.
Apply H0.
Omega.
```

□

Nota: Un resultado equivalente al anterior se puede obtener como consecuencia de las bibliotecas de Coq, en particular, del uso del módulo *wf_inverse_image* de la biblioteca *Wellfounded* cuyo autor es Bruno Barras. No obstante por coherencia con nuestra formalización dejamos la prueba anterior e incluimos en el apéndice la prueba de dicho resultado.

Es difícil dar una versión constructiva de la noción de relación bien fundada. La traducción que damos se debe a Gérard Huet [69] y está formalizada en las bibliotecas del sistema. Una relación binaria R sobre un conjunto A es bien fundada (artiniana) si todo subconjunto A' de A no vacío contiene un elemento minimal a_0, es decir un elemento que verifica $\neg(aRa_0)\ \forall a \in A$; se define en Coq utilizando la noción de accesibilidad.

[4] **Omega** es una biblioteca de Coq donde están probadas propiedades de las operaciones y de las relaciones en los enteros con la aritmética lineal de Presburger.

Definición 3.4.8. *El conjunto de los elementos de un tipo A accesibles para una relación R es el menor conjunto Y, que verifica:*

$$\forall x.(\forall y.\ yRx \Rightarrow y \in Y) \Rightarrow x \in Y$$

Este conjunto suele denotarse por Acc_R.

Expresión de **Coq**:

```
Inductive Acc [A:Set; R:A->A->Prop] : A->Prop :=
      Acc_intro : (x:A)((y:A)(R y x)->(Acc A R y))->(Acc A R x).
```

La relación R se dice que es bien fundada si todos los elementos de A son accesibles.

Expresión de **Coq**:

```
Definition well_founded := [A:Set][R:A->A->Prop](a:A)(Acc A R a).
```

Nota: Posteriormente, probamos que $<_L$ verifica las propiedades de orden estricto.

Teorema 3.4.2. *El orden lexicográfico $<_L$ sobre T, es bien fundado.*

Prueba: La demostración se realiza mediante recurrencia sobre la longitud de los términos, utilizando el metateorema *list_indlg* antes demostrado.

Para probar que $<_L$ es bien fundado es necesario probar que todo término es accesible; y para ello hace falta demostrar que sea cual sea la manera de "decrecer" este término, el término resultante también es accesible. En general obtenemos esta prueba por hipótesis de recurrencia, salvo en el caso base, donde la prueba se obtiene por introducción de la negación debido a que el término al cual se reduce es minimal. La primera etapa consiste en hacer razonamientos por recurrencia, hasta obtener hipótesis de recursión que abarquen todas las maneras de "decrecer" un término. Después sólo queda estudiar las diferentes posibilidades, que se resuelven aplicando las hipótesis de recursión obtenidas anteriormente.

En el orden lexicográfico hay tres cantidades que pueden decrecer: la longitud de un término, el valor del exponente de cabeza y el número máximo de pasos posibles para reducir la cola del término. El procedimiento que se sigue es:

1. Razonamos por recurrencia sobre la longitud del término. De este modo obtenemos una hipótesis de recursión que dice que todo término más corto es accesible.

2. Se demuestra, por introducción de la negación, el caso base del término de longitud cero.

3. Para un término del tipo $(cons\ n\ t)$, tenemos dos casos; que disminuya n (el exponente) y t tome cualquier valor pero conservando su longitud, o que decrezca t y no varie n. El primer caso se prueba basándose en el hecho de que n es accesible para $<$ (lt) sobre enteros, prueba existente en la biblioteca **Wf_nat**, mediante la táctica **ElimType**.

4. En el último caso sabemos que no se puede reducir infinitas veces debido a que todo elemento de longitud menor que $(cons\ n\ t)$ es accesible por Ttm basándonos en la hipótesis de recursión obtenida en el caso 1.

Traducción en **Coq**:

```
Lemma Ttm_wf : (well_founded ? Ttm).
```

Sustituimos *well_founded* por su valor.

```
Ttm_wf < Red; Intros.
1 subgoal

  a : term
  ============================
   (Acc term Ttm a)
```

1. Aplicamos la recursión sobre la longitud de a.

```
Ttm_wf < Elim a using list_indlg.
1 subgoal

  a : term
  ============================
   (n:nat)
    ((v:(list nat))(lt (length v) n)->(Acc term Ttm v))
     ->(v:(list nat))(length v)=n->(Acc term Ttm v)

Ttm_wf < Destruct v; Simpl; Intros.
2 subgoals

  a : term
  n : nat
  H : (v:(list nat))(lt (length v) n)->(Acc term Ttm v)
  v : (list nat)
  H0 : (0)=n
  ============================
   (Acc term Ttm (nil nat))

subgoal 2 is:
 (Acc term Ttm (cons n0 l))
```

2. Caso base, término nil.

```
Ttm_wf < Constructor; Intros.
2 subgoals

  a : term
  n : nat
  H : (v:(list nat))(lt (length v) n)->(Acc term Ttm v)
  v : (list nat)
  H0 : (0)=n
  y : term
  H1 : (Ttm y (nil nat))
  ============================
   (Acc term Ttm y)
```

```
subgoal 2 is:
 (Acc term Ttm (cons n0 l))

Ttm_wf < Inversion H1.
1 subgoal

  a : term
  n : nat
  H : (v:(list nat))(lt (length v) n)->(Acc term Ttm v)
  v : (list nat)
  n0 : nat
  l : (list nat)
  H0 : (S (length l))=n
  ============================
   (Acc term Ttm (cons n0 l))
```

3. **Paso inductivo, caso de** $(cons\ n0\ l)$**. Generalizamos para todo término** l_0 **más corto que** $(cons\ n0\ l)$.

```
Ttm_wf < Cut (lt (length l) n).
2 subgoals

  a : term
  n : nat
  H : (v:(list nat))(lt (length v) n)->(Acc term Ttm v)
  v : (list nat)
  n0 : nat
  l : (list nat)
  H0 : (S (length l))=n
  ============================
   (lt (length l) n)->(Acc term Ttm (cons n0 l))

subgoal 2 is:
 (lt (length l) n)

Ttm_wf < Generalize l.
2 subgoals

  a : term
  n : nat
  H : (v:(list nat))(lt (length v) n)->(Acc term Ttm v)
  v : (list nat)
  n0 : nat
  l : (list nat)
  H0 : (S (length l))=n
  ============================
   (l0:(list nat))(lt (length l0) n)->(Acc term Ttm (cons n0 l0))

subgoal 2 is:
 (lt (length l) n)

Ttm_wf < Clear H0 l.
2 subgoals

  a : term
  n : nat
```

```
  H : (v:(list nat))(lt (length v) n)->(Acc term Ttm v)
  v : (list nat)
  n0 : nat
  ============================
   (l:(list nat))(lt (length l) n)->(Acc term Ttm (cons n0 l))

subgoal 2 is:
 (lt (length l) n)
```

Razonamos por recurrencia basándonos en el hecho de que $n0$ es accesible para lt ($<$); con ello obtenemos una hipótesis de recursión que resuelve todos los casos en que disminuye el exponente $n0$.

```
Ttm_wf < ElimType (Acc ? lt n0); Intros.
3 subgoals

  a : term
  n : nat
  H : (v:(list nat))(lt (length v) n)->(Acc term Ttm v)
  v : (list nat)
  n0 : nat
  x : nat
  H0 : (y:nat)(lt y x)->(Acc nat lt y)
  H1 : (y:nat)
       (lt y x)
       ->(l:(list nat))(lt (length l) n)->
                    (Acc term Ttm (cons y l))
  l : (list nat)
  H2 : (lt (length l) n)
  ============================
   (Acc term Ttm (cons x l))

subgoal 2 is:
 (Acc nat lt n0)
subgoal 3 is:
 (lt (length l) n)

Ttm_wf < Generalize H2.
3 subgoals

  a : term
  n : nat
  H : (v:(list nat))(lt (length v) n)->(Acc term Ttm v)
  v : (list nat)
  n0 : nat
  x : nat
  H0 : (y:nat)(lt y x)->(Acc nat lt y)
  H1 : (y:nat)
       (lt y x)
       ->(l:(list nat))(lt (length l) n)->
                    (Acc term Ttm (cons y l))
  l : (list nat)
  H2 : (lt (length l) n)
  ============================
   (lt (length l) n)->(Acc term Ttm (cons x l))

subgoal 2 is:
```

```
 (Acc nat lt n0)
subgoal 3 is:
 (lt (length l) n)
```

4. En el caso de que decrezca el término l y no el exponente $n0$, hace falta una hipótesis de recursión.

```
Ttm_wf < ElimType (Acc ? Ttm l); Intros.
4 subgoals

  a : term
  n : nat
  H : (v:(list nat))(lt (length v) n)->(Acc term Ttm v)
  v : (list nat)
  n0 : nat
  x : nat
  H0 : (y:nat)(lt y x)->(Acc nat lt y)
  H1 : (y:nat)
       (lt y x)
       ->(l:(list nat))(lt (length l) n)->
                    (Acc term Ttm (cons y l))
  l : (list nat)
  H2 : (lt (length l) n)
  x0 : term
  H3 : (y:term)(Ttm y x0)->(Acc term Ttm y)
  H4 : (y:term)(Ttm y x0)->(lt (length y) n)->
                      (Acc term Ttm (cons x y))
  H5 : (lt (length x0) n)
  ============================
   (Acc term Ttm (cons x x0))

subgoal 2 is:
 (Acc term Ttm l)
subgoal 3 is:
 (Acc nat lt n0)
subgoal 4 is:
 (lt (length l) n)
```

Una vez obtenidas todas las hipótesis de recursión necesarias, sólo queda comprobar que, sea cual sea la manera de "decrecer" el término $(cons\ x\ x0)$, lo hace a un término accesible.

```
Ttm_wf < Constructor; Intros.
4 subgoals

  a : term
  n : nat
  H : (v:(list nat))(lt (length v) n)->(Acc term Ttm v)
  v : (list nat)
  n0 : nat
  x : nat
  H0 : (y:nat)(lt y x)->(Acc nat lt y)
  H1 : (y:nat)
       (lt y x)
       ->(l:(list nat))(lt (length l) n)->
                    (Acc term Ttm (cons y l))
```

```
  l : (list nat)
  H2 : (lt (length l) n)
  x0 : term
  H3 : (y:term)(Ttm y x0)->(Acc term Ttm y)
  H4 : (y:term)(Ttm y x0)->(lt (length y) n)->
                        (Acc term Ttm (cons x y))
  H5 : (lt (length x0) n)
  y : term
  H6 : (Ttm y (cons x x0))
  ============================
   (Acc term Ttm y)

subgoal 2 is:
 (Acc term Ttm l)
subgoal 3 is:
 (Acc nat lt n0)
subgoal 4 is:
 (lt (length l) n)

Ttm_wf < Inversion_clear H6.
5 subgoals

  a : term
  n : nat
  H : (v:(list nat))(lt (length v) n)->(Acc term Ttm v)
  v : (list nat)
  n0 : nat
  x : nat
  H0 : (y:nat)(lt y x)->(Acc nat lt y)
  H1 : (y:nat)
       (lt y x)
       ->(l:(list nat))(lt (length l) n)->(Acc term Ttm (cons y l))
  l : (list nat)
  H2 : (lt (length l) n)
  x0 : term
  H3 : (y:term)(Ttm y x0)->(Acc term Ttm y)
  H4 : (y:term)(Ttm y x0)->(lt (length y) n)->(Acc term Ttm (cons x y))
  H5 : (lt (length x0) n)
  y : term
  n1 : nat
  u1 : term
  H7 : (lt n1 x)
  H8 : (length u1)=(length x0)
  ============================
   (Acc term Ttm (cons n1 u1))

subgoal 2 is:
 (Acc term Ttm (cons x u1))
subgoal 3 is:
 (Acc term Ttm l)
subgoal 4 is:
 (Acc nat lt n0)
subgoal 5 is:
 (lt (length l) n)
```

En el caso de disminuir el exponente, utilizamos la segunda hipótesis de recursión.

```
Ttm_wf < Apply H1; Trivial.
5 subgoals

  a : term
  n : nat
  H : (v:(list nat))(lt (length v) n)->(Acc term Ttm v)
  v : (list nat)
  n0 : nat
  x : nat
  H0 : (y:nat)(lt y x)->(Acc nat lt y)
  H1 : (y:nat)
       (lt y x)
        ->(l:(list nat))(lt (length l) n)->
                    (Acc term Ttm (cons y l))
  l : (list nat)
  H2 : (lt (length l) n)
  x0 : term
  H3 : (y:term)(Ttm y x0)->(Acc term Ttm y)
  H4 : (y:term)(Ttm y x0)->(lt (length y) n)->
                       (Acc term Ttm (cons x y))
  H5 : (lt (length x0) n)
  y : term
  n1 : nat
  u1 : term
  H7 : (lt n1 x)
  H8 : (length u1)=(length x0)
  ============================
   (lt (length u1) n)

subgoal 2 is:
 (Acc term Ttm (cons x u1))
subgoal 3 is:
 (Acc term Ttm l)
subgoal 4 is:
 (Acc nat lt n0)
subgoal 5 is:
 (lt (length l) n)

Ttm_wf < Rewrite H8; Trivial.
4 subgoals

  a : term
  n : nat
  H : (v:(list nat))(lt (length v) n)->(Acc term Ttm v)
  v : (list nat)
  n0 : nat
  x : nat
  H0 : (y:nat)(lt y x)->(Acc nat lt y)
  H1 : (y:nat)
       (lt y x)
        ->(l:(list nat))(lt (length l) n)->
                    (Acc term Ttm (cons y l))
  l : (list nat)
  H2 : (lt (length l) n)
  x0 : term
  H3 : (y:term)(Ttm y x0)->(Acc term Ttm y)
  H4 : (y:term)(Ttm y x0)->(lt (length y) n)->
                       (Acc term Ttm (cons x y))
```

```
  H5 : (lt (length x0) n)
  y : term
  u1 : term
  H7 : (Ttm u1 x0)
  ============================
   (Acc term Ttm (cons x u1))

subgoal 2 is:
 (Acc term Ttm l)
subgoal 3 is:
 (Acc nat lt n0)
subgoal 4 is:
 (lt (length l) n)
```

En el caso en que se reduce el término $x0$, se utiliza la tercera hipótesis de recursión.

```
Ttm_wf < Apply H4; Trivial.
4 subgoals

  a : term
  n : nat
  H : (v:(list nat))(lt (length v) n)->(Acc term Ttm v)
  v : (list nat)
  n0 : nat
  x : nat
  H0 : (y:nat)(lt y x)->(Acc nat lt y)
  H1 : (y:nat)
       (lt y x)
        ->(l:(list nat))(lt (length l) n)->
                    (Acc term Ttm (cons y l))
  l : (list nat)
  H2 : (lt (length l) n)
  x0 : term
  H3 : (y:term)(Ttm y x0)->(Acc term Ttm y)
  H4 : (y:term)(Ttm y x0)->(lt (length y) n)->
                      (Acc term Ttm (cons x y))
  H5 : (lt (length x0) n)
  y : term
  u1 : term
  H7 : (Ttm u1 x0)
  ============================
   (lt (length u1) n)

subgoal 2 is:
 (Acc term Ttm l)
subgoal 3 is:
 (Acc nat lt n0)
subgoal 4 is:
 (lt (length l) n)

Ttm_wf < Apply le_lt_trans with (length x0); Trivial.
4 subgoals

  a : term
  n : nat
  H : (v:(list nat))(lt (length v) n)->(Acc term Ttm v)
```

```
  v : (list nat)
  n0 : nat
  x : nat
  H0 : (y:nat)(lt y x)->(Acc nat lt y)
  H1 : (y:nat)
       (lt y x)
       ->(l:(list nat))(lt (length l) n)->
                       (Acc term Ttm (cons y l))
  l : (list nat)
  H2 : (lt (length l) n)
  x0 : term
  H3 : (y:term)(Ttm y x0)->(Acc term Ttm y)
  H4 : (y:term)(Ttm y x0)->(lt (length y) n)->
                           (Acc term Ttm (cons x y))
  H5 : (lt (length x0) n)
  y : term
  u1 : term
  H7 : (Ttm u1 x0)
  ============================
   (le (length u1) (length x0))

subgoal 2 is:
 (Acc term Ttm l)
subgoal 3 is:
 (Acc nat lt n0)
subgoal 4 is:
 (lt (length l) n)
```

Esta meta se resuelve aplicando el carácter contractivo del orden lexicográfico (Teorema 1.1.9), pues tenemos en el contexto la hipótesis que $u_1 <_L x_0$.

```
Ttm_wf < Apply Ttm_shorter;Trivial.
3 subgoals

  a : term
  n : nat
  H : (v:(list nat))(lt (length v) n)->(Acc term Ttm v)
  v : (list nat)
  n0 : nat
  x : nat
  H0 : (y:nat)(lt y x)->(Acc nat lt y)
  H1 : (y:nat)
       (lt y x)
       ->(l:(list nat))(lt (length l) n)->
                       (Acc term Ttm (cons y l))
  l : (list nat)
  H2 : (lt (length l) n)
  ============================
   (Acc term Ttm l)

subgoal 2 is:
 (Acc nat lt n0)
subgoal 3 is:
 (lt (length l) n)
```

Probamos que el término l es accesible debido a que tiene menos

de n variables.

```
Ttm_wf < Apply H; Trivial.
2 subgoals

  a : term
  n : nat
  H : (v:(list nat))(lt (length v) n)->(Acc term Ttm v)
  v : (list nat)
  n0 : nat
  ============================
   (Acc nat lt n0)

subgoal 2 is:
 (lt (length l) n)
```

Sabemos que todo entero es accesible para lt (<).

```
Ttm_wf < Apply lt_wf.
1 subgoal

  a : term
  n : nat
  H : (v:(list nat))(lt (length v) n)->(Acc term Ttm v)
  v : (list nat)
  n0 : nat
  l : (list nat)
  H0 : (S (length l))=n
  ============================
   (lt (length l) n)


Elim H0; Auto.
Subtree proved!
```

Finalmente hemos probado el Lema; podemos registrarlo utilizando el comando Save o Qed. Coq nos da entonces la lista de tácticas que hemos aplicado, y nos dice que el lema está definido; a partir de este momento podemos utilizarlo en cualquiera otra prueba donde lo necesitemos.

```
Ttm_wf < Save.

Red; Intros.
Elim a using list_indlg.
Destruct v; Simpl; Intros.
Constructor; Intros.
Inversion H1.

Cut (lt (length l) n).
Generalize l.
Clear H0 l.
ElimType (Acc ? lt n0).
Intros.
Generalize H2.
ElimType (Acc ? Ttm l); Intros.
```

```
Constructor; Intros.
Inversion_clear H6.
Apply H1; Trivial.
Rewrite H8; Trivial.

Apply H4; Trivial.
Apply le_lt_trans with (length x0); Trivial.
Apply Ttm_shorter; Trivial.

Apply H; Trivial.

Apply lt_wf.

Elim H0; Auto.

Ttm_wf is defined
□
```

Teorema 3.4.3. *El orden lexicográfico $<_L$ sobre T^n, verifica la propiedad transitiva.*

$$\forall t_1, t_2, t_3 \in T^n;\ (t_1 <_L t_2 \wedge t_2 <_L t_3) \Rightarrow t_1 <_L t_3$$

Prueba: La hacemos por recurrencia estructural del orden lexicográfico lo que produce dos submetas correspondientes a los dos constructores de Ttm, introduciendo las hipótesis obtenidas en el contexto. En cada una de estas metas mediante la táctica **Inversion** sobre la hipótesis $(Ttm\ t_2\ t_3)$ obtenemos, a su vez, dos submetas.

Las submetas referidas al primer constructor de Ttm, es decir cuando los exponentes de cabeza son distintos, debemos utilizar la reescritura sobre la longitud de los términos, lemas relativos a longitud de términos anteriormente probados, así como la transitividad de $<$ (**lt**) sobre enteros. La meta generada por el segundo constructor Ttm_cdr se resuelve utilizando la hipótesis de inducción y lemas relativos a la longitud de los términos.

Traducción en **Coq**:

```
Lemma Ttm_trans: (t1,t2:term)(Ttm t1 t2)->(n:nat)(full n t1)->
   (full n t2)->(t3:term)(full n t3)->(Ttm t2 t3)->(Ttm t1 t3).

Induction 1; Intros.
Generalize H0 H4.
Inversion H5; Intros.
Rewrite H10 in H1.
Constructor 1; Trivial.
Apply lt_trans with n2; Auto.
Constructor 1; Trivial.
Transitivity (pred n).
Cut (full (pred n) u1).
Auto.
Apply full_S with n1:=n1; Auto.
Cut (full (pred n) u3).
Auto.
Apply full_S with n1:=n2; Auto.
Generalize H0 H4.
Inversion H5; Intros.
```

```
Generalize H8.
Constructor 1; Trivial.
Transitivity (length u2); Trivial.
Transitivity (pred n0).
Cut (full (pred n0) u1).
Auto.
Apply full_S with n1:=n1; Auto.
Cut (full (pred n0) u2).
Auto.
Apply full_S with n1:=n2; Auto.
Constructor 2.
Apply H1 with n:=(pred n0) t3:=u3; Trivial.
Apply full_S with n1:=n; Auto.
Apply full_S with n1:=n; Auto.
Apply full_S with n1:=n; Auto.
□
```

Teorema 3.4.4. *El orden lexicográfico $<_L$ sobre T^n, verifica la propiedad asimétrica.*

$$\forall t_1, t_2 \in T^n;\ \neg[(t_1 <_L t_2) \wedge (t_2 <_L t_1)]$$

Prueba: Explicitamos la negación mediante el comando **Red** e introducimos la conjución; una vez hecho esto aplicamos la definición recursiva del orden lexicográfico en $(Ttm\ t_1\ t_2)$ y cada una de las submetas obtenidas se resuelven por casos de manera análoga al lema de la transitividad, salvo en uno de ellos que se resuelve mediante la propiedad antisimétrica del orden (**lt**) de enteros, para ello se utiliza la táctica **Omega**.

Traducción en **Coq**:

```
Lemma Ttm_antisym : (t1,t2:term)(n:nat)(full n t1)->(full n t2)->
                                  ~((Ttm t1 t2)/\(Ttm t2 t1)).

Intros.
Red; Intros.
Elim H1; Intros.
Clear H1.
Generalize t1 t2 n H H0 H2 H3.
Clear H3 H2 H0 H n t2 t1.
Induction 3; Intros.
Inversion H4.
Omega.
Rewrite H8 in H1.
Omega.
Inversion H4.
Omega.
Apply H3; Trivial.
□
```

Teorema 3.4.5. *El orden lexicográfico $<_L$ sobre T verifica la propiedad irreflexiva.*

$$\forall t \in T;\ \neg(t <_L t)$$

Prueba: Se prueba utilizando la definición inductiva de *term*; mediante la táctica **Inversion** sobre la definición del orden lexicográfico estudiamos los casos

posibles, estos son resueltos por contradicción y utilizando que el orden sobre enteros (lt) verifica la propiedad irreflexiva.

Traducción en **Coq**:

```
Lemma Ttm_nonrefl: (t:term)~(Ttm t t).

Induction t; Intros.
Red; Intro.
Inversion H.
Red; Intro.
Inversion H0.
Omega.
Elim H; Trivial.
□
```

Una demostración alternativa de la propiedad anterior se puede encontrar en los apéndices. Esta demostración se basa en el hecho de que toda relación de orden bien fundado verifica la propiedad irreflexiva.

Teorema 3.4.6. *El orden lexicográfico $<_L$ sobre T^n es un orden estricto.*

$$\forall t_1, t_2 \in T^n;\ t_1 <_L t_2 \Rightarrow t_1 \neq t_2$$

Prueba: Es una consecuencia simple de la propiedad de no reflexibilidad del orden lexicográfico, aunque antes de aplicar el lema anterior deben reescribirse los términos t_1 y t_2.

Traducción en **Coq**:

```
Lemma Ttm_strict : (t1,t2:term)(n:nat)(full n t1)->(full n t2)->
                   (Ttm t1 t2)->~t1=t2.
□
```

Teorema 3.4.7. *El orden lexicográfico $<_L$ sobre T^n es un orden total.*

$$\forall t_1, t_2 \in T^n;\ \{(t_1 <_L t_2)\} \vee \{(t_2 <_L t_1)\} \vee \{(t_1 = t_2)\}$$

Prueba: Se realiza por casos sobre el valor de los términos t_1 y t_2. Los diversos casos se resuelven mediante los constructores de Ttm; las submetas de cada caso se obtienen utilizando que la longitud de los términos es la misma, mediante la decidibilidad del orden de enteros *lt_eq_lt_dec* y la hipótesis de recurrencia para términos de longitud $(n-1)$.

Como este lema es constructivo (resultado calculatorio), podemos extraer, de la prueba realizada, el programa.

Traducción en **Coq**:

```
Lemma Ttm_total_good: (t1,t2:term)(n:nat)
{(full n t1)->(full n t2)->(Ttm t1 t2)}+
  {(full n t1)->(full n t2)->(Ttm t2 t1)}+{(full n t1)->(full n t2)->t1=t2}.

Induction t1; Induction t2; Intros.
Right; Auto.
Left; Left; Intros.
Inversion H0.
Inversion H1.
```

```
Cut (length (nil nat))=(length (cons a l)); Intros.
Elim H4.
Inversion H4.
Transitivity n; Auto.
Left; Right; Intros.
Inversion H0.
Inversion H1.
Cut (length (nil nat))=(length (cons a l)); Intros.
Elim H4.
Inversion H4.
Transitivity n; Auto.
Elim (lt_eq_lt_dec a a0); Intros.
Elim a1; Intros; Clear a1.
Left.
Left; Intros.
Apply Ttm_car; Trivial.
Transitivity (pred n).
Inversion H1; Auto.
Inversion H2; Auto.
Elim H with l0 (pred n); Intros.
Elim a1; Intros; Clear a1.
Left.
Left; Intros.
Rewrite b.
Apply Ttm_cdr.
Apply a2.
Inversion H1; Auto.
Inversion H2; Auto.
Left; Right; Intros.
Rewrite b.
Apply Ttm_cdr.
Apply b0.
Inversion H1; Auto.
Inversion H2; Auto.
Right; Intros.
Rewrite b.
Replace l0 with l; Trivial.
Apply b0.
Inversion H1; Auto.
Inversion H2; Auto.
Left.
Right; Intros.
Apply Ttm_car; Trivial.
Transitivity (pred n).
Inversion H2; Auto.
Inversion H1; Auto.
□
```

El programa que se extrae, en **OCaml**, sin optimización "a mano" es:

```
type 'a list =
  | Nil
  | Cons of 'a * 'a list

type 'a sumor =
  | Inleft of 'a
  | Inright

type nat =
```

```
  | 0
  | S of nat

let pred = function
  | 0 -> 0
  | S u -> u

let rec lt_eq_lt_dec n m =
  match n with
    | 0 -> (match m with
              | 0 -> Inleft false
              | S n0 -> Inleft true)
    | S n0 ->
        (match m with
           | 0 -> Inright
           | S n1 -> lt_eq_lt_dec n0 n1)

let rec ttm_total_good l t2 n =
  match l with
    | Nil ->
        (match t2 with
           | Nil -> Inright
           | Cons (a, l0) -> Inleft true)
    | Cons (a, l0) ->
        (match t2 with
           | Nil -> Inleft false
           | Cons (a0, l1) ->
               (match lt_eq_lt_dec a a0 with
                  | Inleft x ->
                      (match x with
                         | true -> Inleft true
                         | false -> ttm_total_good l0 l1 (pred n))
                  | Inright -> Inleft false))
```

Por razones operativas es interesante tener el siguiente resultado que es consecuencia inmediata del anterior.

```
Lemma Ttm_total : (t1,t2:term)(n:nat)(full n t1)->(full n t2)->
                    {Ttm t1 t2}+{Ttm t2 t1}+{t1 = t2}.
```

□

Necesitamos que el orden lexicográfico sea un orden de términos admisible y una de las condiciones necesarias para ello es que tenga un primer elemento.

Teorema 3.4.8. *El término nulo $x_1^0 x_2^0 \ldots x_n^0$ es minimal para el orden lexicográfico $<_L$ sobre T^n.*

$$(x_1^0 x_2^0 \ldots x_n^0) <_L t,\ \forall t \in T^n, t \neq (x_1^0 x_2^0 \ldots x_n^0)$$

Prueba: En primer lugar se aplica inducción sobre el número de variables; el caso de 0 variables se prueba utilizando la identificación, por β-reducción, de $(n\_term\_O\ (0))$ con la lista $(nil\ nat)$. Para resolver el caso de $(n_0 + 1)$ variables hacemos recurrencia estructural sobre la definición de *term*, obteniendo dos nuevas submetas. La primera se resuelve por **Inversion** sobre la longitud de **nil**; para la segunda se necesita utilizar el lema **zerop**[5], que permite discriminar

[5] $(n : nat)\{n = (0)\} + \{(lt\ (0)\ n)\}$

si un natural es 0 o mayor que 0, lo que nos da dos nuevas submetas resueltas mediante los constructores del orden lexicográfico.

Traducción en **Coq**:

```
Lemma ord_adm1: (n:nat)(t:term)(full n t)->
                  (~t=(n_term_0 n))->(Ttm (n_term_0 n) t).

Induction n.
Induction t; Intros.
Simpl in H0.
Elim H0; Trivial.
Simpl.
Auto.
Inversion H0.
Induction t; Intros.
Inversion H0.
Simpl.
(Elim (zerop a); Intros).
Rewrite a0.
Apply Ttm_cdr.
(Apply H; Auto).
(Red; Intros).
Red in H2.
Apply H2.
Rewrite a0.
(Rewrite H3; Auto).
(Apply Ttm_car; Trivial).
(Cut (full n0 l); Intros; Auto).
Red in H3.
(Rewrite H3; Trivial).
□
```

Para que el orden lexicográfico sea **admisible**, además de la propiedad anterior necesitamos que sea compatible con el producto de términos.

Teorema 3.4.9. *El orden lexicográfico $<_L$ sobre T^n, verifica la propiedad de monotonía con respecto al producto por la derecha.*

$$\forall t, u, v \in T^n;\ (t <_L u \ \Rightarrow\ t.v <_L u.v)$$

Prueba: Se hace por recurrencia sobre cada uno de los términos de las hipótesis, las submetas generadas se resuelven aplicando los constructores, la definición de producto de términos y propiedades de los enteros relacionadas con la longitud de los términos, tal y como se ha hecho en lemas anteriores.

Traducción en **Coq**:

```
Lemma Ttm_a2: (t,u:term)(n:nat)(full n t)->(full n u)->(Ttm t u)->
      (v:term)(full n v)->(Ttm (term_mult t v) (term_mult u v)).

Induction t.
Induction u; Intros.
Cut ~(Ttm (nil nat) (nil nat)); Auto; Intros.
Elim H3; Auto.
Inversion H0.
Inversion H1.
Cut (length (nil nat))=(length (cons a l)); Intros.
```

```
Elim H6.
Inversion H6.
Transitivity n; Auto.
Induction u.
Intros.
Inversion H2.
Induction v; Auto; Intros.
Simpl.
Inversion H3.
Apply Ttm_car.
Omega.
Cut (full (pred n) (term_mult 1 l1)); Intros.
Cut (full (pred n) (term_mult l0 l1)); Intros.
Inversion H12.
Inversion H13.
Transitivity (pred n); Auto.
Cut (full (pred n) l0); Intros.
Cut (full (pred n) l1); Auto.
Inversion H5; Auto.
Inversion H2; Auto.
Cut (full (pred n) 1); Intros.
Cut (full (pred n) l1); Auto.
Inversion H5; Auto.
Inversion H1; Auto.
Apply Ttm_cdr.
Apply H with n:=(pred n); Trivial.
Inversion H1; Auto.
Inversion H2; Auto.
Inversion H5; Auto.
□
```

Teorema 3.4.10. *El orden lexicográfico $<_L$ sobre T, verifica la propiedad de monotonía con respecto al producto por la izquierda.*

$$\forall t, u, v \in T;\ (t <_L u \ \Rightarrow\ v.t <_L v.u)$$

Prueba: Trivial, utilizando la conmutatividad del producto de términos y la monotonía por la derecha.

Traducción en **Coq**:

```
Lemma mon_term_mult: (t,u:term)(n:nat)(full n t)->(full n u)->(Ttm t u)->
           (v:term)(full n v)->(Ttm (term_mult v t) (term_mult v u)).
□
```

3.5. Divisibilidad y cociente de términos

En este parágrafo se estudia la relación de divisibilidad en términos. A partir de esta definición, y sólo cuando dos términos son divisibles, se puede dar la definición de cociente de términos.

Definición 3.5.1. *Un término $t = x_1^{\alpha_1} x_2^{\alpha_2} \ldots x_n^{\alpha_n}$ es* **divisible** *por un término $u = x_1^{\beta_1} x_2^{\beta_2} \ldots x_n^{\beta_n}$ (denotado por $u|t$), o equivalentemente t es* **múltiplo** *de u, si*

$$(\forall i\ 1 \leq i \leq n)\ [\beta_i \leq \alpha_i]$$

Definimos la divisibilidad de términos en **Coq** como un predicado "a lo Prolog".

```
Inductive term_div: term->term->Prop :=
   term_div_null : (t:term)(t':term)(null_term t)->(term_div t t')
 | term_div_cons: (n1,n2:nat)(t,t':term)(le n1 n2)->(term_div t t')->
                            (term_div (cons n1 t) (cons n2 t')).
```

Teorema 3.5.1. *El único término que divide al término* 1 *es el* 1.

Prueba: Se utiliza la recursión sobre la definición inductiva de la divisibilidad de términos *term_div*. La primera submeta obtenida es una hipótesis; la segunda se resuelve, utilizando que el término t_2 es el 1, mediante la táctica **Inversion**.

Traducción en **Coq**:

```
Lemma null_div: (t1,t2:term)(term_div t1 t2)->(null_term t2)->(null_term t1).

Induction 1; Intros; Auto.
Generalize H0.
Inversion_clear H3; Intros.
Replace n1 with 0; Auto.
Omega.
□
```

Teorema 3.5.2. *La divisibilidad sobre* T*, verifica la propiedad transitiva.*

$$\forall t_1, t_2, t_3 \in T;\ (t_1|t_2) \wedge (t_2|t_3) \Rightarrow t_1|t_3$$

Prueba: Se hace por recursión estructural sobre la definición *term_div*; las submetas generadas se resuelven utilizando los distintos constructores de la divisibilidad y del término 1. Debemos utilizar, asimismo, la transitividad del orden (lt) de los enteros y el teorema anterior.

Traducción en **Coq**:

```
Lemma div_trans: (t1,t2:term)(term_div t1 t2)->(t3:term)(term_div t2 t3)->
                            (term_div t1 t3).

Induction 1; Intros; Auto.
Inversion H3; Intros.
Inversion H4.
Constructor 1.
Replace n1 with 0.
Constructor 2.
Apply null_div with t'; Auto.
Omega.
Constructor 2; EAuto.
Apply le_trans with n2; Auto.
□
```

Teorema 3.5.3. *La divisibilidad sobre* T*, verifica la propiedad reflexiva.*

$$\forall t \in T;\ (t|t)$$

Prueba: La prueba se deriva trivialmente de los constructores de *term_div*.

Traducción en **Coq**:

```
Lemma term_div_n_eq: (t:term)(term_div t t).
□
```

Teorema 3.5.4.

$$\forall t, t_1, t_2 \in T;\ (t_1|t_2) \Rightarrow t_1|(t.t_2)$$

Prueba: Se utiliza recurrencia sobre la divisibilidad; la primera submeta, una vez hecha la simplificación (β-reducción), coincide exactamente con el primer constructor de *term_div*. La segunda submeta se resuelve por casos según los valores del término t[6]. Nótese que debemos utilizar la hipótesis de recurrencia obtenida al comienzo de la prueba.

Traducción en **Coq**:

```
Lemma term_div_n_comp: (t1,t2:term)(term_div t1 t2)->
                         (t:term)(term_div t1 (term_mult t t2)).

Induction 1; Intros; Auto.
Case t0; Intros.
Simpl; Auto.
Simpl; Apply term_div_cons.
Replace (plus n n2) with (plus n2 n); Omega.
Apply H2.
□
```

Un caso particular del lema anterior, muy útil para simplificar pruebas posteriores, es el siguiente resultado.

Corolario 3.5.1.

$$\forall t_1, t_2 \in T;\ t_1|(t_1.t_2)$$

Prueba: Es consecuencia inmediata del lema anterior y de la conmutatividad del producto de términos.

Traducción en **Coq**:

```
Lemma term_mult_div : (t1,t2:term)(term_div t1 (term_mult t1 t2)).
□
```

Teorema 3.5.5. *La divisibilidad sobre T^n es decidible.*

$$\forall t_1, t_2 \in T^n;\ \{(t_1|t_2)\} \vee \{\neg\,(t_1|t_2)\}$$

Prueba: Por recurrencia sobre los términos t_1 y t_2 se obtienen cuatro casos; cuando uno de ellos es la lista de naturales vacía, como ocurre en los tres primeros casos, se resuelve fácilmente, utilizando los constructores del tipo término, así como, que la longitud de los términos es la misma.La submeta obtenida cuando los términos contienen el constructor *cons*, se prueba mediante la decidibilidad de enteros *lt_eq_lt_dec*, la hipótesis de recurrencia para términos de longitud $(n-1)$ y la discriminación de los constructores de término.

Traducción en **Coq**:

[6] El sistema lo renombra como t_0.

```
Lemma dec_term_div: (t1,t2:term)(n:nat)(full n t1)->(full n t2)->
                                  {(term_div t1 t2)} + {~(term_div t1 t2)}.

Induction t1.
Intros.
Left; Auto.
Induction t2; Intros.
Elim (eq_tm_dec (cons a l) (nil nat)); Intros.
Left.
Rewrite y; Auto.
Right.
Inversion H1.
Simpl in H2.
Inversion H0.
Simpl in H3.
Rewrite <- H2 in H3.
Cut ~(S (length l))=(0); Intros.
Elim H4; Auto.
Omega.
Cut (full (pred n) l); Intros.
Cut (full (pred n) l0); Intros.
Elim (lt_eq_lt_dec a a0); Intros.
Elim y; Intros; Clear y.
Elim H with l0 (pred n); Intros; Trivial.
Left.
Constructor 2; Auto.
Omega.
Right.
Red; Intros.
Inversion H5; Auto.
Inversion H6; Auto.
Elim H with l0 (pred n); Intros; Trivial.
Left.
Rewrite y0.
Constructor 2; Auto.
Right.
Red; Intros.
Rewrite y0 in H5.
Inversion H5; Auto.
Inversion H6; Auto.
Right.
Red; Intros.
Inversion H5.
Cut a=0; Intros.
Rewrite H9 in y.
Omega.
Inversion H6; Auto.
Omega.
Inversion H2; Auto.
Inversion H1; Auto.
□
```

Necesitamos el cociente de términos[7] para la formalización de la **reducción (división) de polinomios**. Este cociente sólo está definido en los casos en que el numerador es múltiplo del denominador.

[7] Al igual que la divisiblidad, lo formalizamos para términos cualesquiera. Cuando necesitemos trabajar con términos de n variables, le añadiremos el predicado *full*.

Definición 3.5.2. *Dados dos términos $t_1, t_2 \in T$, tal que $t_2|t_1$, se llamará cociente al término, denotado por $\frac{t_1}{t_2}$, que se obtiene restando al exponente de cada variable del término t_1 el exponente de la correspondiente variable del término t_2.*

Traducción en **Coq**:

```
Fixpoint div_term [t1,t2:term]: term:= Cases t1 t2 of
      nil x  => x
   |  x nil  => x
   |  (cons n1 t1') (cons n2 t2')  => (cons (minus n1 n2) (div_term t1' t2'))
   end.
```

Nótese que en la anterior formalización nunca se alcanzará el primer caso, pero por el análisis exhaustivo de casos que hace el comando **Fixpoint**, se necesita incluirlo; el tercer caso siempre se utilizará cuando $n_2 \leq n_1$.

Veamos ahora dos resultados, relativos a este cociente, muy útiles posteriormente. El primero dice que *el cociente de dos términos de n variables es otro término de n variables*, el segundo es un caso particular del primero para términos expresados en la forma $(cons\ n\ t)$.

Prueba: La demostración es análoga a la realizada para la multiplicación de términos, salvo el hecho que debemos hacer inducción sobre la longitud del término debido a que el cociente no es una función total; el segundo lema se prueba mediante el primero y resultados sobre la longitud de los términos de secciones anteriores.

Traducción en **Coq**:

```
Lemma term_div_full: (n:nat)(t1:term)(t2:term)(full n t1)->
                (full n t2)->(full n (div_term t1 t2)).


Lemma term_div_full_S:(n,n1,n2:nat)(t,t':term)(full n (cons n1 t))->
   (full n (cons n2 t'))->(full n (cons (minus n2 n1) (div_term t' t))).
```

□

Ahora estamos en condiciones de probar propiedades algebraicas sobre el cociente de términos. Los dos teoremas siguientes se demuestran de manera similar, por recurrencia sobre el término considerado y utilizando técnicas de reescritura.

Teorema 3.5.6. *El cociente de un término de n variables por sí mismo es el elemento neutro $(x_1^0 x_2^0 \ldots x_n^0)$ de la multiplicación de términos.*

$$\forall t \in T^n;\ \frac{t}{t} = (x_1^0 x_2^0 \ldots x_n^0)$$

Traducción en **Coq**:

```
Lemma term_div_eq:(t:term)(n:nat)(full n t)->((div_term t t)=(n_term_0 n)).
```

□

Teorema 3.5.7. *El cociente de un término de n variables por el elemento neutro $(x_1^0 x_2^0 \ldots x_n^0)$ es el propio término.*

$$\forall t \in T^n,\ \frac{t}{(x_1^0 x_2^0 \ldots x_n^0)} = t$$

Traducción en **Coq**:

```
Lemma term_div_full_nulo : (t:term)(n:nat)(full n t)->
                              ((div_term t (n_term_0 n)) = t).
```

□

Los cinco resultados siguientes muestran las relaciones entre el producto y el cociente de términos, necesarias en las pruebas de reducción (división) de polinomios.

Teorema 3.5.8.

$$\forall t, t_1, t_2 \in T^n,\ (t|t_1) \Rightarrow \frac{t_1}{t}.t_2 = \frac{t_1.t_2}{t}$$

Prueba: La prueba por recurrencia sobre la divisibilidad de términos genera dos metas. La primera se resuelve utilizando que el único término 1 de longitud n es $x_1^0.x_2^0 \ldots x_n^0$, el lema anterior *term_div_full_nulo* y que el producto de términos de longitud n es un término de longitud n. La segunda se hace por casos sobre los valores de término t_2, utilizando la hipótesis de recursión obtenida de la divisibilidad y resultados sobre los enteros.

Traducción en **Coq**:

```
Lemma mult_div_term1: (t,t1:term)(term_div t t1)->
   (n:nat) (full n t)->(full n t1)->(t2:term)(full n t2)->
      ((term_mult (div_term t1 t) t2) = (div_term (term_mult t1 t2) t)).

Induction 1.
Intros.
Rewrite -> (nterm_null t0 H0 n H1).
Rewrite -> term_div_full_nulo; Trivial.
Rewrite -> term_div_full_nulo; Auto.
Destruct t2; Simpl; Intros; Trivial.
Generalize H3 H4 H5 .
Case n; Intros.
Inversion H8.
Elim H2 with n3 1; Auto.
Replace (plus (minus n2 n1) n0) with (minus (plus n2 n0) n1).
Auto.
Omega.
```

□

Teorema 3.5.9.

$$\forall u_1, u_2 \in T^n,\ (u_1|u_2) \Rightarrow u_2 = u_1.\frac{u_2}{u_1}$$

Prueba: Se hace formalizando en **Coq** las siguientes igualdades, utilizando la táctica **Transitivity** y el resultado anterior:

$$u_1.\frac{u_2}{u_1} = \frac{u_1.u_2}{u_1} = \frac{u_1}{u_1}.u_2 = u_2$$

Traducción en **Coq**:

```
Lemma divis_im_divt: (u1,u2:term)(term_div u1 u2)->(n:nat)(full n u1)->
                (full n u2)->(u2=(term_mult u1 (div_term u2 u1))).

Intros.
Transitivity (term_mult (div_term u2 u1) u1); Auto.
Transitivity (div_term (term_mult u2 u1) u1).
Transitivity (term_mult (div_term u1 u1) u2).
Replace (div_term u1 u1) with (n_term_0 n).
Transitivity (term_mult u2 (n_term_0 n)); Auto.
Symmetry; Auto.
Symmetry; Auto.
Replace (term_mult u2 u1) with (term_mult u1 u2); Auto.
Apply mult_div_term1 with n; Auto.
Symmetry.
Apply mult_div_term1 with n; Auto.
□
```

El resultado $(\forall t, t_1, t_2 \in T;\ t_1 = t_2 \Rightarrow \frac{t_1}{t} = \frac{t_2}{t})$ es consecuencia inmediata de la igualdad de Leibnitz.

Traducción en **Coq**:

```
Lemma eg_denom_term: (t,t1,t2:term)(t1 = t2)->
□                       ((div_term t1 t)=(div_term t2 t)).
```

Teorema 3.5.10.

$$\forall t_1, t_2 \in T^n,\ \frac{t_2.t_1}{t_2} = t_1$$

Prueba: La demostración de este resultado y el siguiente se hacen utilizando la transitividad, mediante el comando **Transitivity**, de la igualdad de Leibnitz y los tres lemas anteriores.

$$\frac{t_2.t_1}{t_2} = \frac{t_2}{t_2}.t_1 = (x_1^0 x_2^0 \ldots x_n^0).t_1 = t_1$$

Traducción en **Coq**:

```
Lemma div_mult_mon_inv: (t1,t2:term)(n:nat)(full n t1)->(full n t2)->
                              (div_term (term_mult t2 t1) t2)=t1.

Intros.
Transitivity (term_mult (div_term t2 t2) t1).
Symmetry.
Apply mult_div_term1 with n; Auto.
Transitivity (term_mult (n_term_0 n) t1); Auto.
Transitivity (term_mult t1 (n_term_0 n)); Auto.
□
```

Teorema 3.5.11.

$$\forall t, t', t_1 \in T^n,\ (t'|t_1) \wedge (t_1|t) \Rightarrow \frac{t}{t'} = \frac{t_1}{t'}.\frac{t}{t_1}$$

Prueba: Se prueba formalizando en **Coq** las siguientes igualdades:

$$\frac{t}{t'} = \frac{t}{t'}.(x_1^0 x_2^0 \ldots x_n^0) = \frac{t}{t'}.\frac{t_1}{t_1} = \frac{t.\frac{t_1}{t_1}}{t'} = \frac{t_1.\frac{t}{t_1}}{t'} = \frac{t_1}{t'}.\frac{t}{t_1}$$

Traducción en **Coq**:

```
Lemma simpl_term: (t,t',t1:term)(n:nat)(full n t)->(full n t')->
      (full n t1)->(term_div t' t1)->(term_div t1 t)->
        (div_term t t')=(term_mult (div_term t1 t') (div_term t t1)).

Intros.
Transitivity (term_mult (div_term t t') (n_term_0 n)).
Symmetry; Auto.
Transitivity (term_mult (div_term t t') (div_term t1 t1)).
Transitivity (term_mult (n_term_0 n) (div_term t t')); Auto.
Transitivity (term_mult (div_term t1 t1) (div_term t t')); Auto.
Apply term_mult_r.
Symmetry; Auto.
Transitivity (div_term (term_mult t (div_term t1 t1)) t').
Apply mult_div_term1 with n; Auto.
Apply div_trans with t1; Trivial.
Transitivity (div_term (term_mult t1 (div_term t t1)) t').
Apply eg_denom_term.
Transitivity t.
Symmetry.
Transitivity (term_mult t (n_term_0 n)).
Symmetry; Auto.
Transitivity (term_mult (n_term_0 n) t); Auto.
Transitivity (term_mult (div_term t1 t1) t); Auto.
Apply term_mult_r.
Symmetry; Auto.
Apply divis_im_divt with n; Auto.
Symmetry.
Apply mult_div_term1 with n; Auto.
□
```

Ahora estamos ya en condiciones de demostrar la relación entre multiplicidad y orden lexicográfico con respecto a la divisibilidad de términos.

Teorema 3.5.12.

$$\forall t_1, t_2 \in T^n,\ (t_1|t_2) \Rightarrow \exists t_3 \in T^n, t_2 = t_1.t_3$$

Prueba: El término $\frac{t_2}{t_1}$, es de longitud n por serlo t_2 y t_1 y verifica la expresión algebraica pedida, como ya hemos probado anteriormente en el lema *divis_im_divt*.

Traducción en **Coq**:

```
Lemma ex_term_div_n: (t1,t2:term)(term_div t1 t2)->(n:nat)(full n t1)->
   (full n t2)->(Ex [t3:term]((full n t3) /\ (t2 = (term_mult t1 t3)))).
□
```

Teorema 3.5.13. *El orden lexicográfico es más débil que el orden que determina la relación de divisibilidad.*

$$\forall t_1, t_2 \in T^n,\ (t_1|t_2) \Rightarrow [(t_1 <_L t_2) \vee (t_1 = t_2)]$$

Prueba: Por casos, mediante la decidibilidad de términos, *eq_tm_dec*, respecto a t_1 y t_2. El caso $t_1 = t_2$ es trivial. Para el caso $t_1 \neq t_2$, utilizando la relación entre multiplicidad y divisibilidad, *ex_term_div_n*, obtenemos que $(\exists x \in T^n, t_2 = t_1.x)$. Por ser el orden lexicográfico admisible se verifica que

$x_1^0 x_2^0 \ldots x_n^0 \leq x$, probado en *ord_adm1*; y por la segunda condición de órdenes admisibles *mon_term_mult*, tenemos que $t_1 \leq t_1.x = t_2$.

Traducción en **Coq**:

```
Lemma comp_divis_lex: (t1,t2:term)(term_div t1 t2)->(n:nat)(full n t1)->
                                    (full n t2)->(Ttm t1 t2)\/(t1=t2).

Intros.
Elim ex_term_div_n with t1 t2 n; Intros; Trivial.
Elim H2; Intros.
Clear H2.
Elim eq_tm_dec with t1 t2; Intros.
Right; Trivial.
Left.
Elim eq_tm_dec with (n_term_0 n) x; Intros.
Rewrite <- y0 in H4.
Cut t1=(term_mult t1 (n_term_0 n)); Intros.
Rewrite <- H4 in H2.
Elim y; Auto.
Symmetry; Apply term_mult_full_nulo; Auto.
Cut (Ttm (n_term_0 n) x); Intros; Auto.
Replace t1 with (term_mult t1 (n_term_0 n)); Auto.
Rewrite H4.
Apply mon_term_mult with n; Auto.
□
```

3.6. Mínimo común múltiplo de términos

Otra operación sobre términos, que necesitaremos para implementar el Algoritmo de Buchbërger es el mínimo común múltiplo de términos (**lcm**). Antes de formalizar la definición en Coq de dicha operación, debemos definir el máximo de dos números naturales y probar algunos resultados sobre este máximo, utilizando el lema de decibilidad $((n \leq m) \vee (n > m))$ formalizado en Coq, en el módulo *Arith.Compare_dec* como *le_gt_dec*, así como la biblioteca **Omega** del sistema.

```
Lemma le_gt_dec: (n,m:nat){(le n m)}+{(gt n m)}.


Definition max_num: nat->nat->nat:= [n,m:nat]Cases (le_gt_dec n m) of
       (left _) => m
     | (right _) => n
  end.
```

Los cuatro lemas siguientes son resultados generales sobre enteros pero formulados de forma adecuada a nuestros propósitos.

```
Lemma max_num1: (n1,n2:nat)(le n1 n2)->((max_num n1 n2) = n2).


Lemma max_num2:(n,n1:nat)(le n1 n)->(n2:nat)(le n2 n)->(le (max_num n1 n2) n).
```

```
Lemma max_num0: (n:nat)((max_num n 0)=n).


Lemma max_num_conm: (n1,n2:nat)((max_num n1 n2) = (max_num n2 n1)).
□
```

Definición 3.6.1. *El* **mínimo común múltiplo (lcm)** *de dos términos,* $t_1 = x_1^{\alpha_1} x_2^{\alpha_2} \ldots x_n^{\alpha_n}$ *y* $t_2 = x_1^{\beta_1} x_2^{\beta_2} \ldots x_n^{\beta_n}$ *es el término dado por:*

$$x_1^{max(\alpha_1,\beta_1)} x_2^{max(\alpha_2,\beta_2)} \ldots x_n^{max(\alpha_n,\beta_n)}.$$

Traducción en **Coq**:

```
Fixpoint lcm [t1,t2:term]: term:= Cases t1 t2 of
   nil nil => (nil ?)
  |nil (cons y z) => (cons y z)
  |(cons y z) nil => (cons y z)
  |(cons n1 t1') (cons n2 t2') => (cons (max_num n1 n2) (lcm t1' t2'))
  end.
```

Al igual que hemos hecho con otras operaciones sobre términos dadas en las secciones precedentes, probamos que *el mínimo común múltiplo de dos términos de longitud n es otro término de longitud n.* La prueba es análoga a las realizadas para la operaciones antes citadas.

Traducción en **Coq**:

```
Lemma lcm_full: (n:nat)(t1:term)(t2:term)(full n t1)->(full n t2)->
                                          (full n (lcm t1 t2)).
□
```

Los siguientes resultados enuncian propiedades elementales sobre el mínimo común múltiplo.

Teorema 3.6.1. *El mínimo común múltiplo de dos términos verifica la propiedad conmutativa.*

$$\forall t, t' \in T;\ lcm(t, t') = lcm(t', t)$$

Prueba: Se hace por recurrencia sobre los términos y utilizando las propiedades sobre el máximo de dos enteros antes probadas.

Traducción en **Coq**:

```
Lemma lcm_conm: (t,t':term)((lcm t t')=(lcm t' t)).
□
```

Teorema 3.6.2.

$$\forall t_1, t_2 \in T;\ t_1|(lcm(t_1, t_2)) \ \wedge\ t_2|(lcm(t_1, t_2))$$

Prueba: Una vez hecha recurrencia sobre los términos, la prueba se deriva directamente de la aplicación de los constructores de la divisibilidad *term_div*.

Traducción en **Coq**:

```
Lemma div_lcm1: (t1,t2:term)(term_div t1 (lcm t1 t2)).

Lemma div_lcm2: (t1,t2:term)(term_div t2 (lcm t1 t2)).
```
□

Teorema 3.6.3.

$$\forall t \in T^n;\ lcm(t,(x_1^0 x_2^0 \ldots x_n^0)) = t$$

Prueba: Se obtiene directamente, por *Inversion*, de la formalización de mínimo común múltiplo.

Traducción en **Coq**:

```
Lemma lcm_1: (t:term)(n:nat)(full n t)->((lcm t (n_term_0 n)) = t).

Lemma lcm_2: (t:term)(n:nat)(full n t)->((lcm (n_term_0 n) t) = t).
```
□

Para finalizar esta sección, se prueban dos resultados simples que relacionan la divisibilidad con el mínimo común múltiplo de dos términos.

Teorema 3.6.4.

$$\forall t_1, t_2 \in T^n;\ (t_1|t_2) \Rightarrow lcm(t_1,t_2) = t_2$$

Prueba: Se demuestra por recurrencia sobre el predicado *term_div*; la primera submeta obtenida la resuelve el resultado anterior *lcm_2*. La única dificultad para probar la segunda submeta es tener en cuenta la longitud de los términos[8] para poder aplicar la hipótesis de recurrencia, lo cual se realiza mediante la reescritura.

Traducción en **Coq**:

```
Lemma lcm_div2: (t1,t2:term)(term_div t1 t2)->(n:nat)(full n t1)->
                  (full n t2)->((lcm t1 t2) = t2).

Induction 1; Intros.
Rewrite -> (nterm_null t H0 n H1); Auto.
Simpl.
Replace (lcm t t') with t'.
Replace (max_num n1 n2) with n2; Auto.
Symmetry; Auto.
Red in H3 H4.
Symmetry.
Rewrite <- H4 in H3.
Simpl in H3.
Apply H2 with (length t); Auto.
```
□

Teorema 3.6.5.

$$\forall t_1, t_2, t_3 \in T^n;\ (t_1|t_3) \wedge (t_2|t_3) \Rightarrow (lcm(t_1,t_2)|t_3)$$

[8] Este resultado sólo es válido para términos con el mismo número n de variables.

Prueba: La recurrencia estructural asociada a *term_div* genera dos submetas. La primera se resuelve mediante el lema *lcm_2*. Para la segunda aplicamos el principio de inducción asociado a *term* sobre el término t_3, que genera a su vez dos nuevas submetas. Estas se resuelven, una vez hecha la $\beta-$reducción mediante el comando **Simpl**, aplicando el segundo constructor *term_div_cons* de la divisibilidad, la hipótesis obtenida de la recurrencia estructural de *term_div* y las propiedades aritméticas usuales de los naturales.

Traducción en **Coq**:

```
Lemma lcm_div: (t1,t2:term)(term_div t1 t2)->(t3:term)(term_div t3 t2)->
  (n:nat)(full n t1)->(full n t2)->(full n t3)->(term_div (lcm t1 t3) t2).
□
```

Capítulo 4

Formalización de un cuerpo

Como uno de nuestros objetivos es la formalización en **Coq** del **anillo de polinomios sobre un cuerpo**, necesitamos definir e implementar en **Coq** la estructura de **cuerpo** que necesitaremos para la definición de monomio[1].

En este capítulo axiomatizamos la estructura de cuerpo y probamos diversas lemas sobre esta estructura que utilizaremos en los capítulos siguientes.

4.1. Definición

En esta sección, introducimos la definición, paso a paso, de la noción de **cuerpo**. No entraremos a recordar las nociones de **conmutatividad, asociatividad, operación distributiva, elemento neutro, elemento simétrico,** que se pueden encontrar en cualquier libro de álgebra general.

Definición 4.1.1. *Un conjunto G y una operación interna*[2] *$(*)$ definida en él, se dice que forman un* **grupo** *$(G, *)$ si se verifican las siguientes propiedades (axiomas de grupo):*

- *la operación $*$ es asociativa.*
- *G posee elemento neutro.*
- *todo elemento de G es simetrizable.*

Se dice que $(G, *)$ es un **grupo conmutativo** *o abeliano si, además, se verifica que la operación $*$ es conmutativa.*

Definición 4.1.2. *Un conjunto A con dos operaciones internas definidas en él ($\#$ y $*$), se dice que forman un* **anillo** *$(A; \#, *)$*[3] *si se verifican las siguientes propiedades (axiomas de anillo):*

[1] Formalizada en el capítulo siguiente.

[2] Dado un conjunto $G \neq \emptyset$, se llama operación interna definida en G a toda aplicación $*$, de la forma siguiente: $G \times G \xrightarrow{*} G$.

[3] Normalmente la suma $(+)$ y el producto $(.)$.

- $(A, \#)$ *es un grupo conmutativo.*
- *la operación* $*$ *es asociativa.*
- *la operación* $*$ *es distributiva respecto a* $\#$.

Se dice que $(A; \#, *)$ *es* **anillo unitario** *si, además, se verifica que existe elemento neutro*[4] *para la segunda operación* $(*)$.

Se dice que $(A; \#, *)$ *es* **anillo conmutativo** *si, además, se verifica que* $(*)$ *es conmutativa.*

Definición 4.1.3. *Un anillo* $(K; \#, *)$ *se dice que es un* **cuerpo** *si* $K^* = K - \{0\}$ *(donde* 0 *es el elemento nulo) es un grupo conmutativo respecto a* $(*)$. *Los cuerpos son, pues, los anillos unitarios y conmutativos* $(K; \#, *)$ *en los que, además, todo* $a \in K^*$ *admite inverso*[5] $a^{-1} \in K^*$.

Nota: Algunos autores llaman cuerpo a la estructura que resulta al prescindir, en la definición anterior, de la conmutatividad de la segunda operación.

4.2. Axiomatización

Cuando se decide axiomatizar una estructura, tenemos que comenzar por definir el tipo de dato principal, en nuestro caso el tipo K, representando un cuerpo. En **Coq**, como hemos visto en el capítulo 2, se pueden elegir tres clases, es decir que podemos asignar a K los tipos siguientes:
$K : Prop.$
$K : Set.$
$K : Type.$

Desde el punto de vista teórico podríamos asignar a K el tipo **Prop**, pero desde el punto de vista práctico tiene dos grandes problemas. El primero que K es un conjunto, normalmente de números, y no una proposición. La segunda es que con la clase **Prop** no podríamos utilizar la eliminación fuerte, ver [117].

El problema es decidir entre las clases **Set** y **Type**. Elegimos la primera debido a que es una "clase calculatoria", que permite la extracción y es el tipo de las especificaciones. Además esta elección es coherente con los tipos utilizados en este trabajo para números, tales como los enteros naturales **nat**.

4.2.1. Operadores

En primer lugar fijamos el tipo de K.
Traducción en **Coq**:

```
Variable K:Set.
```

[4] Unidad.
[5] Simétrico respecto a la segunda operación $*$

Definimos ahora como parámetros las operaciones de la estructura ($plusK$, opK, $multK$, $invK$), así como los elementos destacados (OK,unK) que nos serán útiles para formalizar el hecho de que K es un cuerpo commutativo. Además también fijamos un predicado sobre K (eqK) que representará la igualdad de elementos del cuerpo ($=_K$).

Traducción en **Coq**:

```
Variable opK: K -> K.        (* el opuesto *)
Variable plusK: K->K->K.     (* la adición *)
Variable multK: K->K->K.     (* el producto *)
Variable invK: K -> K.       (* el inverso *)

Variable OK:K.               (* el 0 del cuerpo *)
Variable unK: K.             (* el 1 del cuerpo *)

Variable eqK: K->K->Prop.    (* la igualdad *)
```

El cuerpo sobre el que se trabajará debe ser **no trivial**, es decir distinto del $\{0\}$, para ello necesitamos que tenga al menos dos elementos.

Traducción en **Coq**:

```
Axiom no_trivial: ~(eqK unK OK).
```

4.2.2. Los axiomas

Axiomatizar una estructura es dar un conjunto mínimo de axiomas que nos permita especificarla. A continuación se detalla la axiomatización de la estructura de cuerpo.

En primer lugar introducimos los axiomas necesarios para que el predicado eqK sea una relación de equivalencia.

Traducción en **Coq**:

```
Hypothesis eqK_refl: (reflexive K eqK).

Hypothesis eqK_sym: (symmetric K eqK).

Hypothesis eqK_trans: (transitive K eqK).
```

Axiomatizamos las propiedades sobre K con respecto a los operadores dados para que la estructura $(K, plusK, multK)$ sea un cuerpo. Los axiomas son los siguientes:

- Propiedades de la adición.

  Traducción en **Coq**:

  ```
  Hypothesis plusK_conm: (k,k':K)(eqK (plusK k k') (plusK k' k)).

  Hypothesis plusK_assoc: (k1,k2,k3:K)(eqK (plusK k1 (plusK k2 k3))
                                           (plusK (plusK k1 k2) k3)).

  Hypothesis neut_plusK: (k:K)(eqK (plusK k OK) k).

  Hypothesis ex_op: (k:K) (eqK (plusK k (opK k)) OK).
  ```

- Propiedades del producto

  Traducción en **Coq**:

```
Hypothesis multK_conm: (k,k':K)(eqK (multK k k') (multK k' k)).

Hypothesis multK_assoc: (k1,k2,k3:K)(eqK (multK k1 (multK k2 k3))
                                         (multK (multK k1 k2) k3)).

Hypothesis neut_multK: (k:K)(eqK (multK k unK) k).

Hypothesis inv_divK:(k:K)(~(eqK k OK))->(eqK (multK k (invK k)) unK).
```

- Propiedad distributiva del producto respecto de la suma.

  Traducción en **Coq**:

```
Hypothesis distrK: (k1,k2,k3:K) (eqK (multK k1 (plusK k2 k3))
                                     (plusK (multK k1 k2) (multK k1 k3))).
```

Además de las propiedades anteriores, necesitamos que las operaciones implementadas, $plusK$ y $multK$, sean compatibles por la derecha[6] con la relación eqK.

Traducción en **Coq**:

```
Axiom plusK_comp_r: (k,k':K)(eqK k k')->(y:K)(eqK (plusK k y) (plusK k' y)).

Axiom multK_comp_r: (k,k':K)(eqK k k')->(y:K)(eqK (multK k y) (multK k' y)).
```

Por último, axiomatizamos la decidibilidad de la relación eqK sobre elementos de K con relación al elemento cero del cuerpo (OK).

```
Axiom decK: (k:K){eqK k OK}+{~(eqK k OK)}.
```

4.3. Propiedades algebraicas de un cuerpo

El cociente[7] de dos elementos de un cuerpo ($k, k' \in K$) donde ($k' \neq_K 0$), que será denotado ($\frac{k}{k'}$), se define, de manera usual, como el producto de k por el inverso de k', normalmente denotado como ($\frac{1}{k'}$) o $(k')^{-1}$.

Traducción en **Coq**:

```
Definition divK:= [k1,k2:K] (multK k1 (invK k2)).
```

Sin detallar las pruebas, salvo las primeras, debido a que todas se realizan utilizando reglas anteriores, el axioma eqK_trans, la decibilidad de eqK y la explicitación de la negación, he aquí algunos resultados básicos.

[6] Por la izquierda lo conseguimos con la simetría de eqK.

[7] Por razones de legibilidad, es más claro definir el operador cociente.

Teorema 4.3.1. *La suma y la multiplicación de elementos del cuerpo son compatibles con la igualdad.*

Prueba: Como tenemos como axioma la compatibilidad por la derecha de estas operaciones con respecto a eqK y además también ambas verifican la propiedad conmutativa, la demostración se basa en aplicar los axiomas anteriores.

Se quiere probar la "igualdad" $(eqK\ (plusK\ y\ k)\ (plusK\ y\ k'))$ a partir de la hipótesis $H : (eqK\ k\ k')$.
Procedemos por etapas de reescritura a partir de la meta:
Aplicando eqK_trans a la meta, con respecto a $(plusK\ k\ y)$, se obtienen dos nuevas metas:

- $(eqK\ (plusK\ y\ k)\ (plusK\ k\ y))$ [axioma $plusK\_conm$]
- $(eqK\ (plusK\ k\ y)\ (plusK\ y\ k'))$

Aplicando eqK_trans, con respecto a $(plusK\ k'\ y)$, a la meta anterior, tenemos:

- $(eqK\ (plusK\ k\ y)\ (plusK\ k'\ y))$ [axioma $plusK\_comp\_r$]
- $(eqK\ (plusK\ k'\ y)\ (plusK\ y\ k'))$ [axioma $plusK\_conm$]

La prueba del resultado $multK\_comp\_l$ es análoga, utilizando los axiomas del producto.

Traducción en **Coq**:

```
Lemma plusK_comp_l: (k,k':K)(eqK k k')->(y:K)(eqK (plusK y k) (plusK y k')).


Lemma multK_comp_l: (k,k':K)(eqK k k')->(y:K)(eqK (multK y k) (multK y k')).


Lemma plusK_comp: (k,k',c,c':K)(eqK k k')->(eqK c c')->
                    (eqK (plusK k c) (plusK k' c')).


Lemma multK_comp: (k,k',c,c':K)(eqK k k')->(eqK c c')->
   (eqK (multK k c) (multK k' c')).
```

□

Se puede comprobar que estas pruebas difieren del camino seguido de las hechas en el capítulo anterior. Esto es debido a que aquí no estamos utilizando tipos inductivos.

Teorema 4.3.2.

$$\forall k, k' \in K;\ (k =_K k') \wedge (k \neq_K 0) \Rightarrow (k' \neq_K 0).$$

Traducción en **Coq**:

```
Lemma comp_nOK: (k,k':K)(eqK k k')->(~(eqK k OK))->(~(eqK k' OK)).
```

□

Teorema 4.3.3.

$$\forall k, k', y \in K;\ k + y =_K k' + y \ \Rightarrow\ k =_K k'.$$

Traducción en **Coq**:

```
Lemma suma_igual:(k,k',y:K)(eqK (plusK k y) (plusK k' y)) ->(eqK k k').
```
□

Teorema 4.3.4.

$$\forall k, k' \in K;\ (k + k' =_K 0)\ \Rightarrow\ (k =_K -k').$$

Traducción en **Coq**:

```
Lemma suma_op_inv: (k,k':K)(eqK (plusK k k') OK)->(eqK k (opK k')).
```
□

Teorema 4.3.5.

$$0 =_K -0.$$

Traducción en **Coq**:

```
Lemma neg_OK: (eqK (opK OK) OK).
```
□

Teorema 4.3.6.

$$\forall k \in K;\ (k \neq_K 0)\ \Rightarrow\ (-k \neq_K 0).$$

Traducción en **Coq**:

```
Lemma multK_n_OK: (k:K)(~(eqK k OK))->(~(eqK (opK ) OK)).
```
□

Teorema 4.3.7.

$$\forall k \in K;\ k =_K -(-k).$$

Traducción en **Coq**:

```
Lemma neg_neg_K: (k:K)(eqK k (opK (opK k))).
```
□

Teorema 4.3.8.

$$\forall k \in K;\ k \,.\, 0 =_K 0$$

Traducción en **Coq**:

```
Lemma multK_zero:(k:K)(eqK (multK k OK) OK).
```
□

Teorema 4.3.9.

$$\forall k \in K;\ k.(-1) =_K -k$$

Traducción en **Coq**:

```
Lemma opK_unK: (k:K)(eqK (multK k (opK unK)) (opK k)).
```
□

Teorema 4.3.10.

$$\forall k, k' \in K;\ -(k + k') =_K (-k) + (-k').$$

Traducción en **Coq**:

```
Lemma opK_plusK: (k,k':K)(eqK (opK (plusK k k')) (plusK (opK k) (opK k'))).
□
```

Teorema 4.3.11.

$$\forall k, k' \in K;\ -(k.k') =_K (-k).k' =_K k.(-k')$$

Traducción en **Coq**:

```
Lemma opK1: (k,k':K)(eqK (opK (multK k k')) (multK (opK k) k')).

Lemma opK2: (k,k':K)(eqK (opK (multK k k')) (multK k (opK k'))).
□
```

Teorema 4.3.12.

$$\forall k, x, y \in K;\ (k \neq_K 0) \wedge (k.x =_K k.y) \Rightarrow x = y$$

Traducción en **Coq**:

```
Lemma igual:(k1,k2,x,y:K)(eqK k1 k2) ->(~(eqK k1 OK))->
          (eqK (multK k1 x) (multK k2 y))->(eqK x y).
□
```

Teorema 4.3.13. *Un cuerpo no tiene divisores de cero propios*[8].

Traducción en **Coq**:

```
Lemma zero_multK: (c,c':K)((eqK c OK)\/(eqK c' OK))->(eqK (multK c c') OK).

Lemma resul_multK_zero:(c1,c2:K)(~(eqK (multK c1 c2) OK))->
                          (~(eqK c1 OK))/\(~(eqK c2 OK)).
□
```

Teorema 4.3.14.

$$\forall k \in K;\ (k \neq_K 0) \Rightarrow (\frac{1}{k} \neq_K 0).$$

Traducción en **Coq**:

```
Lemma no_zero_invK: (k:K)(~(eqK k OK))->~(eqK (invK k) OK).
□
```

Teorema 4.3.15.

$$\forall k, k_1, k_2 \in K;\ (k \neq_K 0) \Rightarrow \frac{k_1}{k}.k_2 =_K \frac{k_1.k_2}{k}$$

Traducción en **Coq**:

[8] En un anillo $(A; +, .)$, se dice que un elemento no nulo $a \in A$ es un *divisor de cero* si existe algún otro elemento no nulo $b \in A$ tal que alguno de los $a\ .\ b$ ó $b\ .\ a$ es el elemento nulo.

```
Lemma multK_divK1: (k,k1,k2:K)(~(eqK k OK))->
      (eqK (multK (divK k1 k) k2) (divK (multK k1 k2) k)).
```
□

Teorema 4.3.16.

$$\forall k_1, k_2 \in K;\ (k_1.k_2 =_K 1) \Rightarrow k_2 =_K \frac{1}{k_1}$$

Traducción en **Coq**:

```
Lemma ec_invK: (k1,k2:K)(eqK (multK k1 k2) unK)->(eqK k2 (invK k1)).
```
□

Teorema 4.3.17.

$$\forall k \in K;\ (k \neq_K 0) \Rightarrow \frac{0}{k} =_K 0.$$

Traducción en **Coq**:

```
Lemma divK_O_n: (k:K)~(eqK k OK)->(eqK (divK OK k) OK).
```
□

Teorema 4.3.18.

$$\forall k, k_1, k_2 \in K;\ (k_2 \neq_K 0) \Rightarrow k + \frac{k_1}{k_2} =_K \frac{k.k_2 + k_1}{k_2}$$

Traducción en **Coq**:

```
Lemma plusK_frac: (k,k1,k2:K)(~(eqK k2 OK))->
   (eqK (plusK k (divK k1 k2)) (divK (plusK (multK k k2) k1) k2)).
```
□

Teorema 4.3.19. *Decibilidad de la relación eqK sobre elementos de un cuerpo K.*

Traducción en **Coq**:

```
Lemma decK_t: (k,k':K) {eqK k k'}+{~(eqK k k')}.
```
□

Capítulo 5

Monomios

En los capítulos anteriores formalizamos en **Coq** los términos de n variables, el conjunto denotado por T^n, y la estructura de cuerpo. A partir de estas formalizaciones construimos ahora la noción de **monomio** con coeficientes en un cuerpo K. A continuación se define un operador que juega el papel de la relación de igualdad sobre monomios, fundamental para definir la igualdad de polinomios. Se termina introduciendo diversas operaciones sobre monomios y la prueba de sus propiedades que serán claves en la implementación del tipo polinomio en el siguiente capítulo.

5.1. Definición e igualdad de monomios

Definición 5.1.1. *Siendo K un cuerpo, un* **monomio** *es un par formado por un coeficiente $a \in K$ y un término $x_1^{\alpha_1} x_2^{\alpha_2} \ldots x_n^{\alpha_n}$, expresado de la forma:*

$$a x_1^{\alpha_1} x_2^{\alpha_2} \ldots x_n^{\alpha_n}$$

Para representar el conjunto de monomios sobre el cuerpo K utilizamos la notación M_K.

Traducción en **Coq**:

```
Definition monom := K*term.
```

Los monomios de n variables son los pares formados por un elemento del cuerpo y un término ($t \in T^n$) de n variables, conjunto que denotamos $M_{K,n}$.

Traducción en **Coq**:

```
Definition full_mon: nat->monom->Prop := [n:nat][m:monom]Cases m of
  (pair c t) => (full n t)
  end.
```

Comenzamos con las nociones de monomio cero 0, y la relación de igualdad sobre monomios *equmon*.

Se define el **monomio cero** 0 mediante un predicado que satisfacen todos los monomios cuyos coeficientes sean cero.

Definición 5.1.2. *Llamamos* **monomio cero**, *denotado* 0, *a cualquier expresión del tipo:*

$$(0_K, t)$$

donde 0_K *es el elemento neutro de la operación aditiva de cuerpo* K *y* t *es un término cualquiera* $(\forall t \in T)$.

Para su formalización utilizamos el tipo producto, *prod*[1], predefinido en las bibliotecas de Coq.

Traducción en **Coq**:

```
Inductive prod [A:Set; B:Set] : Set :=  pair : A->B->(prod A B).

Definition z_monom := [m:monom]Cases m of
   (pair q t) => (eqK q OK)
   end.
```

Hemos definido los monomios como pares (q, t) con $q \in K$ y $t \in T$; por ello, para formalizar su igualdad, lo hacemos comparándolos componente a componente.

Definición 5.1.3. *Se define la igualdad* $(=_M)$ *de dos monomios* $m_1 = ax_1^{\alpha_1}x_2^{\alpha_2}\ldots x_n^{\alpha_n}$ *y* $m_2 = bx_1^{\beta_1}x_2^{\beta_2}\ldots x_n^{\beta_n}$ *de la forma siguiente:*

$$m_1 =_M m_2 :\overset{def}{\Longleftrightarrow} \begin{cases} (a =_K 0_K) \wedge (b =_K 0_K) \\ o \\ ((x_1^{\alpha_1}x_2^{\alpha_2}\ldots x_n^{\alpha_n}) = (x_1^{\beta_1}x_2^{\beta_2}\ldots x_n^{\beta_n})) \wedge (a =_K b) \end{cases}$$

Traducción en **Coq**:

```
Definition equmon :=[m,m':monom]Cases m m' of
  (pair c t)(pair c' t') => ((eqK c OK)/\(eqK c' OK))\/((t=t')/\(eqK c c'))
  end.
```

Comprobamos que, como era de esperar, la relación de igualdad de monomios $(=_M)$ dada es una relación de equivalencia. Las demostraciones de estas propiedades se hacen desdoblando las definiciones de *monom* y de *equmon*; la única dificultad es saber en cada submeta cuál de las ramas de la definición de igualdad de monomios debemos elegir, y utilizar las propiedades vistas para las igualdades de términos y de elementos de un cuerpo.

Hay que hacer notar que en la igualdad de monomios interviene la igualdad de elementos de un cuerpo, donde no se utiliza la igualdad sintáctica (de Leibniz), con lo cual ya no podemos hacer uso de tácticas predefinidas en el sistema para la relación de la igualdad de Leibniz, como **Reflexivity**, **Symmetry** o **Transitiviy**, ni órdenes como **Rewrite** o **Replace** cuando se trabaje con el tipo monomio.

Teorema 5.1.1. *La igualdad de monomios verifica la propiedad reflexiva.*

$$\forall m \in M_K;\ (m =_M m)$$

[1] Ver biblioteca *Init.Datatypes*.

Traducción en **Coq**:

```
Lemma equmon_refl: (m:monom) (equmon m m).
```
□

Teorema 5.1.2. *La igualdad de monomios verifica la propiedad simétrica.*

$$\forall m, m' \in M_K;\ (m =_M m') \Rightarrow (m' =_M m)$$

Traducción en **Coq**:

```
Lemma equmon_sym: (m,m':monom)(equmon m m')->(equmon m' m).
```
□

Teorema 5.1.3. *La igualdad de monomios verifica la propiedad transitiva.*

$$\forall m_1, m_2, m_3 \in M_K;\ (m_1 =_M m_2) \wedge (m_2 =_M m_3) \Rightarrow (m_1 =_M m_3)$$

Traducción en **Coq**:

```
Lemma equmon_trans: (m1,m2,m3:monom) (equmon m1 m2)->
                    (equmon m2 m3)->(equmon m1 m3).
```
□

Los dos resultados siguientes relacionan el predicado *z_monom* (monomio cero) con el predicado igualdad de monomios *equmon*. Se prueban por casos sobre el tipo *monom*. Sirven para automatizar pruebas posteriores.

Teorema 5.1.4.

$$\forall m, m' \in M_K;\ (z\_monom\ m) \wedge (m =_M m') \Rightarrow (z\_monom\ m')$$

Traducción en **Coq**:

```
Lemma equmon_z: (m,m':monom)(z_monom m)->(equmon m m')->(z_monom m').
```
□

Teorema 5.1.5.

$$\forall m, m' \in M_K;\ (z\_monom\ m) \wedge (z\_monom\ m') \Rightarrow (m =_M m')$$

Traducción en **Coq**:

```
Lemma z_equmon:(m,m':monom)(z_monom m)->(z_monom m')->(equmon m m').
```
□

Teorema 5.1.6. *El predicado z_monom es decidible.*

Prueba: Se deduce inmediatamente de la decibilidad del predicado eqK con relación al elemento cero del cuerpo ($decK$), pues el predicado *z_monom* está definido en función de él.

Traducción en **Coq**:

```
Lemma z_monom_dec: (m:monom){z_monom m}+{~(z_monom m)}.
```
□

5.2. Operaciones de monomios

El producto de monomios se define en función del producto de elementos del cuerpo y del producto de términos.

Definición 5.2.1. *Dados dos monomios* $m_1 = ax_1^{\alpha_1}x_2^{\alpha_2}\ldots x_n^{\alpha_n}$ *y* $m_2 = bx_1^{\beta_1}x_2^{\beta_2}\ldots x_n^{\beta_n}$, *el producto de dichos monomios es el monomio* $m_1.m_2$*:*

$$m_1.m_2 := (a.b)(x_1^{\alpha_1+\beta_1}x_2^{\alpha_2+\beta_2}\ldots x_n^{\alpha_n+\beta_n})$$

Traducción en **Coq**:

```
Definition mult_mon:= [m1,m2:monom]Cases m1 m2 of
   (pair k1 t1) (pair k2 t2) => (pair ? ? (multK k1 k2) (term_mult t1 t2))
    end.
```

Hay que hacer notar que las interrogaciones en la definición anterior permiten que la función **pair**, único constructor del tipo producto en Coq, deduzca de forma implícita el tipo empleado. Al igual que en formalizaciones anteriores, esta operación se formaliza para monomios cualesquiera. Cuando sea necesario se trabajará sobre monomios de n variables sin más que agregarle al tipo *monom* el predicado *full_mon*.

Ahora ya estamos en condiciones de enunciar y probar propiedades algebraicas sobre el producto de monomios de n variables. *El producto de monomios de n variables es otro monomio de n variables*, como consecuencia de que el producto de términos de n variables es otro término de n variables.

Traducción en **Coq**:

```
Lemma mul_mon_full: (m,m':monom)(n:nat)(full_mon n m)->(full_mon n m')->
                                        (full_mon n (mult_mon m m')).
```

□

Las demostraciones de los tres lemas siguientes se siguen inmediatamente de las propiedades algebraicas del producto de términos y del producto definido en un cuerpo.

Teorema 5.2.1. *El producto de monomios verifica la propiedad conmutativa.*

Traducción en **Coq**:

```
Lemma mult_mon_conm: (m,m':monom)(equmon (mult_mon m m') (mult_mon m' m)).
```

□

Teorema 5.2.2. *El producto de monomios verifica la propiedad asociativa.*

Traducción en **Coq**:

```
Lemma mult_mon_asoc: (m,m1,m2:monom)
   (equmon (mult_mon m (mult_mon m1 m2)) (mult_mon (mult_mon m m1) m2)).
```

□

Teorema 5.2.3. *El producto de monomios es compatible con la igualdad.*

$$\forall m, m_1, m_2 \in M_K;\ (m_1 =_M m_2) \Rightarrow (m_1.m) =_M (m_2.m)$$

Traducción en **Coq**:

```
Lemma mult_mon_comp_l: (m,m1,m2:monom)(equmon m1 m2)->
          (equmon (mult_mon m1 m) (mult_mon m2 m)).
```

□

Ahora probamos un corolario de los resultados anteriores, útil para posteriores pruebas.

Corolario 5.2.1.

$$\forall m, m_1, m_2 \in M_K;\ (m.m_1).m_2 =_M m_1.(m.m_2)$$

Prueba:

$$(m.m_1).m_2 =_M (m_1.m).m_2 =_M m_1.(m.m_2)$$

Traducción en **Coq**:

```
Lemma mult_mon_perm: (m,m1,m2:monom)
  (equmon (mult_mon (mult_mon m m1) m2) (mult_mon m1 (mult_mon m m2))).
```

□

El único monomio nilpotente es el monomio cero, como prueban los dos resultados siguientes.

Teorema 5.2.4.

$$\forall m, m' \in M_K;\ (z\_monom\ m) \vee (z\_monom\ m') \Rightarrow (z\_monom\ (m.m'))$$

Traducción en **Coq**:

```
Lemma monot_zmonom: (m,m':monom)((z_monom m)\/(z_monom m'))->
                                 (z_monom (mult_mon m m')).
```

□

Teorema 5.2.5.

$$\forall m, m' \in M_K;\ \neg(z\_monom\ m) \wedge \neg(z\_monom\ m') \Rightarrow \neg(z\_monom\ (m.m'))$$

Traducción en **Coq**:

```
Lemma monot_no_zmonom: (m,m':monom)(~ (z_monom m)/\~ (z_monom m'))->
                                     (~(z_monom (mult_mon m m'))).
```

□

A veces es necesario trabajar solamente con el coeficiente o con el término de un monomio; las definiciones que siguen formalizan las proyecciones del tipo *monom*, que permiten obtener el coeficiente y el término de un monomio dado. Para estas proyecciones se utilizará la misma notación que el sistema, $(mon\_coef\ m)$ y $(mon\_term\ m)$, para un monomio m.

Traducción en **Coq**:

```
Definition mon_coef := [m:monom]<K> Case m of
   [q:K][t:term] q
   end.

Definition mon_term := [m:monom]<term> Case m of
   [q:K][t:term] t
   end.
```

Se necesita el concepto de monomio opuesto a uno dado.

Definición 5.2.2. *El monomio opuesto a uno dado* (a,t)*, es el monomio denotado por* $(-a,t)$.

Traducción en **Coq**:

```
Definition monom_op := [m:monom]((opK (mon_coef m)),(mon_term m)).
```

De esta definición se sigue que *un monomio y su opuesto tienen el mismo término.*

Traducción en **Coq**:

```
Lemma mon_opp_term : (m:monom)(mon_term m)=(mon_term (monom_op m)).
```

□

Para finalizar esta sección formalizamos en **Coq** dos operaciones sobre monomios que utilizaremos más adelante, estas son el producto de un elemento del cuerpo por un monomio y el cociente de monomios.

Definición 5.2.3. *Dado un elemento del cuerpo,* $c \in K$*, y un monomio denotado por* $m = ax_1^{\alpha_1}x_2^{\alpha_2}\dots x_n^{\alpha_n}$*, el producto de ellos* $(c.m)$*, es el monomio*

$$c.m := (c.a)x_1^{\alpha_1}x_2^{\alpha_2}\dots x_n^{\alpha_n}.$$

Dados dos monomios $m_1 = ax_1^{\alpha_1}x_2^{\alpha_2}\dots x_n^{\alpha_n}$ *y* $m_2 = bx_1^{\beta_1}x_2^{\beta_2}\dots x_n^{\beta_n}$*, tomando* $b \neq_K 0_K$*, el cociente de dichos monomios es el monomio, denotado* $\frac{m_1}{m_2}$*:*

$$\frac{m_1}{m_2} := (\frac{a}{b})(x_1^{\alpha_1-\beta_1}x_2^{\alpha_2-\beta_2}\dots x_n^{\alpha_n-\beta_n}).$$

Traducción en **Coq**:

```
Definition mult_K_monom:= [c:K][m:monom]Cases m of
   (pair c1 t) => (pair ? ? (multK c c1) t)
   end.

Definition div_mon:= [m1,m2:monom]Cases m1 m2 of
   (k1,t1) (k2,t2) => ((divK k1 k2),(div_term t1 t2))
   end.
```

Hay que resaltar que sólo se podrá dividir el monomio m_1 por el monomio m_2, $\frac{m_1}{m_2}$, cuando el término de m_1 sea divisible por el de m_2.

Capítulo 6

Polinomios

El dominio de los polinomios resulta fundamental tanto en cálculo formal como en computación simbólica, siendo además una herramienta indispensable para la ciencia de la computación (criptografía, códigos, computación algebraica,...) [47, 108]. El principal objetivo de este capítulo es el estudio y la formalización en Coq de los polinomios en varias variables sobre un cuerpo. Una vez construidos los polinomios y dada la relación de igualdad, se implementan las operaciones algebraicas en polinomios, así como operaciones puramente sintácticas, que extraen información directamente accesible de los polinomios (como *hcoef* y *hterm*). Como hemos definido un polinomio como suma de monomios, queremos por razones de eficiencia "arreglar", sin ambigüedades, los términos de un polinomios en orden descendente y eliminar los de coeficiente nulo. Para ello se implementan los polinomios canónicos y se restringen a ellos las operaciones algebraicas antes citadas.

6.1. Definición y notaciones básicas

Dado un número natural n, definiremos los polinomios en n variables, denotadas $x_1, \ldots, x_n$ con coeficientes en un cuerpo K. Salvo en algunos ejemplos en donde utilizaremos polinomios con $n = 2$ o $n = 3$, se considerará un genérico n. Por eso reservamos a partir de ahora, en la formalización en **Coq**, una variable llamada n, de tipo **nat**, que expresará la longitud de los términos (número de variables) de un polinomio.

```
Variable n:nat.
```

Definición 6.1.1. *Un* **polinomio** *es una suma finita de monomios con coeficientes en* K.

Por ejemplo, $f = 2\,x_2 - 3\,x_1x_2^3 + 7\,x_2x_3^3 + 5$ es un polinomio en tres variables cuyo primer monomio se compone del coeficiente 2 y del término $x_1^0x_2x_3^0$, mientras que el último monomio tiene por coeficiente 5 y por término $x_1^0x_2^0x_3^0$.

En Coq representamos, de forma recursiva, un polinomio como una lista de monomios.

Traducción en **Coq**:

```
Inductive pol : Set := vpol : pol
                     | cpol : monom -> pol->pol.
```

Una restricción que debe tener en cuenta esta representación computacional de los polinomios es que no haya ambigüedad con las notaciones usuales en matemáticas.

El polinomio del ejemplo anterior se puede escribir poniendo las tres variables de forma explícita en todos los monomios, empleando exponentes cero para todas aquellas variables que no aparecen en el monomio. El polinomio anterior de esta forma quedaría:

$$f = 2\ x_1^0 x_2 x_3^0 - 3\ x_1 x_2^3 x_3^0 + 7\ x_1^0 x_2 x_3^3 + 5\ x_1^0 x_2^0 x_3^0$$

Un polinomio p cuyos monomios tienen todos n variables, de forma explícita como el anterior, lo identificamos en Coq mediante un predicado denotado *full_term_pol*.

Traducción en **Coq**:

```
Inductive full_term_pol: pol->Prop := full_term_pol_nil: (full_term_pol vpol)
  | full_term_pol_cons : (t:term)(a:K)(p:pol)(full n t)->(full_term_pol p)->
                         (full_term_pol (cpol (a,t) p)).
```

Un polinomio (*cpol m vpol*), con un solo monomio m de n variables, es el primer ejemplo de este tipo de polinomios.

Traducción en **Coq**:

```
Lemma full_term_cpol: (m:monom)(full_mon n m)->(full_term_pol (cpol m vpol)).
□
```

Análogamente, si tenemos un polinomio que satisface el predicado anterior, *full_term_pol*, podemos demostrar que cualquier monomio que lo compone tiene n variables. La prueba, una vez desdoblado m y el predicado *full_mon*, se hace obteniendo, mediante "inversion", la información del predicado *full_term_pol*, una de cuyas premisas es precisamente la meta en curso.

Traducción en **Coq**:

```
Lemma full_term_pol_mon: (m:monom)(p:pol)
        (full_term_pol (cpol m p))->(full_mon n m).
□
```

Denotamos por $K[x_1, \ldots, x_n]$ el conjunto de todos los polinomios en n variables con coeficientes en un cuerpo K; la notación $K[X]$ significará el conjunto de polinomios con coeficientes en K y cuyas variables son los elementos de un conjunto X.

Debemos probar el lema de decidibilidad de términos de n variables; la prueba es trivial debido a que es una variante del lema de decidibilidad de **nat**.

Traducción en **Coq**:

```
Lemma full_tm_dec : (t:term){(full n t)}+{~(full n t)}.
□
```

El programa extraído, en **OCaml** y sin optimización es:

```
type nat =
  | O
  | S of nat

let rec eq_nat_dec nO m =
  match nO with
    | O -> (match m with
              | O -> true
              | S n1 -> false)
    | S n1 -> (match m with
                 | O -> false
                 | S n2 -> eq_nat_dec n1 n2)

type 'a list =
  | Nil
  | Cons of 'a * 'a list

let rec length = function
  | Nil -> O
  | Cons (a, m) -> S (length m)

let full_tm_dec t =
  eq_nat_dec n (length t)
```

A su vez la demostración de la decidibilidad del predicado *full_term_pol* se hace por inducción sobre el polinomio *p*. El caso *vpol* se deduce de la formalización del predicado *full_term_pol*; el caso *(cpol m p)* se obtiene mediante el lema de decidibilidad de términos, la hipótesis de inducción y la información obtenida del predicado *full_term_pol* mediante la táctica de "inversion". El correspondiente lema, del cual se incluye una demostración, es:

```
Lemma full_pol_tm_dec : (p:pol){(full_term_pol p)}+{~(full_term_pol p)}.

Induction p.
Left; Auto.
Induction m; Intros c t p' HR.
Elim (full_tm_dec t).
Intros vrai; Elim HR.
Intros v2; Left; Auto.
Intros f2; Right ; Unfold not; Intros f3.
Apply f2; Inversion f3; Trivial.
Intros f2; Right ; Unfold not; Intros f3; Apply f2; Inversion f3; Trivial.
□
```

El programa extraído, en **OCaml** y sin optimización "a mano" es:

```
type ('a, 'b) prod =
  | Pair of 'a * 'b

type term = nat list

type monom = (Obj.t, term) prod
```

```
type pol =
  | Vpol
  | Cpol of monom * pol

let rec full_pol_tm_dec = function
  | Vpol -> true
  | Cpol (m, p0) ->
      let Pair (x, x0) = m in
      (match full_tm_dec x0 with
         | true -> full_pol_tm_dec p0
         | false -> false)
```

Hay que decir que en la representación computacional de polinomios hemos elegido dos niveles de aproximación:

- Polinomios definidos como listas de monomios, posiblemente no ordenados.
- Más adelante, cuando formalicemos los algoritmos para el cálculo de bases de Gröbner, trabajaremos exclusivamente con polinomios **canónicos** (ordenados y sin coeficientes cero).

Esto significa que primero definimos versiones eficientes, simples y claras de las operaciones matemáticas como la suma y el producto de polinomios. Otras variantes de estas operaciones se definirán más adelante y se mostrará que son equivalentes a las primeras cuando se aplican a polinomios canónicos.

Se define el producto de un elemento del cuerpo por un polinomio como el polinomio con los mismos términos y los coeficientes multiplicados por dicho elemento del cuerpo.

```
Fixpoint mult_K_pol [c:K;p:pol]: pol:=Cases p of
      vpol    =>   vpol
    | (cpol m1 p1) => (cpol (mult_K_monom c m1) (mult_K_pol c p1))
    end.
```

6.2. Adición e igualdad de polinomios

Presentamos aquí las formalizaciones de suma e igualdad de polinomios, así como las propiedades necesarias para el desarrollo posterior.

Observemos que, igual que en capítulos anteriores, no explicitamos en la formalización de las operaciones el número de variables. Las razones son las mismas que las expuestas en dichos capítulos.

6.2.1. Función suma

La función *suma_app* ($+_p$) consiste en la concatenación de listas de monomios. Como hemos hecho para formalizar el producto de un elemento de un cuerpo por un polinomio, utilizamos **Fixpoint**.

Traducción en **Coq**:

```
Fixpoint suma_app [p:pol]: pol -> pol :=[q:pol]Cases p of
      vpol  => q
    | (cpol m1 p1) => (cpol m1 (suma_app p1 q))
    end.
```

El tipo lista (de objetos de un tipo α dado) forma un monoide con la concatenación, siendo el neutro la lista vacía. En particular para el tipo *pol*, y la función *suma_app*, comprobamos que el tipo *vpol* es el neutro de esta estructura.

Traducción en **Coq**:

```
Lemma vpol_suma_leibn: (p:pol) (suma_app vpol p)=p.

Lemma suma_vpol_leibn: (p:pol) (suma_app p vpol)=p.
□
```

6.2.2. Igualdad de polinomios

Cuando comenzamos a calcular con polinomios en varias variables nos encontramos con la siguiente dificultad: ¿calculamos con $(x^2y + xy)$, $(xy + x^2y)$, $(x^2y + 3xy - 2xy)$ ó con $(0x^3y^2 + x^2y + xy)$? Matemáticamente, en todos estos casos, estamos tratando con el mismo objeto. Para evitar estos problemas formalizamos una relación de igualdad de polinomios con la ayuda de la función *suma_app* que se ha implementado anteriormente.

Nota: Utilizamos la notación $(m : p)$, en donde $m \in M_K$ y $p \in K[X]$, para denotar un polinomio cuyo monomio (de cabeza) es m y p es el resto de monomios (cola).

Definición 6.2.1. *La relación de igualdad de polinomios* $(=_p)$ **(igualdad extensional)** *es aquella que verifica:*

1. *la propiedad reflexiva*
2. *la propiedad simétrica*
3. *la propiedad transitiva*
4. $\forall p, q \in K[X];\ \forall m \in M_K;\ (m : (p +_p q)) =_p p +_p (m : q)$
5. $\forall p, q, p', q' \in K[X];\ (p =_p p') \wedge (q =_p q') \Rightarrow (p +_p q) =_p (p' +_p q')$
6. $\forall k, k' \in K;\ \forall t \in T;\ \forall p \in K[X];\ ((k, t) : ((k', t) : p)) =_p (((k +_K k'), t) : p)$
7. $\forall m \in M_K;\ \forall p \in K[X];\ (z\_monom\ m) \Rightarrow (m : p) =_p p$

Las reglas anteriores se formalizan en **Coq** mediante los constructores de la definición inductiva del predicado *equpol*, siendo ésta la menor relación de equivalencia que verifica las siete condiciones anteriores.

Traducción en **Coq**:

```
Inductive equpol: pol->pol->Prop :=
   equpol_refl: (p:pol)(equpol p p)
 | equpol_sym: (p,q:pol)(equpol p q)->(equpol q p)
 | equpol_tran: (p,q,r:pol)(equpol p q)->(equpol q r)->(equpol p r)
 | equpol_cm: (m:monom)(p,q:pol)
         (equpol (cpol m (suma_app p q)) (suma_app p (cpol m q)))
 | equpol_ss: (p,p',q,q':pol)(equpol p p')->(equpol q q')->
               (equpol (suma_app p q) (suma_app p' q'))
 | equpol_smt: (k,k':K)(t:term)(p:pol)
    (equpol (cpol (k,t) (cpol (k',t) p)) (cpol ((plusK k k'),t) p))
 | equpol_elim: (m:monom)(p:pol)(z_monom m)->(equpol (cpol m p) p).
```

Algunas propiedades de la relación *equpol*, muy útiles en todas las pruebas referidas a operaciones de polinomios, se ven a continuación. De algunas se incluye el código de la prueba.

Teorema 6.2.1.

$$\forall m \in M_K;\ \forall p,q \in K[X],\ (p =_p q) \Rightarrow (m : p) =_p (m : q)$$

Prueba: Para demostrarlo utilizamos la táctica **Change**. Esta táctica implementa en Coq la regla **Conv** que dice que dos elementos t_1 y t_2 son convertibles (o equivalentes) si, y sólo si existe un elemento u tal que t_1 y t_2 se reducen a él mediante las reglas de β-reducción, ι-reducción y δ-reducción. En nuestro caso lo que hace es sustituir el constructor *cpol* por *suma_app*, tomando un monomio m como un polinomio de la forma (*cpol m vpol*) lo que permite aplicar el constructor *equpol_ss* de la igualdad de polinomios.

Traducción en **Coq**:

```
Lemma equpol_cpol: (m:monom)(p,q:pol)(equpol p q)->
                     (equpol (cpol m p) (cpol m q)).

Intros m p q H.
Change (equpol (suma_app (cpol m vpol) p) (suma_app (cpol m vpol) q)); Auto.
□
```

Otro resultado que necesitamos es el siguiente:

Teorema 6.2.2.

$$\forall m \in M_K;\ \forall p \in K[X],\ ((-m) : (m : p)) =_p p$$

Prueba: La prueba se consigue mediante dos de los constructores *equpol_smt* y *equpol_elim* de la igualdad de polinomios antes definida.

Traducción en **Coq**:

```
Lemma sym_mon : (m:monom)(p:pol) (equpol (cpol (monom_op m) (cpol m p)) p).

Intros; Elim m; Intros.
Unfold monom_op.
Simpl.
Apply equpol_tran with (cpol ((plusK (opK y) y),y0) p); Auto.
□
```

Necesitamos lemas que permitan simplificar las expresiones formales de polinomios; por ejemplo,

Teorema 6.2.3.

$$\forall m \in M_K;\ \forall p, q \in K[X];\ (m : p) =_p (m : q) \Rightarrow (p =_p q)$$

Prueba: Para la prueba utilizamos los lemas *equpol_cpol* y *sym_mon* ya demostrados; mediante la táctica **Generalize** reproducimos en Coq el esquema siguiente:

$$(m : p) =_p (m : q) \Rightarrow (-m : (m : p)) =_p (-m : (m : q)) \Rightarrow (p =_p q)$$

Traducción en **Coq**:

```
Lemma eq_cpol: (m:monom)(p,q:pol)(equpol (cpol m p) (cpol m q))->(equpol p q).

Intros.
Generalize (equpol_cpol (monom_op m) (cpol m p) (cpol m q)); Intros.
Generalize (H0 H); Intro.
Generalize (sym_mon m p) ; Intro.
Generalize (sym_mon m q) ; Intro.
Generalize (equpol_sym (cpol (monom_op m) (cpol m p)) p H2); Intro.
Generalize (H0 H) ; Clear  H0; Intros.
Generalize (equpol_tran p (cpol (monom_op m) (cpol m p)) q H4).
Intros.
Apply H5.
Apply equpol_tran with (cpol (monom_op m) (cpol m q)); Trivial.
□
```

Un polinomio puede constar de un solo monomio como hemos visto anteriormente; por ello necesitamos probar que la igualdad de polinomios *equpol* extiende a la igualdad de monomios *equmon*.

Teorema 6.2.4.

$$\forall m, m' \in M_K;\ m =_M m' \Rightarrow (m : vpol) =_p (m' : vpol)$$

Prueba: En primer lugar sustituimos por su valor los monomios, para luego hacer la prueba por casos según la formalización de *equmon*. El primer caso, donde los coeficientes son cero, es trivial aplicando el constructor *equpol_elim*; el segundo caso se resuelve utilizando la transitividad de *equpol* y las propiedades de *suma_app* así como las de la relación de igualdad definida en el cuerpo de coeficientes.

Traducción en **Coq**:

```
Lemma equmon_cons : (m,m':monom)(equmon m m')->
                      (equpol (cpol m vpol)(cpol m' vpol)).
□
```

Mediante el teorema anterior, probamos que, si dos monomios son iguales, entonces añadiéndole el mismo polinomio obtenemos polinomios iguales.

Teorema 6.2.5.

$$\forall m, m' \in M_K;\ \forall p \in K[X];\ m =_M m' \Rightarrow (m : p) =_p (m' : p)$$

Traducción en **Coq**:

```
Lemma eq_mon_cpol: (m,m':monom)(equmon m m')->
        (p:pol)(equpol (cpol m p) (cpol m' p)).
```
□

Un caso particular del resultado anterior, utilizado más adelante, es el siguiente:

Traducción en **Coq**:

```
Lemma mon_K_cpol: (a,b:K)(eqK a b)->
   (p:pol)(y:term)(equpol (cpol (a,y) p) (cpol (b,y) p)).
```
□

Tenemos ahora todos los ingredientes para demostrar que si dos monomios y dos polinomios son iguales, con sus respectivas igualdades, entonces también lo son los polinomios formados a partir de ellos por concatenación. La prueba es consecuencia de los resultados anteriores y del constructor *equpol_ss*.

Teorema 6.2.6.

$$\forall m, m' \in M_K;\ \forall p, p' \in K[X];$$
$$(m =_M m') \wedge (p =_p p') \Rightarrow (m : p) =_p (m' : p')$$

Traducción en **Coq**:

```
Lemma equpol_cpol2: (m,m':monom)(equmon m m')->(p,p':pol)(equpol p p')->
                    (equpol (cpol m p) (cpol m' p')).
```
□

Una propiedad importante de la igualdad de polinomios es que si dos polinomios son iguales y uno de ellos no es igual al polinomio *vpol*, el otro tampoco.

Teorema 6.2.7.

$$\forall p, q \in K[X]\ (p =_p q) \wedge (p \neq_p vpol) \Rightarrow (q \neq_p vpol)$$

Prueba: Se prueba explicitando la definición de **not**, denotado en Coq por $(\sim)$, y aplicando la transitividad de *equpol* con uno de los polinomios.

Traducción en **Coq**:

```
Lemma eq_nvpol:(p,q:pol)(equpol p q)->~(equpol p vpol)->~(equpol q vpol).
```
□

6.2.3. Propiedades básicas de la adición

La primera propiedad que verificamos y que será utilizada muy a menudo es que la función *suma_app* es interna en polinomios de n variables.

Prueba: Se demuestra fácilmente por recurrencia sobre la definición de polinomio y aplicando los constructores de *full_term_pol*, predicado que define los polinomios de n variables definido en la sección 6.1.

Traducción en **Coq**:

```
Lemma full_term_suma_app: (p:pol)(full_term_pol p)->
        (q:pol)(full_term_pol q)->(full_term_pol (suma_app p q)).
```
□

Como hemos hecho con las representaciones de términos y monomios, una vez que tenemos representados los polinomios y formalizada una operación sobre ellos, comprobamos las propiedades algebraicas que verifica esa operación.

Los dos resultados siguientes son consecuencia de la definición de *suma_app*. El segundo de ellos es la prueba de que *vpol* es el neutro de la operación.

Traducción en **Coq**:

```
Theorem asoc_mon_pol : (m:monom)(p,q:pol)(equpol
  (suma_app (suma_app (cpol m vpol) p) q)
            (suma_app (cpol m vpol) (suma_app p q))).

Theorem vpol_suma : (q:pol) (equpol (suma_app vpol q) q).

Theorem suma_vpol : (q:pol) (equpol (suma_app q vpol) q).
```
□

Probamos a continuación dos de las propiedades algebraicas más útiles de la función suma de polinomios, utilizando la igualdad de polinomios *equpol* definida anteriormente.

Teorema 6.2.8. *La suma de polinomios verifica las propiedades conmutativa y asociativa.*

$$\forall p, q \in K[X];\ p +_p q =_p q +_p p$$

$$\forall p, q, r \in K[X];\ (p +_p q) +_p r =_p p +_p (q +_p r)$$

Prueba: Ambas demostraciones se hacen por recurrencia sobre el primer polinomio de la suma; las submetas obtenidas se resuelven mediante los resultados anteriores, la hipótesis de recursión obtenida y los constructores *equpol_crn* y *equpol_tran* de la igualdad.

Traducción en **Coq**:

```
Theorem conmt_suma : (p,q:pol)(equpol (suma_app p q) (suma_app q p)).


Theorem asoc_suma: (p,q,r:pol)(equpol (suma_app (suma_app p q) r)
                                      (suma_app p (suma_app q r))).
```
□

Un resultado técnico que necesitamos para evitar demostraciones muy largas y repetitivas en pruebas posteriores, es el siguiente.

Teorema 6.2.9.

$$\forall m, n \in M_K;\ \forall p, q \in K[X];\ (m : p) +_p (n : q) =_p (m : (n : (p +_p q)))$$

Traducción en **Coq**:

```
Lemma suma_cpol: (m,n:monom)(p,q:pol)(equpol
  (suma_app (cpol m p) (cpol n q)) (cpol m (cpol n (suma_app p q)))).
```
□

Necesitamos, por razones de operatividad, una propiedad que combine las propiedades asociativa y conmutativa de la función *suma_app*.

Corolario 6.2.1.

$$\forall p, q, r \in K[X];\ p +_p (q +_p r) =_p q +_p (p +_p r)$$

Prueba: Utilizando el constructor *equpol_tran* reproducimos en el sistema la sucesión de igualdades siguientes:

$$(p +_p (q +_p r)) =_p ((p +_p q) +_p r) =_p ((q +_p p) +_p r) =_p (q +_p (p +_p r))$$

Traducción en **Coq**:

```
Lemma permut_suma_app:(p,q,r:pol)(equpol
   (suma_app p (suma_app q r)) (suma_app q (suma_app p r))).
```

□

Un caso particular del constructor *equpol_ss* son los lemas de monotonía que formalizamos a continuación:

Traducción en **Coq**:

```
Lemma monot_suma: (p,q,r:pol)(equpol p q)->
        (equpol (suma_app p r) (suma_app q r)).


Lemma monot_suma_l: (p,q,r:pol)(equpol p q)->
        (equpol (suma_app r p) (suma_app r q)).
```

□

Con el lema siguiente podemos intercambiar la función *suma_app* y el constructor de polinomios *cpol* utilizando la igualdad del sistema.

Traducción en **Coq**:

```
Lemma sumcpol_l: (m:monom)(p,q:pol)
   ((cpol m (suma_app p q))=(suma_app (cpol m p) q)).
```

□

Después de comprobar algunas propiedades algebraicas de la operación suma, introducimos la noción de polinomio opuesto a uno dado.

Definición 6.2.2. *Dado un polinomio p, el polinomio $(-p)$ opuesto a p, formalizado como (pol_opp p), es aquel que tiene los mismos términos y como coeficientes los opuestos de los coeficientes de p.*

Traducción en **Coq**:

```
Fixpoint pol_opp [p:pol]: pol:=Cases p of
      vpol           => vpol
    | (cpol (c,t) p) => (cpol ((opK c),t) (pol_opp p))
    end.
```

Verificamos que la operación opuesto es interna en polinomios de n variables. La prueba es análoga a la hecha para la suma de polinomios.

Traducción en **Coq**:

```
Lemma full_term_opp: (p:pol)(full_term_pol p)->(full_term_pol (pol_opp p)).
```

□

Una vez definido el polinomio opuesto a uno dado, nos interesa probar aquellas propiedades relacionadas con este polinomio, que juegan un papel relevante en el anillo de polinomios que intentamos construir.

Teorema 6.2.10.

$$\forall p \in K[X];\ p +_p (-p) =_p vpol$$

Prueba: La demostración se hace por inducción sobre el polinomio p; la primera submeta obtenida es trivial por la manera de definir la función *pol_opp*; la segunda se alcanza, una vez sustituido por su valor el monomio de cabeza de p, utilizando los resultados que relacionan el constructor *cpol* y la función *suma_app* mediante el constructor *equpol_tran*, así como propiedades generales de los elementos de un cuerpo.

Traducción en **Coq**:

```
Lemma suma_opp: (p:pol)(equpol (suma_app p (pol_opp p)) vpol).
```

□

También demostramos el resultado:

Teorema 6.2.11.

$$\forall m \in M_K;\ (m : (-(m : vpol))) =_p vpol$$

Traducción en **Coq**:

```
Lemma sym_mon1: (m:monom)(equpol (cpol m (pol_opp (cpol m vpol))) vpol).
```

□

Los cinco teoremas expuestos a continuación, se demuestran utilizando los constructores de *equpol*, en especial se utiliza el constructor *equpol_tran* para estar en condiciones de poder aplicar resultados sobre dicho predicado. No se detallarán las pruebas, a excepción de la primera, debido a que no aportan novedades con relación a pruebas anteriores.

Teorema 6.2.12. $\forall p, q \in K[X];\ p +_p q =_p vpol \Leftrightarrow p =_p -q$

Traducción en **Coq**:

```
Lemma sum_vpol_opp: (p,q:pol)(equpol (suma_app p q) vpol)->
                                  (equpol p (pol_opp q)).

Intros.
Apply equpol_tran with (suma_app p (suma_app q (pol_opp q))).
Apply equpol_tran with (suma_app p vpol); Auto.
Apply equpol_tran with (suma_app (suma_app p q) (pol_opp q)).
Auto.
Apply equpol_tran with (suma_app vpol (pol_opp q)).
Auto.
Auto.

Lemma sum_vpol_opp_inv: (p,q:pol)(equpol p (pol_opp q))->
                           (equpol (suma_app p q) vpol).
Intros.
Apply equpol_tran with (suma_app (pol_opp q) q).
Auto.
Apply equpol_tran with (suma_app q (pol_opp q)); Auto.
```

□

Corolario 6.2.2.

$$\forall p, q \in K[X];\ p =_p q \Rightarrow -p =_p -q$$

Traducción en **Coq**:

```
Lemma opp_comp: (f,g:pol)(equpol f g)->(equpol (pol_opp f) (pol_opp g)).
```
□

Teorema 6.2.13.

$$\forall p \in K[X];\ p =_p vpol \Leftrightarrow -p =_p vpol$$

Traducción en **Coq**:

```
Lemma opp_vpol: (p:pol)(equpol p vpol)->(equpol (pol_opp p) vpol).

Lemma eq_pol_opp_no_zero: (f:pol)(equpol (pol_opp f) vpol)->(equpol f vpol).
```
□

Teorema 6.2.14.

$$\forall p, q \in K[X];\ -(p +_p q) =_p (-p) +_p (-q)$$

Traducción en **Coq**:

```
Lemma mult_sum_opp3: (f,g:pol)(equpol (pol_opp (suma_app f g))
   (suma_app (pol_opp f) (pol_opp g))).
```
□

Teorema 6.2.15.

$$\forall p \in K[X];\ (p \neq_p vpol) \Rightarrow (-p \neq_p vpol)$$

Traducción en **Coq**:

```
Lemma no_eq_no_vpol_opp: (f:pol)~(equpol f vpol)->~(equpol (pol_opp f) vpol).
```
□

6.3. Producto de polinomios: Propiedades

Dada la formalización del tipo *pol* y dado que la multiplicación de polinomios distribuye sobre la suma, es indispensable la implementación del producto de un monomio por un polinomio antes de definir el producto "matemático" de polinomios.

El producto de un monomio por un polinomio p $(.{}_M)$ es otro polinomio que contiene todos los monomios que resultan de multiplicar el monomio dado por cada uno de los monomios del polinomio p.

```
Fixpoint mult_m [p:pol] : monom -> pol :=[m:monom]Cases p of
      vpol  =>  vpol
    | (cpol m1 p1) => (cpol (mult_mon m m1) (mult_m p1 m))
    end.
```

Definimos ahora, recursivamente, el producto de polinomios ($._p$) partir de la función anterior *mult_m*.

```
Fixpoint mult_p [p:pol] : pol -> pol :=[q:pol]Cases p of
      vpol    =>   vpol
    | (cpol m1 p1) => (suma_app (mult_m q m1) (mult_p p1 q))
    end.
```

Probamos que al multiplicar el polinomio *vpol* por cualquier monomio el resultado, como era de esperar, es *vpol*.

Teorema 6.3.1.

$$\forall m \in M_K;\ vpol\ ._M\ m =_p vpol$$

```
Theorem mult_m_vpol: (m:monom) (equpol (mult_m vpol m) vpol).
```
□

Asimismo demostramos que *vpol* se comporta como el elemento cero para el producto de polinomios.

Teorema 6.3.2.

$$\forall p \in K[X];\ p\ ._p\ vpol =_p vpol$$

Traducción en **Coq**:

```
Theorem mult_vpol: (q:pol)(equpol (mult_p q vpol) vpol).
```
□

Se prueba ahora que la definición dada del producto de polinomios es coherente con nuestra igualdad extensional de polinomios *equpol*.

Teorema 6.3.3.

$$\forall m \in M_K;\ \forall p, p' \in K[X];\ p\ ._p\ (m : p') =_p p\ ._M\ m +_p p\ ._p\ p'$$

Prueba: Por recurrencia sobre el polinomio p. El caso de *vpol* se resuelve mediante la táctica **Auto**; la segunda meta, una vez hecha la simplificación mediante **Simpl** queda de la forma siguiente:

$((m_0.m) : (p'._M m_0 +_p p_0._p(m : p'))) =_p ((m.m_0) : (p_0._M m +_p (p'._M m_0 +_p p_0._p p')))))$

si aplicamos *equpol_cpol2* (teorema 6.2.6), la nueva meta es:

$$(p'._M m_0 +_p p_0._p(m : p')) =_p (p_0._M m +_p (p'._M m_0 +_p p_0._p p'))$$

Ésta se prueba utilizando la transitividad de la igualdad de polinomios para formalizar el esquema siguiente:

$$p_0._M m +_p (p'._M m_0 +_p p_0._p p') =_p \quad \text{Asociativa de } +_p$$

$$(p_0._M m +_p p'._M m_0) +_p p_0._p p' =_p \quad \text{Conmutativa de } +_p$$

$$(p'._M m_0 +_p p_0._M m) +_p p_0._p p' =_p \quad \text{Asociativa de } +_p$$

$$p'._M m_0 +_p (p_0._M m +_p p_0._p p') =_p \quad \text{Lemma } \textit{mult\_p\_cpol}$$

$$p'._M m_0 +_p p_0._p(m : p')$$

Traducción en **Coq**:

```
Lemma mult_p_cpol: (m:monom)(p,p':pol)(equpol
   (mult_p p (cpol m p')) (suma_app (mult_m p m) (mult_p p p'))).

Intros; Elim p; Auto; Intros.
Simpl.
Apply equpol_cpol2; Auto.
Apply equpol_tran with
 (suma_app (suma_app (mult_m p0 m) (mult_m p' m0)) (mult_p p0 p')); Auto.
Apply equpol_tran with
 (suma_app (suma_app (mult_m p' m0) (mult_m p0 m)) (mult_p p0 p')); Auto.
Apply equpol_tran with
 (suma_app (mult_m p' m0) (suma_app (mult_m p0 m) (mult_p p0 p'))); Auto.
```
□

Antes de verificar ciertas propiedades del producto de polinomios tales como la asociatividad, conmutatividad, distributividad del producto respecto de la suma y algunas otras, probamos dichas propiedades para el producto de polinomios por monomios lo cual simplificará el trabajo en las demostraciones de las propiedades antes citadas.

Los tres próximos resultados se demuestran de manera similar, por recurrencia sobre la definición de *pol* y aplicando resultados anteriores sobre *equpol*.

Teorema 6.3.4.

$$\forall m \in M_K;\ \forall p, q \in K[X];\ (p +_p q) \cdot_M m =_p p \cdot_M m +_p q \cdot_M m$$

Traducción en **Coq**:

```
Lemma mult_m_distr: (p,q:pol)(m:monom)(equpol
   (mult_m (suma_app p q) m) (suma_app (mult_m p m) (mult_m q m))).
```
□

Teorema 6.3.5.

$$\forall m, n \in M_K;\ \forall p \in K[X];\ (p \cdot_M m) \cdot_M n =_p p \cdot_M (m.n)$$

Traducción en **Coq**:

```
Lemma mult_m_asoc: (m,n:monom)(p:pol)(equpol
   (mult_m (mult_m p m) n) (mult_m p (mult_mon m n))).
```
□

Teorema 6.3.6.

$$\forall m_1, m_2 \in M_K;\ \forall p \in K[X];\ m_1 =_M m_2 \Rightarrow p \cdot_M m_1 =_p p \cdot_M m_2$$

Traducción en **Coq**:

```
Lemma mult_pm_comp_2: (m1,m2:monom)(p:pol)(equmon m1 m2)->
       (equpol (mult_m p m1) (mult_m p m2)).
```
□

Teorema 6.3.7.

$$\forall m \in M_K;\ \forall p, q \in K[X];\ (p \cdot_p q) \cdot_M m =_p (p \cdot_M m) \cdot_p q$$

Prueba: Aplicando los lemas anteriores y el constructor *equpol_ss* de la igualdad de polinomios.

Traducción en **Coq**:

```
Lemma mult_p_m_asoc: (m:monom)(p,q:pol)
 (equpol (mult_m (mult_p p q) m) (mult_p (mult_m p m) q)).
□
```

Con los dos teoremas siguientes se comprueba que la función *mult_m* conserva la igualdad extensional sobre polinomios.

Teorema 6.3.8.

$$\forall m \in M_K;\ \forall p, q \in K[X];\ p =_p q \Rightarrow p \cdot_M m =_p q \cdot_M m$$

Prueba: La demostración se hace por recurrencia estructural sobre la definición de *equpol*; obtenemos siete metas, una por cada uno de los constructores de la igualdad. Las dos primeras las resuelve el sistema automáticamente y las restantes se resuelven mediante la transitividad de la igualdad y lemas relativos[1] a la función suma e igualdad de polimomios de la sección anterior.

Traducción en **Coq**:

```
Lemma mult_pm_comp: (m:monom)(p1,p2:pol)(equpol p1 p2)->
                    (equpol (mult_m p1 m) (mult_m p2 m)).
□
```

Teorema 6.3.9. $\forall m_1, m_2 \in M_K;\ \forall p_1, p_2 \in K[X]$, *se verifica que:*
$(p_1 =_p p_2) \wedge (m_1 =_M m_2) \Rightarrow p_1 \cdot_M m_1 =_p p_2 \cdot_M m_2$

Prueba: Se aplica inducción sobre los polinomios, las submetas generadas se deducen inmediatamente de los dos lemas anteriores.

Traducción en **Coq**:

```
Lemma mult_pm_comp_ss: (p1,p2:pol)(equpol p1 p2)->
   (m1,m2:monom)(equmon m1 m2)->(equpol (mult_m p1 m1) (mult_m p2 m2)).
□
```

Teorema 6.3.10. *El producto de polinomios (por la derecha) conserva la igualdad extensional de polinomios.*

$$\forall p, p', r \in K[X];\ p =_p p' \Rightarrow r \cdot_p p =_p r \cdot_p p'$$

Prueba: Consecuencia directa de los lemas *mult_pm_comp* y *mult_pm_comp_2*. Se utiliza también la β-reducción aplicada mediante la táctica **Simpl**.

Traducción en **Coq**:

```
Lemma mult_p_comp: (p,p',r:pol)(equpol p p')->
                     (equpol (mult_p r p) (mult_p r p')).
□
```

Ahora tenemos ya, los resultados necesarios para probar las propiedades algebraicas más importantes del producto de polinomios.

[1] Este tipo de lemas suelen quedar, implícitos en las demostraciones, si bien están en el corazón de las mismas.

Teorema 6.3.11. *El producto de polinomios verifica la propiedad conmutativa.*

$$\forall p, p' \in K[X];\ p \cdot_p p' =_p p' \cdot_p p$$

Prueba: La demostración se hace por recurrencia sobre los polinomios en cuestión, las tres primeras metas las resuelve el sistema automáticamente y la cuarta es consecuencia del lema *mult_p_cpol* y de la hipótesis del contexto obtenida al hacer la recursión.

Traducción en **Coq**:

```
Lemma mult_p_conm: (p,p':pol)(equpol (mult_p p p') (mult_p p' p)).
```
□

Utilizando la conmutatividad del producto probamos fácilmente que el producto de polinomios por la izquierda también conserva la igualdad extensional de polinomios.

Corolario 6.3.1.

$$\forall p, p', r \in K[X];\ p =_p p' \Rightarrow p \cdot_p r =_p p' \cdot_p r$$

Traducción en **Coq**:

```
Lemma mult_p_comp_r: (p,p',r:pol)(equpol p p')->
                     (equpol (mult_p p r) (mult_p p' r)).
```
□

Teorema 6.3.12. *El producto de polinomios verifica la propiedad distributiva por la derecha respecto de la suma.*

$$\forall p, q, r \in K[X];\ (p +_p q) \cdot_p r =_p p \cdot_p r +_p q \cdot_p r$$

Prueba: Aplicando inducción sobre el polinomio r, la primera submeta es trivial y la segunda se prueba utilizando lemas anteriores, en especial los teoremas 6.3.3 y 6.3.4. Hay que hacer notar que utilizamos el comando **Clear** para eliminar del contexto de las hipótesis algunas de las ya utilizadas y no necesarias en la meta en curso; también se usa la táctica **Generalize** para introducir en el contexto otras hipótesis que se necesitan para la prueba y que se obtienen de resultados ya probados.

Traducción en **Coq**:

```
Lemma distr_mult_p: (p,q,r:pol)(equpol (mult_p (suma_app p q) r)
   (suma_app (mult_p p r) (mult_p q r))).
```
□

La distributividad por la izquierda es inmediata utilizando la conmutatividad y la distributividad por la derecha.

Corolario 6.3.2.

$$\forall p, q, r \in K[X];\ p \cdot_p (q +_p r) =_p p \cdot_p q +_p p \cdot_p r$$

Traducción en **Coq**:

```
Lemma distr_mult_p_r: (p,q,r:pol)(equpol (mult_p p (suma_app q r))
      (suma_app (mult_p p q) (mult_p p r))).
```

□

Teorema 6.3.13. *El producto de polinomios verifica la propiedad asociativa.*

$$\forall p, q, r \in K[X];\ p \cdot_p (q \cdot_p r) =_p (p \cdot_p q) \cdot_p r$$

Prueba: Es similar a la de la conmutatividad, excepto que en este caso utilizamos la propiedad distributiva del producto de monomios respecto a polinomios.

Traducción en **Coq**:

```
Lemma mult_p_assoc: (p,q,r:pol)(equpol (mult_p p (mult_p q r))
                                  (mult_p (mult_p p q) r)).
```

□

Para demostrar la existencia del elemento neutro en el producto de polinomios, necesitamos primero introducir el monomio $1.(x_1^0.x_2^0 \ldots x_n^0)$ y comprobar su funcionamiento con la función *mult_m*.

Teorema 6.3.14.

$$\forall p \in K[x_1, \ldots, x_n],\ p =_p p \cdot_M (1.(x_1^0.x_2^0 \ldots x_n^0))$$

Prueba: Por recurrencia sobre p, el primer caso es trivial; para demostrar el caso de $(cpol\ m\ p)$, hacemos explícito el monomio m e invertimos la hipótesis del contexto $(full\_term\_pol\ (cpol\ (y, y0)\ p))$ para obtener de ella la información necesaria para aplicar resultados (como por ejemplo el teorema 3.2.5) obtenidos en la formalización de los términos. Se utiliza también la hipótesis de recurrencia obtenida con p y el constructor *equpol_tran*, para transformar las metas y adecuarlas a resultados ya probados, relativos a la igualdad extensional de polinomios.

Traducción en **Coq**:

```
Lemma  neut_mult_m: (p:pol)(full_term_pol p)->
          (equpol p (mult_m p (unK,(n_term_0 n)))).
```

□

De este resultado, podemos deducir fácilmente el teorema siguiente.

Teorema 6.3.15. *Existe elemento neutro en el producto de polinomios.*

$$\forall p \in K[x_1, \ldots, x_n],\ p \cdot_p ((1.(x_1^0.x_2^0 \ldots x_n^0)) : vpol) =_p p$$

Prueba: Considerando el polinomio con un único monomio $(1.(x_1^0.x_2^0 \ldots x_n^0))$ y aplicando el lema anterior. En esta prueba debemos utilizar la táctica **Generalize** para introducir abstracciones y explicitar en la meta las variables que utilizamos.

Traducción en **Coq**:

```
Lemma neut_mult_p: (p:pol)(full_term_pol p)->(equpol
         (mult_p p (cpol (unK,(n_term_0 n)) vpol)) p).

Induction p; Auto.
Intro m; Elim m; Intros.
Simpl.
Inversion H0.
Generalize (H H5); Intro.
EApply (equpol_tran (cpol ((multK y unK),(term_mult y0 (n_term_0 n)))
           (mult_p p0 (cpol (unK,(n_term_0 n)) vpol)))
         (cpol ((multK y unK),(term_mult y0 (n_term_0 n))) p0)).
Auto.
Generalize (term_mult_full_nulo y0 n H3); Intro.
Rewrite H7.
Generalize (neut_multK y); Intro; Auto.
□
```

Una propiedad particularmente interesante, por su utilización, relativa al producto de polinomios es la regla de los signos.

Teorema 6.3.16.

$$\forall p, q \in K[X],\ p \cdot_p (-q) =_p -(p \cdot_p q)$$

Prueba: Formalizando en Coq la siguiente secuencia de igualdades

$$vpol =_p p.{}_p vpol =_p p.{}_p(-q +_p q) =_p p.{}_p(-q) +_p p.{}_p q$$

Traducción en **Coq**:

```
Lemma regl_sig: (p,q:pol)
                  (equpol (mult_p p (pol_opp q)) (pol_opp (mult_p p q))).
□
```

Un caso particular del lema anterior es el siguiente.

Corolario 6.3.3.

$$\forall p \in K[x_1, \ldots, x_n],\ p \cdot_p ((-1.(x_1^0.x_2^0 \ldots x_n^0)) : vpol) =_p -p$$

Traducción en **Coq**:

```
Lemma mult_neg_unK: (p:pol)(full_term_pol p)->
   (equpol (mult_p p (cpol ((opK unK),(n_term_0 n)) vpol)) (pol_opp p)).
□
```

Cada uno de los cinco resultados siguientes formulan propiedades elementales del producto de polinomios bajo formas más confortables para su utilización en resultados posteriores.

Todos se demuestran de manera similar; la principal dificultad es hallar la formulación adecuada para utilizar directamente resultados anteriores. Se incluyen algunas demostraciones como muestra, remitiendo de nuevo al repositorio del código para los detalles.

El primer resultado *prueba la relación existente entre la operación producto de un polinomio p por un monomio m y el producto de p por m considerado como un polinomio.*

Teorema 6.3.17.

$$\forall m \in M_K;\ \forall p \in K[X];\ p \cdot_M m =_p p \cdot_p (m : vpol)$$

Traducción en **Coq**:

```
Lemma mult_p_m: (p:pol)(m:monom)(equpol (mult_m p m) (mult_p p (cpol m vpol))).

Intros.
Apply equpol_tran with (suma_app (mult_m p m) (mult_p p vpol)); Auto.
Apply equpol_tran with (suma_app (mult_m p m) vpol); Auto.
□
```

Teorema 6.3.18. $\forall m \in M_K;\ \forall p \in K[X];\ m =_M 0_M \Rightarrow p \cdot_M m =_p vpol$

Traducción en **Coq**:

```
Lemma mult_m_z_monom: (p:pol)(m:monom)(z_monom m)->(equpol (mult_m p m) vpol).

Induction p; Auto.
Intros; Simpl.
Apply equpol_tran with (mult_m p0 m0); Auto.
□
```

Corolario 6.3.4. $\forall m \in M_K;\ \forall p \in K[X];\ -(p \cdot_M m) =_p p \cdot_p (-(m : vpol))$

Traducción en **Coq**:

```
Lemma mul_opp_monom: (p:pol)(m:monom)(equpol (pol_opp (mult_m p m))
   (mult_p p (pol_opp (cpol m vpol)))).
□
```

Corolario 6.3.5. $\forall m \in M_K;\ \forall p \in K[X];\ p \cdot_M m =_p -(p \cdot_M (-m))$

Traducción en **Coq**:

```
Lemma mult_opp_monom2:(p:pol)(m:monom)(equpol (mult_m p m)
   (pol_opp (mult_m p (monom_op m)))).
□
```

Teorema 6.3.19.

$$\forall p \in K[X];\ \forall t \in T;\ \forall c, c' \in K;\ p \cdot_M (c,t) +_p p \cdot_M (c',t) =_p p \cdot_M ((c+c'),t)$$

Traducción en **Coq**:

```
Lemma distr_mult_m_monom: (p:pol)(c,c':K)(t:term)(equpol
   (suma_app (mult_m p (c,t)) (mult_m p (c',t))) (mult_m p ((plusK c c'),t))).
□
```

Todas las demostraciones anteriores se hacen aplicando la transitividad de la igualdad extensional de polinomios. Sólo en un caso debemos aplicar la recurrencia sobre la definición de *pol*.

Al igual que hemos hecho con *suma_app*, comprobamos que las operaciones *mult_m* y *mult_p* conservan los polinomios de n variables. La prueba en ambos casos se hace por inducción sobre p, aplicando el constructor *full_term_pol_cons* y obteniendo la información necesaria de las hipótesis mediante la táctica de inversión.

Traducción en **Coq**:

```
Lemma full_mult_m: (c:K)(t:term)(p:pol)(full n t)->(full_term_pol p)->
                    (full_term_pol (mult_m p (c,t))).


Lemma full_term_mult_p: (p:pol)(full_term_pol p)->(q:pol)(full_term_pol q)->
                    (full_term_pol (mult_p p q)).
```

□

6.3.1. Divisores de cero en el producto de polinomios por monomios

Como podremos constatar posteriormente [veáse por ejemplo la prueba de una condición suficiente para la reducción de polinomios (teorema 9.2.1)] se necesitará varias veces el siguiente resultado.

Teorema 6.3.20.

$$\forall p \in K[x_1, \ldots, x_n],\ \forall t \in T^n,\ \forall c \in K;$$
$$(p \neq_p vpol) \wedge (c \neq_K O_K) \Rightarrow (p \cdot_M (c,t) \neq_p vpol)$$

Traducción en **Coq**:

```
Lemma contr_mult_nz_monom: (p:pol)(full_term_pol p)->(~(equpol p vpol))->
   (c:K)(~(eqK c OK))->(t:term)(full n t)->(~(equpol (mult_m p (c,t)) vpol)).
```

Nota: En general en lugar de demostrar $(\neg A \rightarrow \neg B)$, es mejor probar el contrarrecíproco $(B \rightarrow A)$. Estas dos proposiciones son equivalentes en lógica clásica, pero no en lógica intuicionista (la utilizada por Coq). Si partimos de $(B \rightarrow A)$ podemos demostrar $(\neg A \rightarrow \neg B)$, aunque el recíproco no es cierto en lógica intuicionista. $(B \rightarrow A)$ es, a veces, un poco más difícil de probar, pero es un resultado más general: un teorema Teor: $(B \rightarrow A) \rightarrow (\neg A \rightarrow \neg B)$, se puede probar en Coq mediante la secuencia: **Red**; **Intros** *H noA hipB*; **Apply** *noA*; **Apply** *H*; **Trivial**.

Para poder demostrar este lema, lo reformulamos de manera más sencilla según se indica en la nota anterior. En terminología matemática diríamos que "el producto de un polinomio por un monomio no tiene divisores de cero".

Teorema 6.3.21.

$$\forall p \in K[x_1, \ldots, x_n],\ \forall m \in M_{K,n};\ (m \neq_M 0_M) \wedge (p \cdot_M m =_p vpol) \Rightarrow p =_p vpol$$

Traducción en **Coq**:

```
Lemma mult_nz_monom: (p:pol)(full_term_pol p)->(m:monom)(full_mon n m)->
   ~(z_monom m)->(equpol (mult_m p m) vpol)->(equpol p vpol).
```

Para poder demostrar este resultado es imprescindible introducir una operación, que podemos denominar "división de un polinomio por un monomio", de forma análoga a como hemos definido la multiplicación de un polinomio por un monomio.

Definición 6.3.1. *Dado un polinomio $p \in K[X]$ y un monomio $m \in M_K$, se llamará cociente del polinomio p por el monomio m al polinomio que se obtiene dividiendo todos los monomios que forman el polinomio p por el monomio m.*

Traducción en **Coq**:

```
Fixpoint div_m [p:pol] : monom -> pol :=[m:monom]Cases p of
      vpol  =>  vpol
    | (cpol m1 p1) => (cpol (div_mon m1 m) (div_m p1 m))
    end.
```

Hay que hacer notar que la formalización anterior sólo se utilizará cuando todos los términos del polinomio puedan dividirse por el término del monomio dado.

Ejemplo 6.3.1. *No sería válida la operación* $\frac{x^2y+yx}{xy^2}$

En primer lugar probamos que podemos reescribir (*igualdad del sistema*) el polinomio $\frac{p_1+_p p_2}{m}$ por $(\frac{p_1}{m} +_p \frac{p_2}{m})$.

Teorema 6.3.22.

$$\forall p_1, p_2 \in K[X];\ \forall m \in M_K;\ \frac{p_1 +_p p_2}{m} = \frac{p_1}{m} +_p \frac{p_2}{m}$$

Prueba: Aplicando inducción sobre p_1 y las tácticas de reescritura.

Traducción en **Coq**:

```
Lemma distr_div_suma: (m:monom)(p1,p2:pol)
  (div_m (suma_app p1 p2) m)=(suma_app (div_m p1 m) (div_m p2 m)).
□
```

Teorema 6.3.23.

$$\forall p \in K[x_1, \ldots, x_n],\ \forall m \in M_{K,n},\ \frac{p \cdot_M m}{m} =_p p$$

Prueba: Por recursión estructural sobre el predicado *full_term_pol*, la primera submeta generada la resuelve el sistema automáticamente y la segunda se prueba aplicando resultados vistos en la formalización de cuerpos.

Traducción en **Coq**:

```
Lemma div_mult_inv:(m:monom)~(z_monom m)->(full_mon n m)->
   (p:pol)(full_term_pol p)->(equpol (div_m (mult_m p m) m) p).
□
```

Ahora se enuncia y prueba un resultado técnico que se usará para probar el teorema (6.3.21).

Teorema 6.3.24.

$$\forall p \in K[X];\ \forall m \in M_K;\ (p \cdot_M m =_p vpol) \wedge (m \neq_M 0_M) \Rightarrow \frac{p \cdot_M m}{m} =_p \frac{vpol}{m}$$

Prueba: Por recursión sobre *equpol* obtenemos siete metas correspondientes a cada uno de los constructores de *equpol*. Las relativas a la relación de equivalencia son triviales; la cuarta y quinta se prueban por reescritura de términos utilizando el resultado *distr_div_suma*; las relativas a los casos de monomios con términos iguales y monomio cero se resuelven utilizando la definición formal de monomio y aplicando resultados vistos anteriormente sobre la igualdad extensional.

Traducción en **Coq**:

```
Lemma div_pm_comp:(m:monom)~(z_monom m)->(p:pol)(equpol (mult_m p m) vpol)->
   (equpol (div_m (mult_m p m) m) (div_m vpol m)).

Induction 2; Intros; Trivial.
Auto.
Apply equpol_tran with (div_m q m); Auto.
Simpl.
Rewrite distr_div_suma.
Rewrite distr_div_suma.
Simpl; Auto.
Rewrite distr_div_suma.
Rewrite distr_div_suma; Auto.
Generalize H.
Case m; Intros.
Unfold z_monom in H1.
Simpl.
Apply equpol_tran with (cpol ((plusK (divK k q) (divK k' q)),(div_term t t0))
                             (div_m p (q,t0))); Auto.
Simpl.
Apply equpol_elim.
Generalize H H1.
Case m0; Case m; Simpl; Intros.
Apply divK_0; Auto.
□
```

Estamos ya en condiciones de demostrar, fácilmente, el teorema (6.3.21), antes propuesto.

Prueba: Se demuestra aplicando los dos resultados anteriores y tácticas de reescritura sobre la formalización de *div_m*.

Traducción en **Coq**:

```
Lemma mult_nz_monom: (p:pol)(full_term_pol p)->(m:monom)(full_mon n m)->
   ~(z_monom m)->(equpol (mult_m p m) vpol)->(equpol p vpol).

Intros.
Apply equpol_tran with (div_m (mult_m p m) m); Auto.
Replace vpol with (div_m vpol m); Auto.
□
```

A partir de este resultado y siguiendo el esquema de demostración indicado en la nota de la página 104, se prueba el teorema (6.3.20).

Traducción en **Coq**:

```
Lemma contr_mult_nz_monom: (p:pol)(full_term_pol p)->(~(equpol p vpol))->
   (c:K)(~(eqK c OK))->(t:term)(full n t)->(~(equpol (mult_m p (c,t)) vpol)).

Red.
```

```
Intros.
Cut (equpol p vpol); Auto.
Apply mult_nz_monom with m:=(c,t); Auto.
□
```

6.3.2. Algunas simplificaciones polinómicas

Los lemas que siguen serán de mucha utilidad, para simplificar expresiones, cuando una vez formalizado el concepto de **ideal de polinomios**, comprobemos sus propiedades; asimismo serán de vital importancia para probar propiedades sobre la **reducción de polinomios**.

Los resultados que damos a continuación son propiedades de las operaciones definidas en polinomios utilizando la igualdad extensional. Todos se demuestran utilizando el constructor *equpol_tran* y resultados sobre *equpol* ya probados. Debido a que no aportan novedades respecto a lo ya comentado y para que su lectura no se haga muy tediosa, no se detallarán las pruebas, salvo la primera.

Teorema 6.3.25.

$$\forall m \in M_K;\ \forall p,q,r \in K[X];\ p =_p q +_p -(r \cdot_M m) \Rightarrow q =_p p +_p r \cdot_M m$$

Traducción en **Coq**:

```
Lemma ch_m_plus: (p,q,r:pol)(m:monom)(equpol p
  (suma_app q (pol_opp (mult_m r m))))->(equpol q (suma_app p (mult_m r m))).

Intros.
Apply equpol_sym.
Apply equpol_tran with (suma_app (suma_app q
        (pol_opp (mult_m r m))) (mult_m r m)); Auto.
Apply equpol_tran with (suma_app q vpol); Auto.
Apply equpol_tran with (suma_app q
        (suma_app (pol_opp (mult_m r m)) (mult_m r m))); Auto.
□
```

Teorema 6.3.26.

$$\forall m \in M_K;\ \forall p,q,r \in K[X];\ p =_p q +_p -(r \cdot_M m) \Rightarrow q +_p (-p) =_p r \cdot_M m$$

Traducción en **Coq**:

```
Lemma ch_m_plus2: (p,q,r:pol)(m:monom)(equpol p
(suma_app q (pol_opp (mult_m r m))))->
   (equpol (suma_app q (pol_opp p)) (mult_m r m)).
□
```

Teorema 6.3.27.

$$\forall p,q,r \in K[X];\ [p +_p (-q)] +_p [q +_p (-r)] =_p p +_p (-r)$$

Traducción en **Coq**:

```
Lemma equat1: (p,q,r:pol)(equpol (suma_app (suma_app p (pol_opp q))
      (suma_app q (pol_opp r))) (suma_app p (pol_opp r))).
□
```

Teorema 6.3.28.

$$\forall p, q, p', q' \in K[X];\ p =_p (q \cdot_p q') +_p p' \Rightarrow -p =_p (q \cdot_p (-q')) +_p (-p')$$

Traducción en **Coq**:

```
Lemma mult_sum_opp2: (p,p',q,q':pol)(equpol p (suma_app (mult_p q q') p'))->
   (equpol (pol_opp p) (suma_app (mult_p q (pol_opp q')) (pol_opp p'))).
□
```

Teorema 6.3.29.

$$\forall p, q \in K[X];\ q +_p (-p) =_p -(p +_p (-q))$$

Traducción en **Coq**:

```
Lemma mult_sum_opp1: (f,g:pol)(equpol (suma_app g (pol_opp f))
   (pol_opp (suma_app f (pol_opp g)))).
□
```

Teorema 6.3.30.

$$\forall y, y' \in K;\ \forall y_0 \in T;\ \forall p', q', r \in K[X];$$
$$(r =_p p' +_p q') \Rightarrow r =_p ((y, y_0) : p') +_p ((-y, y_0) : q')$$

Traducción en **Coq**:

```
Lemma ayuda_Benj: (y,y':K)(y0:term)(p',q',r:pol)
   (eqK y (opK y'))->(equpol r (suma_app p' q'))->
      (equpol r (suma_app (cpol (y,y0) p') (cpol (y',y0) q'))).
□
```

Teorema 6.3.31.

$$\forall y, y' \in K;\ \forall y_0 \in T;\ \forall p', q', r \in K[X];$$
$$(r =_p p' +_p q') \Rightarrow ((y, y_0) : p') +_p ((y', y_0) : q') =_p ((y, y_0) : ((y', y_0) : r))$$

Traducción en **Coq**:

```
Lemma ayud_Gilb: (y,y':K)(y0:term)(p',q',r:pol)(equpol r (suma_app p' q'))->
      (equpol (suma_app (cpol (y,y0) p') (cpol (y',y0) q'))
              (cpol (y,y0) (cpol (y',y0) r))).
□
```

6.4. Coeficientes de los términos en los polinomios

Vamos a definir la función que extrae y devuelve el coeficiente de un término dado en un polinomio. Como estamos trabajando con polinomios no ordenados debemos recorrer todos los monomios de que consta el polinomio para buscar

si el término estudiado se repite, en cuyo caso sumamos los coeficientes correspondientes.

La formalización en **Coq** se hace por casos, desestructurando el tipo *pol*. Al polinomio *vpol* le asignamos el coeficiente 0; igualmente si el término no está en el polinomio se le asigna el coeficiente 0.

Traducción en **Coq**:

```
Fixpoint coef [p:pol]: term -> K:=[t:term] Cases p of
      vpol        => OK
   | (cpol (c,u) p1) => Cases (eq_tm_dec t u) of
        (left x) => (plusK c (coef p1 t))
      | (right y) => (coef p1 t)
      end
   end.
```

Verificamos ahora diversas propiedades de los coeficientes con respecto a las operaciones sobre polinomios. Denotamos por coef (t, p), el coeficiente del término t en el polinomio p.

Teorema 6.4.1.

$$\forall p, q \in K[X];\ \forall t \in T;\ \text{coef}\ (t, (p +_p q)) =_K \text{coef}\ (t, p) +_K \text{coef}\ (t, q)$$

Prueba: Haciendo recurrencia sobre el primer argumento p obtenemos dos submetas; la primera una vez hecha β-reducción es consecuencia de las propiedades de las operaciones sobre un cuerpo; para la segunda debemos en primer lugar explicitar el momomio m, utilizar el lema de decidibilidad sobre la igualdad de términos *eq_tm_dec* y aplicar propiedades de la estructura de cuerpo.

Nota. Prácticamente en todas las pruebas de esta sección se utilizará la táctica **Simpl** (β-reducción) y la explicitación de los monomios que formen los polinomios correspondientes.

Traducción en **Coq**:

```
Lemma coef_suma_pol: (p,q:pol)(t:term)(eqK
        (coef (suma_app p q) t) (plusK (coef p t) (coef q t))).

Induction p.
Intros.
Simpl.
Apply eqK_sym.
Apply eqK_trans with (plusK (coef q t) OK); Auto.
Induction m; Intros.
Simpl.
Elim (eq_tm_dec t y0); Intros; Auto.
Apply eqK_trans with (plusK y (plusK (coef p0 t) (coef q t)));Auto.
□
```

Los dos próximos resultados prueban que la función *coef* conserva tanto la igualdad de términos como la de polinomios. Es decir, caracterizaremos los polinomios por los coeficientes de sus términos.

Teorema 6.4.2.

$$\forall p \in K[X];\ \forall t_1, t_2 \in T;\ t_1 = t_2 \Rightarrow \text{coef}\ (t_1, p) =_K \text{coef}\ (t_2, p)$$

Traducción en **Coq**:

```
Lemma term_equpol2: (p:pol)(t1,t2:term)(t1=t2)->(eqK (coef p t1) (coef p t2)).
□
```

Teorema 6.4.3.

$$\forall p, p' \in K[X];\ \forall t \in T;\ p =_p p' \Rightarrow \text{coef}\ (t,p) =_K \text{coef}\ (t,p')$$

Prueba: Esta demostración se hace por recurrencia estructural sobre *equpol*, para lo cual se tienen que probar las siete metas correspondientes a los constructores del predicado *equpol*. Las tres primeras se obtienen casi automáticamente; las demás se hacen por casos utilizando la decidibilidad de la igualdad de términos y las propiedades de cuerpo.

Traducción en **Coq**:

```
Lemma term_equpol: (p,p':pol)(t:term)(equpol p p')->
                     (eqK (coef p t) (coef p' t)).

Intros p p' t H; Elim H.
Auto.
Auto.
Intros.
Apply eqK_trans with (coef q t); Auto.
Induction m; Intros.
Simpl.
Elim (eq_tm_dec t y0); Intros.
Apply eqK_trans with (plusK (coef p0 t) (coef (cpol (y,y0) q) t));Auto.
Rewrite y1.
Simpl.
Elim (eq_tm_dec y0 y0); Intros.
Apply eqK_trans with (plusK (plusK (coef p0 y0) y) (coef q y0));Auto.
Apply eqK_trans with (plusK (plusK y (coef p0 y0)) (coef q y0));Auto.
Apply eqK_trans with (plusK y (plusK (coef p0 y0) (coef q y0)));Auto.
Elim y2; Auto.
Apply eqK_trans with (plusK (coef p0 t) (coef q t));Auto.
Apply eqK_trans with (plusK (coef p0 t) (coef (cpol (y,y0) q) t)).
Apply plusK_comp_l.
Simpl.
Elim (eq_tm_dec t y0); Intros.
Elim y1; Auto.
Auto.
Auto.
Intros.
Apply eqK_trans with (plusK (coef p0 t) (coef q t));Auto.
Apply eqK_trans with (plusK (coef p'0 t) (coef q' t));Auto.
Intros.
Simpl.
Elim (eq_tm_dec t t0); Intros; Auto.
Induction m.
Unfold z_monom; Intros.
Simpl.
Elim (eq_tm_dec t y0); Intros; Trivial.
Apply eqK_trans with (plusK OK (coef p0 t)); Auto.
EAuto.
□
```

Teorema 6.4.4.

$$\forall p \in K[X];\ \forall t \in T;\ \text{coef}\ (t,p) =_K -\text{coef}\ (t,(-p))$$

Prueba: Primero se procede por inducción sobre el polinomio p, la primera submeta es trivial y la segunda se resuelve por casos apelando al lema (4.3.10).

Traducción en **Coq**:

```
Lemma coef_pol_opp: (p:pol)(t:term)(eqK (coef p t) (opK (coef (pol_opp p) t))).
```
□

El siguiente resultado es un corolario inmediato del teorema anterior.

Corolario 6.4.1.

$$\forall p \in K[X];\ \forall t \in T;\ -\text{coef}\ (t,p) =_K \text{coef}\ (t,(-p))$$

Traducción en **Coq**:

```
Lemma coef_pol_opp2:(p:pol)(t:term)(eqK (opK (coef p t)) (coef (pol_opp p) t)).
```
□

Ahora formulamos un resultado simple pero muy útil para futuras simplificaciones; la prueba se deduce de la formalización de la noción de coeficiente.

Teorema 6.4.5.

$$\forall p \in K[X];\ \forall t, t_0 \in T;\ \forall c \in K;$$
$$t \neq t_0 \Rightarrow \text{coef}\ (t,[(c,t_0) : p]) =_K \text{coef}\ (t,p)$$

Traducción en **Coq**:

```
Lemma coef_n_term: (c:K)(t,t0:term)(p:pol)(~t=t0)->
      (eqK (coef (cpol (c,t0) p) t) (coef p t)).
```
□

Teorema 6.4.6.

$$\forall p \in K[x_1,\ldots,x_n];\ \forall t, t_0 \in T^n;\ \forall c \in K;$$
$$\text{coef}\ ((t_0.t),[p\ ._M\ (c,t)]) =_K \text{coef}\ (t_0,p)\ ._K\ c$$

Prueba: Se aplica recurrencia sobre el primer argumento p; la submeta correspondiente a *vpol* la resuelve el sistema automáticamente, aplicando los **Hints** introducidos. En cuanto a la segunda, *(cpol m p0)*, en primer lugar explicitamos el monomio m y obtenemos información de la hipótesis que dice que el polinomio (*cpol* $(y,y0)$ $p0$) tiene n variables, mediante **Inversion**, que introducimos en el contexto. Utilizando la decidibilidad de los elementos del cuerpo respecto al elemento nulo, por medio de *Elim (decK c)*, estudiamos los casos ($c =_K 0$) y ($c \neq_K 0$); el primer caso se resuelve introduciendo en el contexto nuevas hipótesis, mediante la táctica **Cut**, y utilizando el hecho de que la igualdad de polinomios conserva los coeficientes (*term_equpol*). En cuanto al caso ($c \neq_K 0$) se prueba de manera análoga, salvo que tenemos que aplicar la decidibilidad de la igualdad de términos *Elim* (*eq_tm_dec y0 t0*) y probar las dos submetas obtenidas. Para ello utilizamos la definición de coeficiente, lo cual

implica simplificar su formalización y estudiarla caso a caso. Nos ayudamos de la hipótesis de recurrencia, lemas relativos a operaciones sobre términos y de la reescritura de términos. La prueba es bastante larga debido a los numerosos casos que debemos tratar.

Traducción en **Coq**:

```
Lemma coef_term_mult:(p:pol)(t,t0:term)(c:K)(full n t0)->(full_term_pol p)->
   (eqK (coef (mult_m p (c,t)) (term_mult t0 t)) (multK (coef p t0) c)).
```

□

Veremos que en el contexto de las reducciones de polinomios (división de Buchberger), a menudo nos interesa discernir si un término figura o no en un polinomio dado. La definición de coeficiente y sus propiedades, demostradas al comienzo de esta sección, nos permiten caracterizar esta pertenencia.

Definición 6.4.1. *Un término $t \in T$ figura en un polinomio $p \in K[X]$ si, y solo si, su coeficiente en ese polinomio no es igual a cero; en otro caso, decimos que dicho término no figura en ese polinomio.*

$$\forall t \in T;\ \forall p \in K[X];\ t \in p :\overset{def}{\Longleftrightarrow} \text{coef}\ (t,p) \neq_K 0$$

Traducción en **Coq**:

```
Definition term_in_pol: pol->term->Prop:=[p:pol][t:term](~(eqK (coef p t) OK)).

Definition no_term_in_pol: pol->term->Prop:=[p:pol][t:term](eqK (coef p t) OK).
```

El siguiente lema es un resultado técnico obtenido directamente de la definición anterior.

Lema 6.4.1.

$$\forall t \in T^n,\ \forall k \in K,\ (k \neq_K 0) \Rightarrow t \in ((k,t) : vpol)$$

Traducción en **Coq**:

```
Lemma term_in_pol_monom: (k:K)(t:term)(~(eqK k OK))->
        (term_in_pol (cpol (k,t) vpol) t).
```

□

Teorema 6.4.7. *La presencia de un término en un polinomio es decidible.*

$$\forall t \in T;\ \forall p \in K[X];\ (t \in p) \vee (t \notin p)$$

Prueba: Este lema, que probamos por recurrencia sobre la estructura del polinomio p, es un lema constructivo, es decir, del cual se puede extraer un programa de la prueba realizada. Como la presencia de un término en un polinomio se formalizó en función de la igualdad sobre el cuerpo, necesitamos utilizar los lemas de decidibilidad sobre K (*decK* y *decK_t*), así como la decidibilidad de la igualdad de términos *eq_tm_dec* para estudiar los casos generados por la formalización

de *coef*. Se razona por casos de manera similar a lo hecho en resultados anteriores. Se demuestra primero el lema coef_*z_in_pol*, aparentemente más débil, que simplifica la prueba del resultado.

Hay que resaltar la utilización de la táctica **Absurd**[2] necesaria para probar la meta $\neg(eqK \quad (plusK\ a\ (coef\ p\ t)) \quad OK)$ a partir de la hipótesis del contexto local $b1 : \neg(eqK \quad (coef\ p\ t) \quad (multK\_neg\ a))$.

Traducción en **Coq**:

```
Lemma coef_z_in_pol: (y:K)(eqK y 0K)->
 (t:term)(p:pol)({(term_in_pol p t)}+{(no_term_in_pol p t)})->
  (y0:term)({(term_in_pol (cpol (y,y0) p) t)}+
              {(no_term_in_pol (cpol (y,y0) p) t)}).


Lemma dec_term_in_pol: (p:pol)(t:term)
   {(term_in_pol p t)}+{(no_term_in_pol p t)}.

Induction p.
Right.
Red; Auto.
Induction m; Intros.
Elim (decK y); Intros.
Apply coef_z_in_pol; Auto.
Elim (eq_tm_dec t y0); Intros.
Elim (decK_t (coef p t) (opK y)); Intros.
Right.
Red.
Simpl.
Elim (eq_tm_dec t y0); Intros.
Apply eqK_trans with (plusK y (opK y)); Auto.
Elim y4; Auto.
Left.
Red.
Simpl.
Elim (eq_tm_dec t y0); Intros.
Red; Intros.
Absurd (eqK (coef p t) (opK y)); Trivial.
Apply suma_op_inv.
Apply eqK_trans with (plusK y (coef p t)); Auto.
Red; Intros.
Elim y3.
Apply eqK_trans with (opK (opK (coef p t))); Auto.
Elim y4; Auto.
Elim H with t:=t; Intros.
Left.
Red.
Red in y3.
Simpl.
Elim (eq_tm_dec t y0); Intros; Auto.
Right.
Red.
Red in y3.
```

[2]Esta táctica se aplica sobre cualquier meta. El uso más frecuente, se produce cuando dada una hipótesis H y suponiendo una propiedad P, se puede deducir del resto del contexto $\neg P$, independientemente de la meta en curso.

```
Simpl.
Elim (eq_tm_dec t y0); Intros.
Elim y2; Auto.
Auto.
```
□

El programa extraído, en **OCaml** y sin optimización "a mano" es:

```
let rec coef p t =
  match p with
    | Vpol -> oK
    | Cpol (m, p1) ->
        let Pair (c, u) = m in
        (match eq_tm_dec t u with
           | true -> plusK c (coef p1 t)
           | false -> coef p1 t)


let rec dec_term_in_pol = function
  | Vpol -> (fun t -> false)
  | Cpol (m, p0) ->
      let h = dec_term_in_pol p0 in
      (fun t ->
      let Pair (x, x0) = m in
      (match decK x with
         | true -> h t
         | false ->
             (match eq_tm_dec t x0 with
                | true ->
                    (match decK_t (coef p0 t) (multK_neg x) with
                       | true -> false
                       | false -> true)
                | false -> h t)))
```

Los dos resultados siguientes son corolarios del teorema 6.4.3.

Corolario 6.4.2.

$$\forall t \in T;\ \forall p, q \in K[X];\ (p =_p q)\ \wedge\ (t \notin p) \Rightarrow (t \notin q)$$

Prueba: Inmediato explicitando la definición *no_term_in_pol* y aplicando directamente el teorema (6.4.3).

Traducción en **Coq**:

```
Lemma comp_no_term_in_pol: (p,q:pol)(t:term)(equpol p q)->
        (no_term_in_pol p t)->(no_term_in_pol q t).
```
□

Fácilmente se prueba el contrarrecíproco del resultado anterior.

Corolario 6.4.3.

$$\forall t \in T;\ \forall p, q \in K[X];\ (p =_p q)\ \wedge\ (t \in p) \Rightarrow (t \in q)$$

Prueba: Inmediato a partir de la definición de *term_in_pol*.

Traducción en **Coq**:

```
Lemma comp_term_in_pol: (p,q:pol)(t:term)(equpol p q)->(term_in_pol p t)->
                        (term_in_pol q t).
```

□

Del teorema 6.4.4 y de las propiedades de los elementos de un cuerpo, se deducen fácilmente los dos resultados siguientes.

Corolario 6.4.4.

$$\forall t \in T;\ \forall p \in K[X];\ [t \in (-p)] \Rightarrow (t \in p)$$

Traducción en **Coq**:

```
Lemma term_in_pol_opp: (p:pol)(t:term)(term_in_pol (pol_opp p) t)->
                       (term_in_pol p t).
```

□

Corolario 6.4.5. $\forall t \in T;\ \forall p \in K[X];\ [t \notin (-p)] \Rightarrow (t \notin p)$

Traducción en **Coq**:

```
Lemma no_term_in_pol_opp: (p:pol)(t:term)(no_term_in_pol (pol_opp p) t)->
                          (no_term_in_pol p t).
```

□

A partir de un término de un polinomio, el predicado *term_in_pol* permite caracterizar los polinomios no nulos.

Teorema 6.4.8. $\forall t \in T;\ \forall p \in K[X];\ (t \in p) \Rightarrow (p \neq_p vpol)$

Prueba: La primera etapa consiste en desdoblar la negación e introducir en el contexto la hipótesis $(p =_p vpol)$ para que como meta quede *False*. Mediante la táctica **Cut**, introducimos en el contexto $(term\_in\_pol\ vpol\ t)$; con esta nueva hipótesis obtenemos fácilmente la meta, debido a que el coeficiente de cualquier término en el polinomio *vpol* es 0. La prueba de $(term\_in\_pol\ vpol\ t)$ se obtiene a partir de las hipótesis $(t \in p)$ y $(p =_p vpol)$ mediante el corolario 6.4.3.

Traducción en **Coq**:

```
Lemma term_in_pol_nvpol: (p:pol)(t:term)(term_in_pol p t)->~(equpol p vpol).
```

□

A partir de este teorema y de las propiedades de los elementos del cuerpo se obtienen fácilmente los dos resultados siguientes.

Corolario 6.4.6.

$$\forall p \in K[x_1, \ldots, x_n];\ \forall t \in T^n;\ \forall k \in K;\ (t \notin p) \wedge (k \neq 0) \Rightarrow [(k,t) +_p p] \neq_p vpol$$

Traducción en **Coq**:

```
Lemma no_equpol_Ttm: (p:pol)(full_term_pol p)->(k:K)(~(eqK k OK))->
  (t:term)(full n t)->(no_term_in_pol p t)->
    (~(equpol (suma_app (cpol (k,t) vpol) p) vpol)).
```

□

Ahora se prueba que "*un polinomio formado por un solo monomio no nulo es distinto de vpol*".

Corolario 6.4.7.

$$\forall m \in M_{K,n};\ \neg(z\_monom\ m) \Rightarrow (m : vpol) \neq_p vpol$$

Traducción en **Coq**:

```
Lemma n_zmonom_n_vpol: (m:monom)(full_mon n m)->(~(z_monom m))->
        ~(equpol (cpol m vpol) vpol).
```
□

Teorema 6.4.9. $\forall t, t' \in T^n,\ \forall k \in K,\ t \in [(k,t') : vpol] \ \Rightarrow\ t = t'$

Prueba: La demostración se hace por casos utilizando el lema de decidibilidad de la igualdad de términos. La primera submeta es una de las hipótesis del contexto y la segunda se obtiene por contradicción $((t = t') \wedge (t \neq t'))$ explicitando la hipótesis $H1 : (term\_in\_pol\ (cpol\ (k,t')\ vpol)\ t)$.

Traducción en **Coq**:

```
Lemma eq_term_in_pol_monom: (k:K)(t,t':term)(full n t)->(full n t')->
   (term_in_pol (cpol (k,t') vpol) t)->t=t'.
```
□

Los dos resultados siguientes se deducen directamente del teorema (6.4.5).

Corolario 6.4.8.

$$\forall t, t_0 \in T;\ \forall y \in K;\ \forall p \in K[X];\ [t \notin ((y, t_0) : p)] \ \wedge\ (t \neq t_0) \Rightarrow t \notin p$$

Traducción en **Coq**:

```
Lemma no_term_in_cpol: (y:K)(y0,t:term)(p:pol)
      (no_term_in_pol (cpol (y,y0) p) t)->~(t=y0)->(no_term_in_pol p t).
```
□

Corolario 6.4.9.

$$\forall t, t_0 \in T;\ \forall y \in K;\ \forall p \in K[X];\ t \in [(y, t_0) : p] \Rightarrow (t = t_0) \vee (t \in p)$$

Traducción en **Coq**:

```
Lemma split_term_in_pol_cons: (y:K)(y0,t:term)(p:pol)
     (term_in_pol (cpol (y,y0) p) t)->(t=y0)\/(term_in_pol p t).
```
□

A continuación se da un resultado valioso por su utilidad como lema auxiliar en pruebas posteriores, respecto a la presencia de un término en la suma de dos polinomios. La prueba, una vez desdoblada la formalización de *term_in_pol*, se basa en aplicar propiedades relativas a las operaciones sobre un cuerpo, ya probadas en el capítulo 4.

Teorema 6.4.10.

$$\forall t \in T;\ \forall p, q \in K[X];\ t \in (p +_p q) \Rightarrow (t \in p) \vee (t \in q)$$

Traducción en **Coq**:

```
Lemma term_in_sum: (p,q:pol)(t:term)(term_in_pol (suma_app p q) t)->
                     (term_in_pol p t)\/(term_in_pol q t).
```
□

Los dos lemas siguientes formalizan propiedades respecto a la "no presencia" de un término en un polinomio.

Teorema 6.4.11.

$$\forall t \in T^n;\ \forall p \in K[X];\ [\forall t' \in T;\ (t' \in p) \Rightarrow (t <_L t')] \Rightarrow (t \notin p)$$

Prueba: Aplicando el lema de decidibilidad *dec_term_in_pol* se generan dos metas. La primera es una de las hipótesis; la segunda se resuelve por contradicción debido a que de las hipótesis $H : (\forall t' \in T;\ (t' \in p) \Rightarrow (t <_L t'))$ y $a : (t \in p)$ se obtiene que $(t <_L t)$, lo que contradice el hecho de que $<_L$ sea un orden estricto.

Traducción en **Coq**:

```
Lemma Ttm_no_term_in_pol: (t:term)(p:pol)(full n t)->
 ((t':term)(term_in_pol p t')->(Ttm t t'))->(no_term_in_pol p t).
```
□

Al igual que en el resultado anterior, si utilizamos la decidibilidad de la presencia de un término en un polinomio y que el orden lexicográfico sobre términos verifica la propiedad irreflexiva, se puede probar que, "*un término no figura en un polinomio si es mayor que todos los términos del polinomio*".

Traducción en **Coq**:

```
Lemma Ttm_no_term_in_pol2: (t:term)(f:pol)(full n t)->
        ((t':term)(term_in_pol f t')->(Ttm t' t))->(no_term_in_pol f t).
```
□

Comprobamos, como era de esperar, que "*si un término figura en un polinomio de n variables, el término tiene n variables*".

Lema 6.4.2.

$$\forall p \in K[x_1, \ldots, x_n];\ (t \in p) \Rightarrow t \in T^n$$

Prueba: Se prueba por recurrencia sobre el predicado *full_term_pol*; la primera submeta es consecuencia de que el coficiente de cualquier término en *vpol* es cero y la segunda se prueba utilizando el lema coef_*n_term*.

Traducción en **Coq**:

```
Lemma full_coef: (f:pol)(t0:term)(full_term_pol f)->
                   (term_in_pol f t0)->(full n t0).
```
□

Otras propiedades aritméticas importantes que verifican los coeficientes de los monomios que componen un polinomio, se enuncian y prueban en los tres lemas siguientes. Resaltar que en el tercer resultado, se tiene como hipótesis adicional el hecho de que t_0 es un término del polinomio suma de p_1, (k, t) y p_2. Para probar estos lemas no se necesita la inducción, las demostraciones se basan en utilizar resultados sobre las operaciones del cuerpo, lemas relativos a la presencia de términos en polinomios, así como la propiedad de que el orden lexicográfico sobre los términos es estricto. Como siempre se incluyen sólo algunas pruebas representativas.

Lema 6.4.3.

$$\forall t \in T;\ \forall f, g \in K[X];\ (t \in f)\ \wedge\ (t \notin g) \Rightarrow t \in (f +_p g)$$

Traducción en **Coq**:

```
Lemma dec_sum_term_in_pol: (f,g:pol)(t:term)(term_in_pol f t)->
           (no_term_in_pol g t)->(term_in_pol (suma_app f g) t).
```

□

Lema 6.4.4.

$$\forall p, p_1, p_2 \in K[x_1, \ldots, x_n];\ \forall t, t_0 \in T^n;$$
$$(t_0 <_L t)\ \wedge\ (\forall t' \in p_1 \Rightarrow t <_L t')\ \wedge\ [p =_p p_1 +_p ((\text{coef}\ t\ p), t) +_p p_2]$$
$$\Rightarrow (\text{coef}\ t_0\ p_2) =_K (\text{coef}\ t_0\ p)$$

Traducción en **Coq**:

```
Lemma coef_Ttm_split_pol: (p,p1,p2:pol)(t:term)(full n t)->
 (full_term_pol p)->(full_term_pol p1)->(full_term_pol p2)->
  (equpol p (suma_app p1 (suma_app (cpol ((coef p t),t) vpol) p2)))->
   (t0:term)(full n t0)->(Ttm t0 t)->
 ((t':term)(term_in_pol p1 t')->(Ttm t t'))->(eqK (coef p2 t0) (coef p t0)).

Intros.
Cut (equpol p (suma_app (suma_app p1 (cpol ((coef p t),t) vpol)) p2)); Intros.
Apply eqK_trans
   with (coef (suma_app (suma_app p1 (cpol ((coef p t),t) vpol)) p2) t0).
Apply eqK_trans with
(plusK (coef (suma_app p1 (cpol ((coef p t),t) vpol)) t0) (coef p2 t0)); Auto.
Apply eqK_trans with (plusK OK (coef p2 t0)).
Apply eqK_trans with (plusK (coef p2 t0) OK); Auto.
Apply plusK_comp; Trivial.
Apply eqK_sym.
Apply eqK_trans with (plusK (coef p1 t0) (coef (cpol ((coef p t),t) vpol) t0)).
Auto.
Apply plusK_zero.
Change (no_term_in_pol p1 t0).
Apply Ttm_no_term_in_pol; Trivial; Intros.
Apply Ttm_trans with t n; Auto.
Apply full_coef with p1; Trivial.
Cut ~t=t0; Intros.
Apply eqK_trans with (coef vpol t0); Auto.
Red; Intros.
Cut t0=t; Auto.
Change ~t0=t.
Apply Ttm_strict with n; Trivial.
Auto.
Apply equpol_tran with
(suma_app p1 (suma_app (cpol ((coef p t),t) vpol) p2)); Auto.
```

□

Lema 6.4.5.

$$\forall p_1, p_2 \in K[x_1, \ldots, x_n];\ \forall t, t_0 \in T^n;\ \forall k \in K;$$
$$(t_0 <_L t)\ \wedge (\forall t' \in p_1 \Rightarrow t <_L t')\ \wedge\ (t_0 \in [p_1 +_p (k, t) +_p p_2])\ \Rightarrow\ t_0 \in p_2$$

Traducción en **Coq**:

```
Lemma split_term_in_pol: (p1,p2:pol)(full_term_pol p1)->(full_term_pol p2)->
 (k:K)(t,t0:term)(full n t)->(full n t0)->
  (term_in_pol (suma_app p1 (suma_app (cpol (k,t) vpol) p2)) t0)->
  (Ttm t0 t)->((t':term)(term_in_pol p1 t')->(Ttm t t'))->(term_in_pol p2 t0).
```

□

6.5. Descomposición de polinomios

En esta sección, introducimos un resultado fundamental a la hora de probar propiedades en la reducción (división) de polinomios. Dicho resultado dice que, dado un orden admisible sobre T^n (Ttm), podemos descomponer un polinomio como suma de tres polinomios que, en caso de existir, reunen respectivamente los términos mayores, iguales y menores que un término dado.

Teorema 6.5.1.

$\forall p \in K[x_1, \ldots, x_n];\ \forall t \in T^n;\ p =_p H(p,t) +_p [(\text{coef } (t,p)), t] +_p L(p,t)$ *siendo*

$$H(p,t) := \sum_{(u \in p)\ \wedge\ (t <_L u)} \text{coef } (u,p).u$$

$$L(p,t) := \sum_{(u \in p)\ \wedge\ (u <_L t)} \text{coef } (u,p).u$$

Prueba: Por recurrencia sobre el polinomio p.

- En el caso *vpol*, $H(f,t)$ y $L(f,t)$ son ambos *vpol* y $((\text{coef } (t,p)),t)$ es el monomio cero. Este caso se resuelve aplicando los constructores de *equpol* y utilizando el hecho de que el coeficiente asignado a cualquier término en el polinomio *vpol* es el 0.
- Para obtener la prueba en el caso (*cpol* m q), en primer lugar desdoblamos el monomio m en la forma (y, y_0) y obtenemos de la hipótesis de recurrencia, la decomposición del polinomio q en la forma $(H(q,t) +_p ((\text{coef } (t,q)),t) +_p L(q,t))$. Aplicando la decidibilidad de términos respecto a t e y_0 obtenemos tres submetas:
  - Si $(y_0 <_L t)$, $H(p,t)$ y $L(p,t)$ son $H(q,t)$ y $((y,y_0) : L(p,t))$, respectivamente y $[(\text{coef } (t,q)),t] =_M [(\text{coef } (t,p)),t]$. Las formalizaciones de las submetas intermedias se resuelven aplicando los resultados precedentes.
  - El caso $(t <_L y_0)$ es simétrico del anterior;
  - Cuando $t = y_0$, trivialmente $H(p,t)$ y $L(p,t)$ son $H(q,t)$ y $L(q,t)$ respectivamente y $((\text{coef } (t,p)),t)$ es el monomio $((\text{coef } (t,q)) +_K y, t)$.

Traducción en **Coq**:

```
Lemma gen_split_pol: (p:pol)(full_term_pol p)->(t:term)(full n t)->
 (EX p1 | (full_term_pol p1) /\ ((t':term)(term_in_pol p1 t')->(Ttm t t'))
  & (EX p2 | (full_term_pol p2) /\ ((t':term)(term_in_pol p2 t')->(Ttm t' t))
  & (equpol p (suma_app p1 (suma_app (cpol ((coef p t),t) vpol) p2))))).

Induction p.
Intros.
Split with vpol.
Split; Trivial.
Intros t' e; Elim e; Auto.
Split with vpol.
Split; Trivial.
Intros t' e; Elim e; Auto.
Simpl; Apply equpol_sym.
Apply equpol_elim; Auto.
Induction m; Intros.
Inversion H0.
Elim H with t; Trivial; Intros.
Elim H7; Intros.
Elim H8; Intros.
Elim H11; Intros.
Clear H11 H8 H7 H.
Elim (Ttm_total y0 t n); Trivial; Intros.
Elim y1; Intros; Clear y1.
Split with x.
Split; Trivial.
Split with (cpol (y,y0) x0).
Split; Auto; Intros.
Cut t'=y0\/(term_in_pol x0 t'); Intros.
Elim H7; Intros; Clear H7.
Rewrite H8; Trivial.
Apply H14; Trivial.
Apply split_term_in_pol_cons with y; Trivial.
Apply equpol_tran with (cpol (y,y0)
        (suma_app x (suma_app (cpol ((coef p t),t) vpol) x0))); Auto.
Apply equpol_tran with (suma_app x
        (cpol (y,y0) (suma_app (cpol ((coef p t),t) vpol) x0))).
Auto.
Apply equpol_ss; Trivial.
Apply equpol_tran with (cpol ((coef (cpol (y,y0) p) t),t)
        (cpol (y,y0) (suma_app vpol x0))); Auto.
Apply equpol_tran with
   (suma_app (cpol ((coef p t),t) vpol) (cpol (y,y0) x0)); Auto.
Apply equpol_tran with
   (cpol ((coef p t),t) (cpol (y,y0) (suma_app vpol x0))).
Auto.
Apply eq_mon_cpol.
Simpl.
Right.
Split; Trivial.
Elim (eq_tm_dec t y0); Auto; Intros.
Cut ~y0=t; Intros.
Elim H; Auto.
Apply Ttm_strict with n; Trivial.
Split with (cpol (y,y0) x).
Split; Auto; Intros.
Cut t'=y0\/(term_in_pol x t'); Intros.
```

```
Elim H7; Intros; Clear H7.
Rewrite H8; Trivial.
Apply H10; Trivial.
Apply split_term_in_pol_cons with y; Trivial.
Split with x0.
Split; Trivial.
Apply equpol_tran with (cpol (y,y0)
        (suma_app x (suma_app (cpol ((coef p t),t) vpol) x0))); Auto.
Simpl.
Apply equpol_cpol2; Trivial.
Apply equpol_ss; Trivial
Apply eq_mon_cpol.
Simpl.
Right.
Split; Trivial.
Elim (eq_tm_dec t y0); Auto; Intros.
Cut ~t=y0; Intros.
Elim H; Auto.
Apply Ttm_strict with n; Trivial.
Split with x.
Split; Auto.
Split with x0.
Split; Auto.
Apply equpol_tran with (cpol (y,y0)
        (suma_app x (suma_app (cpol ((coef p t),t) vpol) x0))); Auto.
Rewrite y1.
Apply equpol_tran with
        (suma_app x (cpol (y,t) (suma_app (cpol ((coef p t),t) vpol) x0))).
Auto.
Apply equpol_ss; Trivial.
Apply equpol_tran with (suma_app (cpol ((coef p t),t) vpol) (cpol (y,t) x0)).
Auto.
Simpl.
Elim (eq_tm_dec t t); Intros.
Apply equpol_tran with (cpol ((plusK (coef p t) y),t) x0); Auto.
Elim y2; Auto.
```

□

Como consecuencia del teorema anterior y del lema (6.4.2) se obtiene el siguiente resultado.

Corolario 6.5.1.

$$\forall p \in K[x_1, \ldots, x_n];\ \forall t \in p;\ p =_p H(p,t) +_p [(\text{coef}\ (t,p)), t] +_p L(p,t)$$

Traducción en **Coq**:

```
Lemma split_pol: (p:pol)(full_term_pol p)->(t:term)(term_in_pol p t)->
   (EX p1 | (full_term_pol p1)/\((t':term)(term_in_pol p1 t')->(Ttm t t'))
     & (EX p2 | (full_term_pol p2)/\((t':term)(term_in_pol p2 t')->(Ttm t' t))
      & (equpol p (suma_app p1 (suma_app (cpol ((coef p t),t) vpol) p2))))).
```

□

6.6. Polinomios en forma canónica

Hasta aquí hemos trabajado con polinomios (lista de monomios) no ordenados y algunos coeficientes posiblemente nulos; en esta sección vamos a definir,

mediante un predicado, los polinomios canónicos como "polinomios ordenados y con todos los coeficientes no nulos". Este predicado será fundamental, más adelante, para la formalización[3] del tipo polinomio canónico.

Para ordenar los monomios necesitamos un predicado, *low_pol*, que determina si un término es mayor que el del monomio de cabeza de un polinomio dado. Dado un término t y un polinomio p, denotamos este predicado como $(low\_pol\ t\ p)$[4]

```
Inductive low_pol : term -> pol -> Prop :=
  low_nil : (t:term)(low_pol t vpol)
| low_cons : (t:term)(u:term)(x:K)(p:pol)(Ttm u t)->(low_pol t (cpol (x,u) p)).
```

A continuación, se prueban cuatro propiedades relativas a este predicado que necesitaremos más adelante. Se demuestran fácilmente, utilizando **Inversion** y los constructores *low_cons* y *low_nil*. Serán muy útiles para simplificar pruebas posteriores.

Lema 6.6.1. $\forall p, p' \in K[X];\ \forall t, t' \in T;\ \forall c \in K,$ *se verifica que:*

$$[low\_pol\ t\ ((c,t') : p)] \Rightarrow [low\_pol\ t\ ((c,t') : p')]$$

Traducción en **Coq**:

```
Lemma low_pol_cpol: (t,t':term)(c:K)(p,p':pol)
   (low_pol t (cpol (c,t') p))->(low_pol t (cpol (c,t') p')).
□
```

Lema 6.6.2.

$$\forall p \in K[x_1, \ldots, x_n];\ \forall t, u \in T^n;\ (t <_L u)\ \wedge\ (low\_pol\ t\ p) \Rightarrow (low\_pol\ u\ p).$$

Traducción en **Coq**:

```
Lemma low_trans: (t,u:term)(full n t)->(full n u)->(p:pol)(full_term_pol p)->
           (low_pol t p)->(Ttm t u)->(low_pol u p).
□
```

Lema 6.6.3.

$$\forall p \in K[X];\ \forall t \in T;\ (low\_pol\ t\ p) \Rightarrow (low\_pol\ t\ (-p)).$$

Traducción en **Coq**:

```
Lemma low_opp: (t:term)(p:pol)(low_pol t p)->(low_pol t (pol_opp p)).
□
```

Lema 6.6.4.

$$\forall p, q \in K[X];\ \forall t \in T;\ [low\_pol\ t\ (p +_p q)] \Rightarrow (low\_pol\ t\ p).$$

Traducción en **Coq**:

[3] Denominada representación "sparse".

[4] Conservamos la misma notación de la formalización en Coq.

```
Lemma aux_suma_app_full: (t:term)(p,q:pol)(low_pol t (suma_app p q))->
                          (low_pol t p).
□
```

Dado que tenemos un orden total noetheriano ($<_L$) sobre términos, es posible representar los polinomios canónicos de forma inductiva, como listas ordenadas de monomios de n variables en forma estrictamente decreciente, para lo cual utilizamos el predicado anterior, suprimiendo los monomios con coeficiente nulo. Por ejemplo, el polinomio $(3x^3y - 5xy^2 + 7x^0y + 2x^0y^0)$ está ordenado con respecto al orden lexicográfico si $y <_L x$, y es igual (*equpol*) al polinomio $(-5xy^2 + 3x^3y + 0xy + 7y + 2)$ que no está en forma canónica.

```
Inductive full_pol : pol -> Prop :=
  full_pol_nil: (full_pol vpol)
| full_pol_cons : (t:term)(a:K)(P:pol) (~(eqK a OK))->(full n t)->
                    (full_pol P)->(low_pol t P)->(full_pol (cpol (a,t) P)).
```

Dado un polinomio p, para indicar que está en forma canónica utilizaremos la misma notación que su formalización en Coq; $(full\_pol\ p)$.

Destacar que la condición de listas de monomios estrictamente decrecientes implica la no existencia de términos iguales en un polinomio canónico.

Los tres lemas siguientes prueban resultados sobre las relaciones entre los predicados *full_pol*, *full_term_pol* y *full_mon*. Estos resultados se aplicarán constantemente para simplificar demostraciones posteriores. Las pruebas, al igual que los lemas previos, se realizan aplicando sus respectivos constructores y obteniendo información de las hipótesis, mediante la táctica **Inversion**.

Nota: Recordar que hemos fijado un n de tipo *nat* como variable al comienzo de la formalización de polinomios (página 85), y que el predicado *full_term_pol* identifica en Coq los polinomios de n variables.

Lema 6.6.5. *Todo polinomio canónico tiene n variables:*

$$\forall p \in K[X];\ (full\_pol\ p) \Rightarrow p \in K[x_1, \ldots, x_n].$$

Traducción en **Coq**:

```
Lemma fullp_impl_fullt:(p:pol)(full_pol p)->(full_term_pol p).

Induction p; Auto.
Induction m; Intros.
Inversion H0; Auto.
□
```

Lema 6.6.6. *Todo monomio no nulo de n variables es un polinomio canónico.*

$$\forall m \in M_{K,n};\ m \neq 0 \Rightarrow [full\_pol\ (m : vpol)].$$

Traducción en **Coq**:

```
Lemma full_monom: (m:monom)(~(z_monom m))->(full_mon n m)->
                    (full_pol (cpol m vpol)).
□
```

Lema 6.6.7.

$$\forall p \in K[X];\ \forall m \in M_K;\ [full\_pol\ (m : p)] \Rightarrow (full\_mon\ m).$$

Traducción en **Coq**:

```
Lemma full_pol_mon: (m:monom)(p:pol)(full_pol (cpol m p))->(full_mon n m).
```
□

Veamos que algunas de las operaciones formalizadas sobre polinomios conservan la relación *full_pol*. En primer lugar se prueba que "*el opuesto de un polinomio canónico también lo es*".

Teorema 6.6.1. $\forall p \in K[X];\ (full\_pol\ p) \Rightarrow [full\_pol\ (-p)]$.

Traducción en **Coq**:

```
Lemma full_opp: (p:pol)(full_pol p)->(full_pol (pol_opp p)).
```
□

Teorema 6.6.2.

$$\forall p, q \in K[X];\ [full\_pol\ (p +_p q)] \Rightarrow (full\_pol\ p).$$

Prueba: Por inducción sobre p; la primera submeta la resuelve directamente el constructor *full_pol_nil* y la segunda se obtiene a partir del lema 6.6.4 y de la hipótesis de inducción.

Traducción en **Coq**:

```
Lemma suma_app_full: (p,q:pol)(full_pol (suma_app p q))->(full_pol p).
```
□

Para probar que la relación *full_pol* se conserva por la operación *mult_m*, se necesita obtener primero el siguiente resultado sobre el predicado *low_pol*.

Teorema 6.6.3.

$$\forall p \in K[X];\ \forall c \in K;\ \forall t, t_1 \in T^n;$$
$$(full\_pol\ p) \wedge (low\_pol\ t_1\ p) \Rightarrow [low\_pol\ (t\ .\ t_1)\ (p\ ._M\ (c,t))]$$

Prueba: Por inducción sobre p; la primera submeta la resuelve, por medio de la táctica **Auto**, el constructor *low_nil*. La segunda, se resuelve mediante el constructor *low_cons* y utilizando que el orden lexicográfico sobre términos $<_L$ verifica la propiedad de monotonía por la izquierda, por ser un orden admisible, teorema (3.4.10).

Traducción en **Coq**:

```
Lemma low_pol_term_mult: (c:K)(t,t1:term)(p:pol)(full n t)->(full n t1)->
   (full_pol p)->(low_pol t1 p)->(low_pol (term_mult t t1) (mult_m p (c,t))).
```
□

Teorema 6.6.4.

$$\forall p \in K[X];\ \forall c \in K;\ \forall t \in T^n;\ (c \neq_K 0) \wedge (full\_pol\ p) \Rightarrow [full\_pol\ (p\ ._M\ (c,t))]$$

Prueba: Por recurrencia sobre la definición de polinomio, para demostrar que el polinomio resultante está ordenado se aplica el resultado anterior. Sólo queda probar que todos los monomios que componen el polinomio obtenido del producto $(p \cdot_M (c,t))$ son no nulos. Esto se obtiene de la hipótesis $(c \neq_K 0)$, junto con la propiedades del cuerpo K.

Traducción en **Coq**:

```
Lemma full_mult_mon: (c:K)(t:term)(p:pol)(full n t)->(~(eqK c 0K))->
                     (full_pol p)->(full_pol (mult_m p (c,t))).
```
□

Ahora probamos algunos lemas que relacionan los predicados definidos al comienzo de esta sección con los coeficientes de los monomios de un polinomio.

Teorema 6.6.5. *Si un término es mayor que todos los términos de un polinomio canónico, el polinomio no contiene a dicho término.*

$$\forall p \in K[X];\ \forall t \in T^n;\ (full\_pol\ p)\ \wedge\ (low\_pol\ t\ p) \Rightarrow (\text{coef}\ t\ p) =_K 0$$

Prueba: Por recurrencia estructural sobre el polinomio p, la primera meta es consecuencia de la formalización de coeficiente de un término en un polinomio; para resolver la segunda utilizamos el carácter transitivo del predicado *low_pol*. Las submetas generadas por este lema se obtienen de las hipótesis $(low\_pol\ t\ p)$ y $(full\_pol\ p)$, aplicando sucesivamente resultados anteriores.

Traducción en **Coq**:

```
Lemma coef_low_pol:(t:term)(p:pol)(low_pol t p)->(full n t)->(full_pol p)->
                   (eqK (coef p t) 0K).
```
□

Como consecuencia del teorema anterior y de la definición de polinomio canónico obtenemos los tres resultados siguientes.

Corolario 6.6.1.

$$\forall p \in K[X];\ (full\_pol\ p)\ \wedge\ (p =_p vpol) \Rightarrow p = vpol$$

Traducción en **Coq**:

```
Lemma full_eq_vpol: (p:pol)(full_pol p)->(equpol p vpol)->p=vpol.
```
□

Corolario 6.6.2.

$$\forall p \in K[X];\ \forall y_0 \in T;\ \forall y \in K;\ [full\_pol\ ((y, y_0) : p)] \Rightarrow y_0 \notin p$$

Traducción en **Coq**:

```
Lemma full_no_term: (y:K)(y0:term)(p:pol)(full_pol (cpol (y,y0) p))->
        (no_term_in_pol p y0).
```
□

Corolario 6.6.3.

$$\forall p \in K[X];\ \forall t \in T;\ \forall c \in K;\ [full\_pol\ ((c,t) : p)] \Rightarrow [\text{coef}\ t\ ((c,t) : p)] =_K c$$

Traducción en **Coq**:

```
Lemma coef_first : (p:pol)(c:K)(t:term)(full_pol (cpol (c,t) p))->
      (eqK (coef (cpol (c,t) p) t) c).
```
□

A partir del teorema 6.4.5 y la definición de polinomio canónico obtenemos el siguiente resultado.

Teorema 6.6.6.

$$\forall p \in K[X];\ \forall t, t' \in T;\ \forall a \in K;\ [full\_pol\ ((a,t) : p)] \wedge (t' \in p) \Rightarrow t' \in [(a,t) : p]$$

Traducción en **Coq**:

```
Lemma full_pol_cons_term: (a:K)(t,t':term)(p:pol)(full_pol (cpol (a,t) p))->
   (term_in_pol p t')->(term_in_pol (cpol (a,t) p) t').
```
□

Cuando se trabaje con polinomios canónicos, se simplificarán muchas demostraciones empleando los resultados que se prueban a continuación: *en un polinomio canónico no aparecen monomios repetidos* y *un polinomio canónico formado con el constructor cpol no puede ser igual a vpol.*

Teorema 6.6.7. $\forall p \in K[X];\ \forall m \in M_K;\ \neg[full\_pol\ (m : (m : p))]$

Prueba: Aplicando la irreflexividad de la relación orden sobre términos Ttm y la definición del predicado $full\_pol$.

Traducción en **Coq**:

```
Theorem  no_mon_rep_full: (m:monom)(p:pol)~(full_pol (cpol m (cpol m p))).
```
□

Teorema 6.6.8.

$$\forall p \in K[X];\ \forall m \in M_K;\ [full\_pol\ (m : p)] \Rightarrow (m : p) \neq_p vpol$$

Prueba: Se prueba que el coeficiente de m no es 0 y que el término de m no aparece en más monomios del polinomio p mediante **Inversion** de la hipótesis del contexto fp : $(full\_pol\ (cpol\ m\ p))$.

Traducción en **Coq**:

```
Lemma full_cp_neq_vp : (p:pol)(m:monom)(full_pol (cpol m p))->
      ~(equpol (cpol m p) vpol).
```
□

Mediante este teorema podemos probar fácilmente "la decidibilidad para polinomios canónicos respecto al polinomio *vpol*".

Traducción en **Coq**:

```
Lemma vpol_dec: (p:pol)(full_pol p)->{(equpol p vpol)}+{~(equpol p vpol)}.
```
□

Como una aplicación de los resultados anteriores comprobamos que: *si un término es mayor que el término de cabeza de un polinomio canónico, entonces es mayor que todos los términos de ese polinomio.*

Prueba: Se prueba por inducción sobre el polinomio p; la primera submeta se resuelve al tener una hipótesis falsa $H : (t \in vpol)$[5]. Cuando p es de la forma $(cpol\ (y, y0)\ p0)$ aplicamos la decidibilidad de la igualdad de términos a t e $y0$; si $t = y0$ es trivial pues la meta es una de las hipótesis, si $t \neq y0$ se demuestra utilizando la transitividad del orden lexicográfico de términos y el corolario (6.4.9).

Traducción en **Coq**:

```
Lemma term_in_Ttm_pol: (p:pol)(full_pol p)->(y1,t:term)(full n y1)->
        (low_pol y1 p)->(term_in_pol p t)->(Ttm t y1).
```

□

En próximos capítulos será esencial utilizar que todo polinomio admite, para un orden dado entre los términos, una única representación canónica, módulo *equpol*. Para ello demostramos primero, un lema más débil: como se puede insertar un monomio de n variables no nulo en un polinomio canónico, así como una "condición de ordenación del polinomio resultante".

Lema 6.6.8.

$$\begin{array}{l}\forall p \in K[X];\ \forall m \in M_{K,n};\ (m \neq 0)\ \wedge\ (full\_pol\ p) \Rightarrow \\ \{\exists\ q \in K[X];\ (full\_pol\ q)\ \wedge\ [q =_p (m : p)] \wedge \\ (\forall t \in T^n;\ (low\_pol\ t\ p)\ \wedge\ [(mon\_term\ m) <_L t] \Rightarrow (low\_pol\ t\ q))\}\end{array}$$

Prueba: Es un lema constructivo que se demuestra por recurrencia sobre p. El monomio no nulo m es de la forma (x, t) donde $x \in K$ y $t \in T^n$.

- En el caso de $p = vpol$, q es el polinomio $((x, t) : vpol)$. Las tres condiciones pedidas en la meta se prueban automáticamente por los resultados dados en esta sección.
- Si $(p = (y, u) : p0)$ en donde $y \in K$ y $u \in T^n$, utilizando el caracter total del orden $(<_L)$ se estudian los posibles casos para t y u:
  - Cuando $(t <_L u)$, q es el polinomio $((y, u) : p')$ siendo p' el polinomio obtenido por la hipótesis de recurrencia sobre $p0$.
  - Si $(u <_L t)$, es $((x, t) : ((y, u) : p0))$.
  - Por último cuando $t = u$ procedemos de nuevo por casos:
    - Si $(x +_K y =_K 0)$ entonces, trivialmente el polinomio buscado es $p0$.
    - Si $(x +_K y \neq_K 0)$ tenemos que el polinomio q es $((x +_K y, t) : p0)$.

En todos los casos anteriores para obtener la prueba de ordenación del enunciado $(\forall t \in T^n;\ (low\_pol\ t\ p)\ \wedge\ ((mon\_term\ m) <_L t) \Rightarrow (low\_pol\ t\ q))$ se utilizan los lemas relacionados con el predicado *low_pol*, probados al comienzo de esta sección.

Traducción en **Coq**:

[5] Se resuelve con *Elim H*; *Auto*. Independientemente de la meta.

```
Lemma insert : (m:monom)(full_mon n m)->(~(z_monom m))->(p:pol)(full_pol p)->
   {q:pol|(full_pol q)
    /\(equpol q (cpol m p))
    /\(t:term)(full n t)->(low_pol t p)->(Ttm (mon_term m) t)->(low_pol t q)}.

Induction m; Intros x t fm zm; Induction p.
Intros; Split with (cpol (x,t) vpol); Auto.
Induction m0; Intros y u p' HR fuy.
Elim (Ttm_total t u n).
Induction 1; Intros fp.
Elim HR.
Intros q; Destruct 1; Intros fq; Destruct 1; Intros eq lq.
Split with (cpol (y,u) q).
Split.
Inversion fuy; Auto.
Split.
Apply equpol_tran with (cpol (y,u) (cpol (x,t) p')).
Auto.
Exact (equpol_cm (y,u) (cpol (x,t) vpol) p').
Intros v fv lv tv; Inversion lv; Auto.
Inversion fuy; Auto.
Split with (cpol (x,t) (cpol (y,u) p')); Auto.
Intros e; Elim (decK (plusK x y)).
Intros ei; Split with p'.
Split.
Inversion fuy; Trivial.
Split.
Rewrite e.
Apply equpol_tran with (cpol ((plusK x y),u) p'); Auto.
Intros v fv l i; Simpl in i; Inversion fuy; Inversion l;
   Apply low_trans with u; Auto.
Rewrite <- e; Intros neq; Split with (cpol ((plusK x y),t) p');
   Inversion fuy; Split.
Apply full_pol_cons; Auto.
Rewrite e; Auto.
Split.
Auto.
Intros v fv lv tv; Inversion lv; Apply low_cons; Auto.
Inversion fm; Auto.
Inversion fuy; Auto.
□
```

Nota: Hemos utilizado la táctica **Exact**[6], porque la meta es lo suficientemente simple para ser dada directamente y, así se evitan pasos intermedios, normalmente muy engorrosos.

El programa extraído, en **OCaml** y sin optimización "a mano" es:

```
let insert m x =
  let Pair (x0, x1) = m in
  let rec f = function
    | Vpol -> Cpol ((Pair (x0, x1)), Vpol)
    | Cpol (m0, p0) ->
        let Pair (x2, x3) = m0 in
        (match ttm_total x1 x3 n with
           | Inleft x4 ->
```

[6]Esta táctica se puede aplicar a cualquier meta. Si T es la meta, y p un término de tipo U entonces *Exact* p tiene éxito si T y U son convertibles; en otro caso no hace nada.

```
            (match x4 with
               | true -> Cpol ((Pair (x2, x3)), (f p0))
               | false -> Cpol ((Pair (x0, x1)), (Cpol ((Pair (x2,
                  x3)), p0))))
         | Inright ->
            (match decK (plusK x0 x2) with
               | true -> p0
               | false -> Cpol ((Pair ((plusK x0 x2), x1)), p0)))
in f x
```

Teorema 6.6.9.

$$\forall p \in K[x_1, \ldots, x_n];\ \{\exists\, q \in K[x_1, \ldots, x_n];\ (full\_pol\ q)\ \wedge\ (p =_p q)\}$$

Prueba: Se hace recurrencia sobre el polinomio p; el primer caso es trivial pues *vpol* es un polinomio canónico. Para el caso $(m : p')$, de la hipótesis de recurrencia obtenemos un polinomio canónico q igual a p'. Utilizando sobre el monomio m el lema de decidibilidad de monomios con respecto al monomio cero, *z_monom_dec*, desdoblamos la meta en dos submetas; si m es el monomio cero el polinomio buscado es trivialmente p', en otro caso se utiliza el lema anterior para insertar dicho monomio en el polinomio q.

Traducción en **Coq**:

```
Lemma can_fun : (p:pol)(full_term_pol p)->{q:pol|(full_pol q)/\(equpol p q)}.

Induction p.
Intros; Split with vpol; Auto.
Intros m p' HR fp.
Cut (full_term_pol p'); Try (Inversion fp ; Auto).
Intros fp'; Elim (HR fp'); Intros q; Destruct 1; Intros fq eq.
Elim (z_monom_dec m).
Intros zm; Exists q; Split; Trivial.
Apply equpol_tran with p'; Auto.
Intros nzm; Elim insert with m q; Try (Assumption Orelse  Inversion fp; Auto).
Intros q0; Destruct 1; Intros fq0; Destruct 1; Intros eq0 lq0;Split with q0;
Split; [Assumption | Apply equpol_tran with (cpol m q); Auto].
□
```

Al igual que en pruebas constructivas anteriores, incluimos el programa extraído de la prueba anterior en **OCaml** y sin optimización "a mano":

```
let z_monom_dec = function
  | Pair (x, x0) -> decK x

let rec can_fun = function
  | Vpol -> Vpol
  | Cpol (m, p0) ->
      let x = can_fun p0 in
      (match z_monom_dec m with
         | true -> x
         | false -> insert m x)
```

Ahora podemos asociar a un nombre, *fun_can*, la construcción de la forma canónica de un polinomio en n variables, que está incluida en la prueba del resultado anterior, y asociando el valor *vpol* a los restantes elementos de *pol*. La forma canónica de polinomio p, obtenida mediante esta función, se denotará $(can\ p)$.

Se usa aquí **Definition** en lugar de **Lemma**, para dar la construcción de *fun_can*. Nótese que aquí no vale cualquier término que tenga de tipo el objetivo de la meta en cada momento. Es preciso elegir entre las posibles soluciones aquella que satisfaga las propiedades oportunas.

Traducción en **Coq**:

```
Definition fun_can : pol -> pol.

Intros p; Elim (full_pol_tm_dec p).
Intros cp; Elim (can_fun p cp); Intros q; Intros; Exact q.
Intros; Exact vpol.
Defined.
□
```

Nota: Coq define un atributo asociado a las definiciones: la "transparencia". Una definición $x := t : T$ es transparente si nos interesa tanto el término t como su tipo T; es "opaca" si solamente nos interesa el tipo T y la existencia de t, es decir que T tiene un elemento[7].

Otra razón para utilizar en este caso, **Definition** en lugar de **Lemma**, es que combinando las órdenes **Definition** y **Defined** en lugar de **Lemma** y **Save** o **Qed**, los términos de prueba son transparentes y, en el caso de la obtención de la forma canónica de un polinomio nos interesa sobre todo su construcción. Por otra parte si se utiliza **Lemma** y **Save** o **Qed**, el sistema también permite hacer la prueba transparente mediante la táctica **Transparent**.

A continuación se presenta el programa extraído, en **OCaml** y sin optimización "a mano", donde se ve claramente que se construye el canónico en el caso de polinomios de n variables y, en otro caso, se devuelve un valor arbitrario.

```
let fun_can p =
  match full_pol_tm_dec p with
    | true -> can_fun p
    | false -> Vpol
```

Las dos lemas siguientes comprueban que el polinomio obtenido al aplicar la función *fun_can* a un polinomio de n variables, es un polinomio en forma canónica igual al dado. Ambos se prueban directamente desdoblando la construcción[8] de la función *fun_can*.

```
Lemma can_corr : (p:pol)(full_term_pol p)->(equpol p (fun_can p)).

Intros p fp; Unfold fun_can; Elim (full_pol_tm_dec p); Simpl.
Clear fp; Intros fp; Elim (can_fun p fp); Simpl.
Tauto.
Intros XX; Elim (XX fp).


Lemma can_is_full : (p:pol)(full_term_pol p)->(full_pol (fun_can p)).

Intros p fp; Unfold fun_can; Elim (full_pol_tm_dec p); Simpl.
```

[7] En francés "est habité".

[8] El programa está escrito en el lenguaje de especificación subyacente en Coq que es **Gallina**.

```
Clear fp; Intros fp; Elim (can_fun p fp); Simpl.
Tauto.
Intros XX; Elim (XX fp).
```
□

La decidibilidad de los polinomios respecto a *vpol* es ahora inmediata para polinomios cualesquiera de n variables. La prueba se obtiene a partir de la forma canónica del polinomio dado y aplicando los teoremas (6.2.7 y 6.6.8).

```
Lemma vpol_dec_full_term: (p:pol)(full_term_pol p)->
                  {(equpol p vpol)}+{~(equpol p vpol)}.
```
□

El programa extraído, en **OCaml** y sin optimización "a mano" es:

```
let vpol_dec_full_term p =
  match can_fun p with
    | Vpol -> true
    | Cpol (m, p0) -> false
```

A partir del lema de decidibilidad anterior y utilizando propiedades de las operaciones sobre polinomios se obtiene la prueba del lema de decidibilidad de dos polinomios cualesquiera de n variables, resultado clave para muchas demostraciones y formalizaciones posteriores.

```
Lemma equpol_dec_full_term: (p,q:pol)(full_term_pol p)->(full_term_pol q)->
   {(equpol p q)}+{~(equpol p q)}.
```
□

6.6.1. Operaciones con polinomios en forma canónica

Ahora damos las operaciones suma y producto de polinomios canónicos, probando que el resultado de realizar esas operaciones entre polinomios cualesquiera de n variables es el mismo que se obtiene al aplicarlas a sus representantes canónicos, explicitando además que el polinomio resultante verifica la condición de ordenación, *low_pol*, respecto a un término dado t.

Teorema 6.6.10.

$$\begin{array}{l}\forall p, q \in K[X] \Rightarrow \{\, \exists\, r \in K[X];\\ (r =_p p +_p q) \wedge [(full\_pol\ p) \wedge (full\_pol\ q) \Rightarrow (full\_pol\ r)] \wedge\\ [\forall t \in T^n;\ (low\_pol\ t\ p) \wedge (low\_pol\ t\ q) \Rightarrow (low\_pol\ t\ r)]\ \}\end{array}$$

Prueba: Este resultado constructivo se demuestra por recurrencia estructural sobre los polinomios p y q. En el caso de $p = vpol$, r es el polinomio q y viceversa. Para resolverlo cuando $p = ((c,t) : p')$ y $q = ((c',t') : q')$ utilizamos la decidibilidad de términos respecto a t y t' y se prueba por casos:

- Cuando $(t <_L t')$, r es $((c',t') : x)$ siendo x el polinomio obtenido de la hipótesis de recurrencia.
- Si $(t' <_L t)$, r es $((c,t) : s)$, siendo s el polinomio obtenido de la hipótesis de recurrencia.

- Por último cuando $t = t'$ se estudian los casos $(c =_K -c')$ y $(c \neq_K -c')$.

En cada caso para obtener las pruebas de la submetas obtenidas, se utilizan los lemas relacionados con los predicados *low_pol* y *full_pol*, probados en el comienzo de la sección.

Traducción en **Coq**:

```
Theorem add_effic: (p,q:pol){r:pol|(full_pol p)->(full_pol q)->(full_pol r)
       /\(equpol r (suma_app p q))
       /\((t:term)(full n t)->(low_pol t p)->(low_pol t q)->(low_pol t r))}.
```

□

Al igual que hemos hecho con la construcción de la forma canónica de un polinomio, asociamos el nombre, *add_effic_fun*, a la operación suma de polinomios en forma canónica, que está incluida en la prueba del resultado anterior. A esta operación la llamaremos suma eficaz $(+_{ef})$. Hacemos transparente esta construcción por la mismas razones dadas para la función *fun_can*.

Traducción en **Coq**:

```
Definition add_effic_fun : pol -> pol -> pol.
Intros p q.
Elim (add_effic p q).
Intros pq; Intros; Exact pq.
Defined.
```

□

Los dos resultados siguientes prueban que la suma eficaz de polinomios canónicos es un polinomio canónico y que el resultado de sumar dos polinomios canónicos mediante la suma y la suma eficaz es el mismo. Sus pruebas se hacen explicitando la función suma eficaz y simplificando la expresión resultante.

Traducción en **Coq**:

```
Lemma add_effic_full : (p,q:pol)(full_pol p)->(full_pol q)->
                          (full_pol (add_effic_fun p q)).

Intros.
Unfold add_effic_fun;Elim (add_effic p q); Simpl.; Tauto.


Lemma add_effic_cor_l : (p,q:pol)(full_pol p)->(full_pol q)->
       (equpol (add_effic_fun p q)(suma_app p q)).

Intros.
Unfold add_effic_fun;Elim (add_effic p q); Simpl; Tauto.
```

□

Nota: En las pruebas de los lemas anteriores se utiliza la táctica *Tauto*. Esta táctica, debida a Cesar Muñoz [90] proporciona un procedimiento de decisión para las tautologías proposicionales inductivas. Este procedimiento permite probar fórmulas lógicas que no pueden ser resueltas por la táctica **Auto**. En particular, si el contexto posee, como en los lemas anteriores, conjunciones y disyunciones, **Auto** no sabe utilizarlas mientras que **Tauto** si las utiliza.

El resultado anterior, *add_effic_cor_l*, se puede extender a polinomios cualesquiera de n variables.

Teorema 6.6.11.

$$\forall p, q \in K[x_1, \ldots, x_n];\ p +_p q =_p (can\ p) +_{ef} (can\ q)$$

Prueba: Como hemos probado que $p =_p (can\ p)$ y $q =_p (can\ q)$, formalizamos en Coq la siguientes igualdades, utilizando los resultados precedentes y otros obtenidos sobre polinomios canónicos.

$$p +_p q =_p (can\ p) +_p (can\ q) =_p (can\ p) +_{ef} (can\ q)$$

Traducción en **Coq**:

```
Lemma comp_sum_effic: (p,q:pol)(full_term_pol p)->(full_term_pol q)->
   (equpol (add_effic_fun (fun_can p) (fun_can q)) (suma_app p q)).
```

□

Basándonos en la suma eficaz definida anteriormente, construimos, de manera totalmente natural, el producto de polinomios en forma canónica. Tanto los resultados como sus pruebas son similares a los de la suma, por ello sólo daremos algunas indicaciones.

En primer lugar construimos la multiplicación de un polinomio en forma canónica por un monomio de n variables.

Teorema 6.6.12.

$$\forall p \in K[X];\ \forall m \in M_K\{\exists\, r \in K[X];\ (full\_mon\ m)\ \wedge\ (full\_pol\ p) \Rightarrow [(full\_pol\ r)\ \wedge\ (r =_p p\ .{}_M\ m)]\}$$

Prueba: Se hace por inducción sobre el polinomio p, el caso *vpol* es trivial y el caso correspondiente al constructor *cpol* se resuelve utilizando el resultado de la suma de polinomios canónicos.

Traducción en **Coq**:

```
Lemma mult_m_full: (p:pol)(m:monom)
{r:pol | (full_mon n m)->(full_pol p)->(full_pol r)/\(equpol r (mult_m p m))}.
```

□

Al igual que hemos hecho en la suma de polinomios canónicos asociamos un nombre, *mult_m_effic_fun*, a la operación producto de un polinomio en forma canónica por un monomio de n variables. Siguiendo la notación anterior llamaremos a esta operación producto eficaz de un polinomio por un monomio.

```
Definition mult_m_effic_fun : pol -> monom -> pol.

Intros p m.
Elim (mult_m_full p m).
Intros pm; Intros; Exact pm.
Defined.
```

□

Comprobamos que producto eficaz de un polinomio por un monomio es otro polinomio canónico.

```
Lemma mult_m_effic_full : (p:pol)(m:monom)(full_pol p)->(full_mon n m)->
       (full_pol (mult_m_effic_fun p m)).

Intros.
Unfold mult_m_effic_fun; Elim (mult_m_full p m); Simpl.
Tauto.
□
```

Los resultados siguientes permiten comprobar que el computo del producto eficaz de un polinomio por un monomio es igual, módulo *equpol*, al del producto de un polinomio por un monomio.

```
Lemma mult_m_effic_cor_1 : (p:pol)(m:monom)(full_pol p)->(full_mon n m)->
       (equpol (mult_m_effic_fun p m)(mult_m p m)).
Intros.
Unfold mult_m_effic_fun; Elim (mult_m_full p m); Simpl.
Tauto.


Lemma comp_mult_m_effic: (p:pol)(m:monom)(full_term_pol p)->(full_mon n m)->
   (equpol (mult_m_effic_fun (fun_can p) m) (mult_m p m)).

Intros.
Cut (equpol p (fun_can p)); Intros.
Apply equpol_tran with (mult_m (fun_can p) m).
Apply mult_m_effic_cor_1; Trivial.
Apply can_is_full; Trivial.
Auto.
Apply can_corr; Trivial.
□
```

Utilizando el producto eficaz de un polinomio por un monomio y la suma eficaz de polinomios construimos el producto de polinomios en forma canónica.

Teorema 6.6.13.

$$\forall p, q \in K[x_1, \ldots, x_n];\ \{\exists\, r \in K[x_1, \ldots, x_n];\ (full\_pol\ p)\ \wedge\ (full\_pol\ q) \Rightarrow [(full\_pol\ r)\ \wedge\ (r =_p p\ ._p\ q)]\}$$

Traducción en **Coq**:

```
Lemma mult_p_can : (p,q:pol)
    {r:pol|(full_pol p)->(full_pol q)->(full_pol r)/\(equpol r (mult_p p q))}.
□
```

Asociamos el nombre, *mult_p_effic_fun*, a la operación producto de polinomios en forma canónica, que está incluida en la prueba del resultado anterior, obteniendo como en los casos anteriores lemas relativos a esta operación, que comprueban que es la función buscada. A esta operación la denominaremos producto eficaz $(._{ef})$.

```
Definition mult_p_effic_fun: pol -> pol -> pol.
Intros p q.
Elim (mult_p_can p q).
Intros pq; Intros; Exact pq.
Defined.
```

```
Lemma mult_p_effic_full: (p,q:pol)(full_pol p)->(full_pol q)->
       (full_pol (mult_p_effic_fun p q)).


Lemma mult_p_effic_cor_1: (p,q:pol)(full_pol p)->(full_pol q)->
       (equpol (mult_p_effic_fun p q)(mult_p p q)).


Lemma comp_mult_p_effic: (p,q:pol)(full_term_pol p)->(full_term_pol q)->
   (equpol (mult_p_effic_fun (fun_can p) (fun_can q)) (mult_p p q)).
□
```

6.7. Polinomios similares

En algunas formalizaciones[9] de la sección siguiente necesitaremos una relación, sobre polinomios cualesquiera de n variables, más fina que *equpol*. Esta relación que llamaremos *same_pol* dice que dos polinomios están relacionados (son **similares**) si sus listas de términos correspondientes son "sintácticamente iguales" y los coeficientes respectivos pertenecen a la misma clase en la relación *eqK*; es decir son polinomios con el mismo número de variables, con los mismos términos y en el mismo orden. La notación que se utilizará para polinomios similares es $p \approx q$.

Ejemplo 6.7.1. *Se verifica que* $3x^2y + 2xy =_p 2xy + 3x^2y$*; sin embargo* $3x^2y + 2xy \not\approx 2xy + 3x^2y$*, pero* $2xy + 3x^2y \approx \frac{4}{2}xy + \frac{6}{2}x^2y$.

Traducción en **Coq**:

```
Inductive same_pol : pol -> pol -> Prop :=
 same_nil:   (same_pol vpol vpol)
| same_cons: (t:term)(full n t)->(c1,c2:K)(eqK c1 c2)->
   (p1,p2:pol)(same_pol p1 p2)->(same_pol (cpol (c1,t) p1) (cpol (c2,t) p2)).
```

Los siguientes resultados enuncian propiedades sobre la relación *same_pol* que se acaba de definir. Comenzamos demostrando que como esperábamos, *dicha relación es de equivalencia sobre polinomios de n variables*. Las pruebas de las propiedades reflexiva, simétrica y transitiva se hacen por recurrencia sobre el predicado *same_pol*.

Traducción en **Coq**:

```
Lemma same_refl : (p:pol)(full_term_pol p)->(same_pol p p).


Lemma same_sym : (p,q:pol)(same_pol p q)->(same_pol q p).


Lemma same_trans : (p,q,r:pol)(same_pol p q)->(same_pol q r)->(same_pol p r).
□
```

Ahora probamos que *la relación dada conserva los polinomios canónicos y el predicado low_pol*. La prueba, al igual que en los casos anteriores, se hace por

[9] Para especificar propiedades de elementos destacados de un polinomio.

recurrencia sobre el predicado *same_pol*, en este caso además debemos utilizar el constructor *full_pol_cons* y la hipótesis de recurrencia obtenida.

Traducción en **Coq**:

```
Lemma same_length : (p1,p2:pol)(same_pol p1 p2)->(full_pol p1)->
□ ((full_pol p2) /\ (u:term)(low_pol u p1)->(low_pol u p2)).
```

La demostración de que *la relación same_pol es más fina que equpol* es consecuencia de los constructores que definen *same_pol* y de resultados probados sobre la relación *equpol*.

```
Lemma same_equpol: (p,q:pol)(same_pol p q)->(equpol p q).
□
```

Otro resultado técnico que relaciona los predicados *equpol* y *same_pol* es que, *en el caso de polinomios canónicos, si dos polinomios verifican la relación equpol, entonces son similares.*

Prueba: Se utiliza recurrencia sobre $p1$ y $p2$, si ambos son *vpol* entonces son similares por el constructor *same_nil*. En los casos en que sólo uno de ellos sea *vpol*, se resuelve aplicando el teorema 6.6.8, debido a que en las hipótesis tenemos un polinomio canónico, del tipo $(cpol\ m\ p)$, igual a *vpol*. Cuando los polinomios canónicos son de la forma, $(cpol\ (c1, t1)\ q1)$ y $(cpol\ (c2, t2)\ q2)$, mediante la decidibilidad de la igualdad de términos obtenemos tres casos; si $(t1 < t2)$ o $(t2 < t1)$, de estas hipótesis y de la igualdad de los polinomios canónicos deducimos que $c2 =_K 0$ y $c1 =_K 0$, respectivamente, lo que contradice las hipótesis $c2 \neq_K 0$ $c1 \neq_K 0$ obtenidas por *Inversion* del hecho que ambos polinomios son canónicos. Por último, si se tiene que $t1 = t2$, de la igualdad de los polinomios canónicos se obtiene que $c1 =_K c2$ y, de la hipótesis de recurrencia, que $q1$ y $q2$ son similares; así, aplicando el constructor *same_cons* y propiedades ya demostradas sobre la igualdad de polinomios, se termina la demostración del resultado.

Traducción en **Coq**:

```
Lemma full_same: (p1,p2:pol)(equpol p1 p2)->(full_pol p1)->
□                                (full_pol p2)->(same_pol p1 p2).
```

A partir de este resultado se obtienen dos corolarios; el primero relativo a la forma canónica de los polinomios relacionados mediante *equpol*, y el segundo relativo a la forma canónica del producto de un polinomio por un monomio no nulo.

Traducción en **Coq**:

```
Lemma same_can: (p,q:pol)(equpol p q)->(full_term_pol p)->(full_term_pol q)->
                  (same_pol (fun_can p)(fun_can q)).


Lemma comm_mult_can: (p:pol)(full_term_pol p)->(m:monom)(full_mon n m)->
□      ~(z_monom m)->(same_pol (fun_can (mult_m p m))(mult_m (fun_can p) m)).
```

Ahora estamos en condiciones de enunciar y probar resultados sobre definiciones y operaciones algebraicas precedentes, que se conservan para polinomios similares. Las pruebas se hacen utilizando lemas anteriores, β-reducción y los constructores de la relación *same_pol*.

Teorema 6.7.1.

$$\forall p, q \in K[X];\ \forall t \in T;\ (p \approx q)\ \wedge\ (t \in p) \Rightarrow (t \in q)$$

Traducción en **Coq**:

```
Lemma term_in_pol_same: (p,q:pol)(t:term)(same_pol p q)->
       (term_in_pol p t)->(term_in_pol q t).
```

□

Teorema 6.7.2.

$$\forall p \in K[x_1, \ldots, x_n];\ (-p) \approx [p \cdot_M (-1_K, (x_1^0.x_2^0 \ldots x_n^0))]$$

Traducción en **Coq**:

```
Lemma pol_opp_mult: (p:pol)(full_term_pol p)->
   (same_pol (pol_opp p) (mult_m p ((multK_neg unK),(n_term_0 n)))).
```

□

6.8. Elementos destacados de un polinomio

Dado un *orden admisible* sobre términos, como es el lexicográfico, en cualquier polinomio en $K[X]$ existe un monomio cuyo término es mayor, respecto a ese orden dado (Ttm), que todos los demás. Adoptaremos la siguiente notación.

Definición 6.8.1. *El monomio principal de un polinomio $p \in K[X]$ con respecto a $<_L$ es el monomio de p cuyo término es el mayor en p. Denotaremos este monomio de la forma $M_L(p)$, o simplemente por $M(p)$ mientras se utilice el orden lexicográfico sobre términos. También se define $hterm(p)$ como el mayor término, y hcoef(p) como su coeficiente correspondiente. Así se tiene que*

$$\boxed{M(p) = (\mathrm{hcoef}(p), hterm(p))}$$

A partir del representante canónico de un polinomio p; se formaliza el monomio principal de la forma siguiente: si la clase de p es *vpol* se le asigna $(0, x_1^0 \ldots x_n^0)$, y en otro caso le asignamos el primer monomio de su representante canónico.

Traducción en **Coq**:

```
Definition hmonom: pol->monom :=[p:pol]Cases (fun_can p) of
     vpol => (OK,(n_term_0 n))
   | (cpol m q) => m
  end.

Definition hterm := [p:pol](mon_term (hmonom p)).

Definition hcoef := [p:pol](mon_coef (hmonom p)).
```

Ejemplo 6.8.1. *Sea el polinomio*

$$p = -2x^2yz + x^2y^2 + x^2z^2 + x^2y + 2xy^2z^2 - 3xyz^3 - xy + yz + z^2 - 4x^2y^2 + 5$$

un elemento de $\mathbb{Q}[x, y, z]$. *De su representante canónico con respecto al orden lexicográfico sobre términos,*

$$p = -2x^2y^2 - 2x^2yz + x^2y + x^2z^2 + 2xy^2z^2 - 3xyz^3 - xy + yz + z^2 + 5.$$

se obtiene, $M(p) = -2x^2y^2$, $hterm(p) = x^2y^2$, $hcoef = -2$

Considerando p como un elemento de $\mathbb{Q}[z, y, x]$, *entonces se tiene que su representante canónico con respecto a este nuevo orden de términos es,*

$$p = -3z^3yx + 2z^2y^2x + z^2x^2 + z^2 - 2zyx^2 + zy - 2y^2x^2 + yx^2 - yx + 5.$$

por consiguiente, $Mp) = -3z^3yx$, $hterm(p) = z^3yx$, $hcoef = -3$

Explicitando las formalizaciones anteriores y simplificándolas mediante β-reducción se comprueba fácilmente que se verifica la igualdad de la definición (6.8.1). Hay que resaltar que en este caso se utiliza la igualdad del sistema.

```
Lemma hmct_Leib: (f:pol) (hmonom f) = ((hcoef f),(hterm f)).
```
□

El lema siguiente prueba que la definición de coeficiente del término principal de un polinomio es coherente, como era de esperar, con la definición de coeficiente principal de dicho polinomio. La prueba se basa en la construcción de la función *fun_can* hecha en la sección 6.6 (página 130), la β-reducción mediante *Simpl* y la aplicación de resultados de coeficientes y cuerpos.

Lema 6.8.1. $\forall f \in K[x_1, \ldots, x_n]$; (hcoef f) $=_K$ (coef (*hterm* f) f)

Traducción en **Coq**:

```
Lemma coef_hterm:(f:pol)(full_term_pol f)->(eqK (hcoef f) (coef f (hterm f))).
```
□

Ahora verificamos dos resultados técnicos que se necesitan en la mayoría de la pruebas de esta sección, el primero "confirma" que *el término principal de un polinomio, no nulo, de n variables figura en un polinomio* y el segundo que *el término principal de un polinomio de n variables tiene n variables.* Para construir pruebas de estos resultados se utiliza la función que nos da el representante canónico del polinomio dado y se trabaja sobre él simplificando las definiciones dadas al comienzo de esta sección.

Traducción en **Coq**:

```
Lemma occ_n_vpol:(f:pol)(~(equpol f vpol))->(full_term_pol f)->
        (term_in_pol f (hterm f)).

Lemma full_hterm_full_pol: (f:pol)(full_term_pol f)->(full n (hterm f)).
```
□

Del lema (6.8.1) y de los resultados anteriores se obtiene la siguiente propiedad elemental.

Corolario 6.8.1.

$$\forall f \in K[x_1, \ldots, x_n];\ (f \neq_p vpol) \Rightarrow (\text{hcoef } f) \neq_K 0$$

Traducción en **Coq**:

```
Lemma cond_ht_pol: (f:pol)(full_term_pol f)->(~(equpol f vpol))->
        (~(eqK (hcoef f) OK)).
```

□

Nota: En esta prueba se utilizó la táctica **Change**, que implementa la regla de conversion **Conv**, ver [73]. *Change* (*term_in_pol f* (*hterm f*)) reemplaza la meta en curso $\sim$ (*eqK* (*coef p* (*hterm p*)) *OK*) por la nueva meta (*term_in_pol f* (*hterm f*)), ver definición (6.4.1).

Para monomios no nulos de n variables, los que se manejarán en lo que sigue, la igualdad *equmon*, dada en la sección 5.1.3, corresponde al siguiente predicado inductivo, que permitirá un uso más fácil del predicado de igualdad en el tipo *monom*.

Traducción en **Coq**:

```
Inductive equm: monom -> monom -> Prop :=
 equmoni: (c,c':K)(t:term)(full n t)->(eqK c c')->(equm (c,t)(c',t)).
```

Verificamos que la relación *equm* es de equivalencia.

Traducción en **Coq**:

```
Lemma equm_refl: (m:monom)(full_mon n m)->(equm m m).

Lemma equm_sym: (m,n:monom)(equm m n)->(equm n m).

Lemma equm_trans: (m,n,o:monom)(equm m n)->(equm n o)->(equm m o).
```

□

Por razones de comodidad y eficacia, implementamos una función que devuelve el primer elemento de un polinomio cualquiera. Al igual que *hmonom* si el polinomio es de tipo *vpol* devuelve $(0, x_1^0 \ldots x_n^0)$.

```
Definition fst_term:= [p:pol]Cases p of
                        vpol => (OK,(n_term_0 n))
                      | (cpol m _) => m
                      end.
```

La reflexividad del sistema garantiza que *la función fst_term aplicada a la expresión canónica de un polinomio, devuelve el monomio principal de dicho polinomio.* Hay que resaltar que en este caso se utiliza la igualdad del sistema, es decir se pueden reescribir directamente, mediante **Rewrite**, estas dos expresiones de tipo *monom*.

```
Lemma eq_homnom_fst: (p:pol)(hmonom p)=(fst_term (fun_can p)).
```

□

De la definición de polinomios similares se sigue fácilmente el siguiente resultado.

```
Lemma fst_equm: (p,q:pol)(same_pol p q)->(equm (fst_term p)(fst_term q)).
```
□

Para *polinomios canónicos de la forma* $((c,t) : q)$ *probamos que* (c,t), t *y* c *son respectivamente el monomio, término y coeficiente principal de dicho polinomio.* Las pruebas se basan en la aplicación de la función *fst_term* a dicho polinomio, resolviendo las metas generadas mediante los resultados anteriores sobre la relación *equm* y propiedades de los polinomios similares.
Nota: Se podrá comprobar más adelante que estos resultados simplifican muchas de las pruebas realizadas sobre propiedades de los elementos principales de los polinomios.

Traducción en **Coq**:

```
Lemma equm_full_hmonom: (c:K)(t:term)(q:pol)(full_pol (cpol (c,t) q))->
                        (equm (hmonom (cpol (c,t) q)) (c,t)).

Lemma full_hterm: (c:K)(t:term)(q:pol)(full_pol (cpol (c,t) q))->
                    (hterm (cpol (c,t) q))=t.

Lemma full_hcoef: (c:K)(t:term)(q:pol)(full_pol (cpol (c,t) q))->
                    (eqK (hcoef (cpol (c,t) q)) c).
```
□

Ahora, utilizando la función *fst_term* y aplicando el hecho de que dos polinomios canónicos que verifiquen la relación *equpol* son similares, probamos que *si dos polinomios de n variables son iguales, módulo equpol, entonces también son iguales sus monomios, términos y coeficientes principales.*

```
Lemma equpol_can: (p,q:pol)(equpol p q)->(full_term_pol p)->
                    (full_term_pol q)-> (equm (hmonom p)(hmonom q)).


Lemma eq_hterm: (h,h':pol)(full_term_pol h)->(full_term_pol h')->
                  (equpol h h')->(hterm h)=(hterm h').


Lemma eq_hcoef: (h,h':pol)(full_term_pol h)->(full_term_pol h')->
                  (equpol h h')->(eqK (hcoef h) (hcoef h')).
```
□

Como consecuencia de la reescritura, del resultado anterior *eq_hterm* y del lema *full_hterm* (página 140), se obtienen las tres siguientes propiedades elementales, de los coeficientes y términos principales.

Proposición 6.8.1.

$$\forall p, q \in K[x_1, \ldots, x_n];\ \forall t \in T;\ (p =_p q)\ \wedge\ [\ (hterm\ p) \mid t\ ] \Rightarrow (hterm\ q) \mid t$$

Traducción en **Coq**:

```
Lemma eq_term_div: (p,q:pol)(t:term)(full_term_pol p)->(full_term_pol q)->
       (equpol p q)->(term_div (hterm p) t)->(term_div (hterm q) t).
```
□

Proposición 6.8.2.

$$\forall t \in T^n;\ \forall k \in K;\ (k \neq_K 0) \Rightarrow [\ hterm\ ((k,t) : vpol)\ ] = t$$

Traducción en **Coq**:

```
Lemma term_monom_hterm: (k:K)(t:term)(full n t)->(~(eqK k OK))->
                        (hterm (cpol (k,t) vpol))=t.
```
□

Proposición 6.8.3. $\forall t \in T^n;\ \forall k \in K;\ k =_K [\ hcoef\ ((k,t) : vpol)\ ]$

Traducción en **Coq**:

```
Lemma eq_coef_monom:(k:K)(t:term)(full n t)->(eqK k (hcoef (cpol (k,t) vpol))).
```
□

A partir de estos resultados y utilizando la implementación de elementos destacados de un polinomio, la construcción de la forma canónica de un polinomio y el corolario 6.6.1 comprobamos que el coeficiente y término principal[10] del polinomio *vpol* son, respectivamente, 0 y $x_1^0 \ldots x_n^0$.

Proposición 6.8.4. *El coeficiente principal del polinomio vpol es* 0

Traducción en **Coq**:

```
Lemma hcoef_vpol: (eqK (hcoef vpol) OK).
```
□

Proposición 6.8.5. *El término principal del polinomio vpol es* $x_1^0 \ldots x_n^0$

Traducción en **Coq**:

```
Lemma funcanvpol: (p:pol)((fun_can p) = vpol)->(hterm p)=(n_term_0 n).


Lemma hterm_vpol: (hterm vpol) = (n_term_0 n).
```
□

De la proposición anterior se obtienen dos resultados muy útiles que relacionan el término principal de un polinomio p con la decidibilidad de dicho polinomio respecto al polinomio *vpol*. Queremos resaltar que estos lemas técnicos no se explicitan en las demostraciones matemáticas, pero para nuestras pruebas "formales" son imprescindibles.

Corolario 6.8.2.

$$\forall p \in K[x_1, \ldots, x_n];\ [(hterm\ p) \neq (x_1^0 \ldots x_n^0)] \Rightarrow (p \neq_p vpol)$$

Prueba: Es consecuencia del resultado anterior y la extensionalidad de la definición de *hterm* respecto a la relación *equpol*.

Traducción en **Coq**:

```
Lemma contr_hterm_vpol: (p:pol)(full_term_pol p)->
                        ~((hterm p)= (n_term_0 n))->~(equpol p vpol).
```
□

[10] En la representación matemática suele seguirse está notación, ver [54].

Corolario 6.8.3.

$$\forall f, h \in K[x_1, \ldots, x_n];\ [(hterm\ h) <_L (hterm\ f)] \Rightarrow (f \neq_p vpol)$$

Prueba: Se demuestra utilizando el corolario anterior así como las propiedades de no simetría y no reflexividad del orden lexicográfico $<_L$ probadas en los teoremas (3.4.4 y 3.4.5), respectivamente.

Traducción en **Coq**:

```
Lemma gr_ht_nvpol: (f,h:pol)(full_term_pol f)->(full_term_pol h)->
                   (Ttm (hterm h) (hterm f))->(~(equpol f vpol)).
```

□

De la axiomatización del concepto de término principal de un polinomio se derivan las tres propiedades siguientes, que hacen referencia a que dicho término es el mayor de los términos de un polinomio dado, respecto al orden lexicográfico. Serán utilizadas en las pruebas de la reducción de polinomios.

Lema 6.8.2.

$$\forall f \in K[x_1, \ldots, x_n];\ (\forall t \in f)\ \wedge\ [t \neq (hterm\ f)] \Rightarrow t <_L (hterm\ f)$$

Prueba: Se prueba en primer lugar un resultado auxiliar, utilizando un polinomio canónico. De dicho resultado auxiliar, aplicando el teorema 6.6.9 y la compatibilidad respecto a *equpol* de la presencia de un término en un polinomio, se obtiene la prueba del lema enunciado. La prueba del lema auxiliar se hace por inducción sobre el polinomio f, la submeta correspondiente al caso *vpol* se resuelve directamente a partir de la formalización de *term_in_pol* y la segunda submeta mediante lemas relativos a los coeficentes de los términos de un polinomio y la hipótesis de que el polinomio está ordenado decrecientemente. Incluimos el detalle de la prueba para ilustrar cada paso.

Traducción en **Coq**:

```
Lemma full_Ttm_pol_hterm: (f:pol)(full_pol f)->(t:term)(term_in_pol f t)->
                          ~t=(hterm f)->(Ttm t (hterm f)).

Induction f.
Intros.
Elim H0; Auto.
Induction m; Intros.
Cut (hterm (cpol (y,y0) p))=y0; Intros; Auto.
Rewrite H3.
Rewrite H3 in H2.
Inversion H0.
Cut (term_in_pol p t); Intros.
Apply term_in_Ttm_pol with p; Trivial.
Cut t=y0\/(term_in_pol p t); Intros.
Elim H11; Intros; Trivial.
Elim H2; Auto.
Apply split_term_in_pol_cons with y; Auto.


Lemma Ttm_pol_hterm: (f:pol)(full_term_pol f)->(t:term)(term_in_pol f t)->
                     (~(t=(hterm f)))->(Ttm t (hterm f)).
```

```
Intros.
Elim (can_fun f); Intros; Auto.
Elim y; Intros; Clear y.
Replace (hterm f) with (hterm x); Auto.
Apply full_Ttm_pol_hterm; Trivial.
Apply comp_term_in_pol with f; Auto.
Replace (hterm x) with (hterm f); Auto.
□
```

Lema 6.8.3.

$$\forall f \in K[x_1, \ldots, x_n];\ \forall t \in T^n;\ [(hterm\ f) <_L t)] \Rightarrow t \notin f$$

Prueba: Se obtiene la prueba por contradicción, utilizando el lema anterior, las propiedades de que el orden $(<_L)$ es estricto y asimétrico, junto con el teorema (6.4.8).

Traducción en **Coq**:

```
Lemma Ttm_hterm: (f:pol)(full_term_pol f)->(t:term)(full n t)->
                    (Ttm (hterm f) t)->(no_term_in_pol f t).
□
```

Lema 6.8.4.

$$\forall f \in K[x_1, \ldots, x_n];\ \forall t \in f;\ [\forall t' \in f;\ (t \neq t') \Rightarrow (t' <_L t)] \Rightarrow t = (hterm\ f)$$

Prueba: Se demuestra por contradicción. Supongamos que $t \neq (hterm\ f)$. De la hipótesis del contexto $H : [\forall t' \in f;\ (t \neq t') \Rightarrow (t' <_L t)]$ y, utilizando el hecho que el término principal de un polinomio no nulo está presente en un polinomio, se obtiene $[(hterm\ f) <_L t]$. Por otra parte del lema (6.8.2), deducimos la desigualdad $[t <_L (hterm\ f)]$. De estas nuevas hipótesis y utilizando la propiedad de asimetría, *Ttm_antisym*, del orden lexicográfico $(<_L)$ se construye la prueba del lema.

Traducción en **Coq**:

```
Lemma aux1_hterm_suma_Ttm: (f:pol)(full_term_pol f)->
   (t:term)(term_in_pol f t)->
      ((t':term)(term_in_pol f t')->(~t=t')->(Ttm t' t))->(t=(hterm f)).
□
```

A continuación se enuncian y se construyen pruebas de una serie de lemas sobre la relación que hay entre polinomios y sus opuestos, con respecto al término y coeficiente principal. El esquema común de prueba es demostrar, en primer lugar, el resultado para polinomios canónicos, utilizando los resultados sobre términos y coeficientes principales obtenidos en la página 140; después se extienden fácilmente a polinomios cualesquiera de n variables, utilizando los lemas auxiliares anteriores sobre los representantes canónicos de los polinomios, que se pueden obtener mediante la función *can_fun*.

Lema 6.8.5.

$$\forall f \in K[x_1, \ldots, x_n];\quad (hterm\ f) = [hterm\ (-f)]$$

Traducción en **Coq**:

```
Lemma hterm_pol_opp: (f:pol)(full_pol f)->((hterm f)=(hterm (pol_opp f))).


Lemma hterm_pol_opp_term: (f:pol)(full_term_pol f)->
                          ((hterm f)=(hterm (pol_opp f))).
```

□

Lema 6.8.6.

$$\forall f \in K[x_1, \ldots, x_n]; \quad (hcoef\ f) =_K -[hcoef\ (-f)]$$

Traducción en **Coq**:

```
Lemma hcoef_pol_opp:(f:pol)(full_pol f)->
                    (eqK (hcoef f) (multK_neg (hcoef (pol_opp f)))).

Lemma hcoef_pol_opp_term: (f:pol)(full_term_pol f)->
                          (eqK (hcoef f) (multK_neg (hcoef (pol_opp f)))).
```

□

Como corolarios de los lemas anteriores demostramos que *si dos polinomios tienen sus coeficientes y términos principales iguales, también lo son los de sus respectivos polinomios opuestos.*

```
Lemma eq_pol_opp_hcoef: (f,g:pol)(full_term_pol f)->(full_term_pol g)->
   (eqK (hcoef f) (hcoef g))->(eqK (hcoef (pol_opp f)) (hcoef (pol_opp g))).


Lemma eq_pol_opp_hterm: (f,g:pol)(full_term_pol f)->(full_term_pol g)->
           (hterm f)=(hterm g)->(hterm (pol_opp f))=(hterm (pol_opp g)).
```

□

Los lemas siguientes formulan propiedades "formales"[11] de los coeficientes y términos principales respecto a la suma de polinomios. Estos resultados serán indispensables para realizar las pruebas en la reducción (división) de polinomios. Todos se demuestran fácilmente (utilizando la decidibilidad de términos y la presencia en un polinomio así como resultados del comienzo de esta sección).

Lema 6.8.7.

$$\forall f, h \in K[x_1, \ldots, x_n]; \ [(hterm\ h) <_L (hterm\ f)] \Rightarrow (hterm\ f) \in (f +_p h)$$

Traducción en **Coq**:

```
Lemma aux_hterm_suma_Ttm: (f,h:pol)(full_term_pol f)->(full_term_pol h)->
   (Ttm (hterm h) (hterm f))->(term_in_pol (suma_app f h) (hterm f)).
```

□

Lema 6.8.8. $\forall f, h \in K[x_1, \ldots, x_n]$, *se verifica que:*

$$[(hterm\ h) <_L (hterm\ f)] \Rightarrow (hterm\ f) = [hterm\ (f +_p h)]$$

[11] Propiedades tan simples que a menudo parecen rebuscadas y que no suelen aparecer en las demostraciones matemáticas con lápiz y papel, pero para construir en este contexto intuicionista es preciso disponer de todas ellas.

Traducción en **Coq**:

```
Lemma hterm_suma_Ttm: (f,h:pol)(full_term_pol f)->(full_term_pol h)->
          (Ttm (hterm h) (hterm f))->(hterm (suma_app f h))=(hterm f).
□
```

Lema 6.8.9. $\forall f, h \in K[x_1, \ldots, x_n]$, *se verifica que:*

$$[(hterm\ h) <_L (hterm\ f)] \Rightarrow [hcoef\ (f +_p h)] =_K (hcoef\ f)$$

Traducción en **Coq**:

```
Lemma hcoef_suma_Ttm: (f,h:pol)(full_term_pol f)->(full_term_pol h)->
     (Ttm (hterm h) (hterm f))->(eqK (hcoef (suma_app f h)) (hcoef f)).
□
```

Lema 6.8.10.

$$\forall f, h \in K[x_1, \ldots, x_n];\ \forall t \in T^n;\ (t \in f) \wedge$$
$$[\ \forall t_1 \in T;\ (t_1 \in h) \Rightarrow (hterm\ f) <_L t_1)\ ] \Rightarrow t \in (f +_p h)$$

Traducción en **Coq**:

```
Lemma term_in_Ttm_hterm: (f,h:pol)(full_term_pol f)->(full_term_pol h)->
   (t:term)(term_in_pol f t)->((t1:term)(term_in_pol h t1)->
      (Ttm (hterm f) t1))->(term_in_pol (suma_app f h) t).
□
```

Lema 6.8.11.

$$\forall f, h \in K[x_1, \ldots, x_n];\ \forall t \in T^n;\ (t \in f) \wedge$$
$$[\ \forall t_1 \in T;\ (t_1 \in h) \Rightarrow (hterm\ f) <_L t_1)\ ] \Rightarrow (coef\ t\ f) =_K [\ coef\ t\ (f +_p h)\ ]$$

Traducción en **Coq**:

```
Lemma coef_suma_Ttm: (f,h:pol)(full_term_pol f)->(full_term_pol h)->
   (t:term)(term_in_pol f t)->((t1:term)(term_in_pol h t1)->
      (Ttm (hterm f) t1))->(eqK (coef f t) (coef (suma_app f h) t)).
□
```

Lema 6.8.12.

$$\forall f, g \in K[x_1, \ldots, x_n];\ [(hterm\ f) = (hterm\ g)] \wedge$$
$$[\ \forall t \in T^n;\ t \in (f +_p g)]\ \wedge\ [t \neq (hterm\ f)\ ] \Rightarrow t <_L (hterm\ f)$$

Traducción en **Coq**:

```
Lemma aux1_hterm_eq_suma_Ttm: (f,g:pol)(full_term_pol f)->(full_term_pol g)->
              (hterm f)=(hterm g)->(t:term)(term_in_pol (suma_app f g) t)->
                  (~(t=(hterm f)))->(Ttm t (hterm f)).
□
```

Lema 6.8.13. $\forall f, g \in K[x_1, \ldots, x_n]$, *se verifica que:*

$$[\ (hterm\ f) = (hterm\ g)\ ]\ \wedge\ [\ (hterm\ f) = (hterm\ (f +_p g))\ ] \Rightarrow$$
$$[hcoef\ (f +_p g)] =_K (hcoef\ f) +_K (hcoef\ g)$$

Traducción en **Coq**:

```
Lemma aux2_hterm_eq_suma_Ttm: (f,g:pol)(full_term_pol f)->(full_term_pol g)->
   (hterm f)=(hterm g)->(hterm f)=(hterm (suma_app f g))->
      (eqK (hcoef (suma_app f g)) (plusK (hcoef f) (hcoef g))).
□
```

Lema 6.8.14.

$$\forall f, g \in K[x_1, \ldots, x_n];\ [\ (hterm\ f) = (hterm\ g)\ ] \wedge [\ (f +_p g) \neq_p vpol\ ] \wedge [\ (hcoef\ f) =_K -(hcoef\ g)\ ] \Rightarrow [\ (hterm\ (f +_p g)\ ] <_L (hterm\ f)$$

Traducción en **Coq**:

```
Lemma hterm_eq_suma_Ttm:(f,g:pol)(full_term_pol f)->(full_term_pol g)->
   ~(equpol (suma_app f g) vpol)->(eqK (hcoef f) (multK_neg (hcoef g)))->
      (hterm f)=(hterm g)->(Ttm (hterm (suma_app f g)) (hterm f)).
□
```

Lema 6.8.15.

$$\forall f, g \in K[x_1, \ldots, x_n];\ [\ (hterm\ f) = (hterm\ g)\ ] \wedge [\ f, g \neq_p vpol\ ] \wedge [\ (hcoef\ f) \neq_K -(hcoef\ g)\ ] \Rightarrow [\ (hterm\ (f +_p g)\ ] =_L (hterm\ f)$$

Traducción en **Coq**:

```
Lemma hcoef_no_eq_hterm: (f,g:pol)(full_term_pol f)->(full_term_pol g)->
  ~(equpol f vpol)->~(equpol g vpol)->(hterm f)=(hterm g)->
     ~(eqK (hcoef f) (multK_neg (hcoef g)))->(hterm f)=(hterm (suma_app f g)).
□
```

Lema 6.8.16.

$$\forall f, g \in K[x_1, \ldots, x_n];\ \forall t \in T^n;\ [\ (f +_p g) \neq_p vpol\ ] \wedge [\ (hterm\ f) <_L t\ ] \wedge [\ (hterm\ g) <_L t\ ] \Rightarrow [\ (hterm\ (f +_p g)\ ] <_L t$$

Traducción en **Coq**:

```
Lemma Ttm_suma_hterm: (f,g:pol)(t:term)(full_term_pol f)->(full_term_pol g)->
   (full n t)->~(equpol (suma_app f g) vpol)->
      (Ttm (hterm f) t)->(Ttm (hterm g) t)->(Ttm (hterm (suma_app f g)) t).
□
```

Para finalizar esta sección se prueban resultados, similares a los lemas precedentes, pero en este caso relativos al producto de un polinomio por un monomio. Al igual que los anteriores, serán esenciales en la reducción de polinomios.

Lema 6.8.17.

$$\forall p \in K[x_1, \ldots, x_n];\ \forall m \in M_{K,n};\ \neg(z\_monom\ m) \wedge (p \neq_p vpol) \Rightarrow M(p\ ._M\ m) =_M m\ .\ M(p)$$

Prueba: En primer lugar se demuestra un lema auxiliar directamente, por inducción y simplificación de expresiones, que es el mismo resultado pero utilizando la función *fst_term*[12] anteriormente implementada. El lema se prueba reescribiendo (ver lema *eq_homnom_fst*) la implementación de $M(p)$ mediante la función *fst_term* y utilizando el lema auxiliar sobre el representante canónico del polinomio dado, así como la transitividad de las relaciones *equm* y *equpol*.

Traducción en **Coq**:

[12] Extrae el primer elemento de un polinomio cualquiera.

```
Lemma fst_mult: (p:pol)(full_term_pol p)->~(equpol p vpol)->
                (m:monom)(full_mon n m)->~(z_monom m)->
                   (equm (fst_term (mult_m p m))(mult_mon m (fst_term p))).


Lemma hmonom_mult: (p:pol)(full_term_pol p)->~(equpol p vpol)->
      (m:monom)(full_mon n m)->~(z_monom m)->
            (equm (hmonom (mult_m p m))(mult_mon m (hmonom p))).
```
□

Ahora, se pueden probar los dos corolarios siguientes. Ambas pruebas son análogas y se resuelven utilizando los resultados anteriores sobre la relación *equm* y reescribiendo lemas relativos a las formalizaciones de los elementos destacados de un polinomio, vistos al comienzo de esta sección.

Corolario 6.8.4. $\forall f \in K[x_1, \ldots, x_n]$ *y* $\forall m \in M_{K,n}$, *se verifica que:*

$$[\text{hcoef } (f \cdot_M m)] =_K [(\text{hcoef } f) \cdot_K (\text{coef } m)]$$

Traducción en **Coq**:

```
Lemma hcoef_mult: (f:pol)(full_term_pol f)->(m:monom)(full_mon n m)->
              (eqK (hcoef (mult_m f m)) (multK (hcoef f) (mon_coef m))).
```
□

Corolario 6.8.5.

$$\forall f \in K[x_1, \ldots, x_n];\ \forall m \in M_{K,n};\ (f \neq_p vpol) \wedge \neg(z\_monom\ m) \Rightarrow hterm\ (f \cdot_M m) = (hterm\ f)\ .\ (term\ m)$$

Traducción en **Coq**:

```
Lemma hterm_mult_m: (f:pol)(full_term_pol f)->(m:monom)(full_mon n m)->
   ~(equpol f vpol)->(~(z_monom m))->
      (hterm (mult_m f m))=(term_mult (hterm f) (mon_term m)).
```
□

A su vez de los dos últimos corolarios se obtienen seis nuevos corolarios; las pruebas de todos ellos se obtienen directamente aplicando los corolarios anteriores y resultados relativos a operaciones de términos y elementos del cuerpo K.

Corolario 6.8.6.

$$\forall f \in K[x_1, \ldots, x_n];\ \forall m \in M_{K,n};\ (f \neq_p vpol) \wedge \neg(z\_monom\ m) \Rightarrow hterm\ (f \cdot_M m) = hterm\ [\ f \cdot_M (-m)\ ]$$

Traducción en **Coq**:

```
Lemma hterm_pol_m_opp: (f:pol)(full_term_pol f)->(m:monom)(full_mon n m)->
   ~(equpol f vpol)->~(z_monom m)->
      (hterm (mult_m f m))=(hterm (mult_m f (monom_op m))).
```
□

Corolario 6.8.7.

$$\forall f \in K[x_1, \ldots, x_n];\ \forall m \in M_{K,n};\ (f \neq_p vpol) \wedge \neg(z\_monom\ m) \Rightarrow (hterm\ f)\ |\ [\ hterm\ (\ f \cdot_M m)\ ]$$

Traducción en **Coq**:

```
Lemma term_div_hterm: (f:pol)(full_term_pol f)->(m:monom)~(z_monom m)->
  (full_mon n m)->~(equpol f vpol)->(term_div (hterm f) (hterm (mult_m f m))).
□
```

Corolario 6.8.8.

$$\forall f \in K[x_1,\ldots,x_n];\ \forall y \in K;\ \forall y_0 \in T^n;\ (f \neq_p vpol) \wedge \neg[\, z\_monom\ (y,y_0)\,] \Rightarrow y_0 = \frac{hterm\ [\, f\ ._M\ (y,y_0)\,]}{hterm\ f}$$

Traducción en **Coq**:

```
Lemma div_term_hterm: (f:pol)(full_term_pol f)->~(equpol f vpol)->
   (y:K)(y0:term)(full n y0)->~(z_monom (y,y0))->
      y0=(div_term (hterm (mult_m f (y,y0))) (hterm f)).
□
```

Corolario 6.8.9.

$$\forall f \in K[x_1,\ldots,x_n];\ \forall t \in T^n;\ \forall k \in K;\ (f \neq_p vpol)\ \wedge\ (k \neq_K 0) \wedge (hterm\ f) \mid t\ \Rightarrow t = hterm\ \left[\ f\ ._M\ (k, \tfrac{t}{hterm\ f})\ \right]$$

Traducción en **Coq**:

```
Lemma hterm_mult_m2: (f:pol)(full_term_pol f)->~(equpol f vpol)->
   (k:K)(~(eqK k OK))->(t:term)(full n t)->(term_div (hterm f) t)->
        (hterm (mult_m f (k,(div_term t (hterm f)))))=t.
□
```

Corolario 6.8.10.

$$\forall f \in K[x_1,\ldots,x_n];\ \forall t \in T^n;\ \forall k \in K;\ hcoef\ \left[\ f\ ._M\ (k, \tfrac{t}{hterm\ f})\ \right] =_K (hcoef\ f)\ ._K\ \left[\ coef\ (k, \tfrac{t}{hterm\ f})\ \right]$$

Traducción en **Coq**:

```
Lemma hcoef_mult2: (f:pol)(full_term_pol f)->(t:term)(full n t)->
   (k:K)(eqK (hcoef (mult_m f (k,(div_term t (hterm f)))))
             (multK (hcoef f) (mon_coef (k,(div_term t (hterm f)))))).
□
```

Corolario 6.8.11.

$$\forall f \in K[x_1,\ldots,x_n];\ \forall a \in K;\ \forall t,t',t_0 \in T^n;\ (f \neq_p vpol)\ \wedge\ (a \neq_K 0) \wedge (t_0 <_L t)\ \wedge\ [\,(hterm\ f) \mid t_0\,]\ \wedge\ t' \in \left[\ f\ ._M\ (\tfrac{a}{hcoef\ f}, \tfrac{t_0}{hterm\ f})\ \right] \Rightarrow t' <_L t$$

Traducción en **Coq**:

```
Lemma term_in_Ttm: (fj:pol)(t,t',t0:term)(a:K)~(eqK a OK)->(full_term_pol fj)->
   (full n t)->(full n t')->(full n t0)->~(equpol fj vpol)->(Ttm t0 t)->
      (term_div (hterm fj) t0)->
  (term_in_pol (mult_m fj ((divK a (hcoef fj)),(div_term t0 (hterm fj)))) t')->
       (Ttm t' t).
□
```

Capítulo 7

Orden sobre polinomios

Precisamos implementar una "reducción" sobre polinomios, la cual puede ser interpretada como un tipo de "congruencia dirigida", reemplazar polinomios por otros "más pequeños". Por ello en este capítulo se implementa, en primer lugar, un orden estricto sobre polinomios cualesquiera, basado únicamente en el orden admisible de los términos del polinomio; se verifican a continuación sus principales propiedades. Para demostrar que este orden es **bien fundado** se formaliza el tipo polinomio canónico y sobre este tipo se restringe el orden anterior.

7.1. Definición

Cualquier orden admisible sobre términos se puede extender a un orden estricto parcial sobre polinomios. En nuestro caso extendemos el orden lexicográfico sobre términos, que hemos probado que es admisible, de la manera siguiente.

Definición 7.1.1. *Sea $<_L$ el orden lexicográfico sobre términos (T^n) (ver, definición 3.4.5), y dados dos polinomios $p, q \in K[x_1, \ldots, x_n]$,*

$$p \prec q :\Longleftrightarrow \begin{cases} hterm\,(p) <_L hterm\,(q) \\ o \\ \exists t \in T^n;\ [H(p,t) = H(q,t) \ \wedge\ t \notin p \ \wedge\ t \in q] \end{cases}$$

donde:

$$H(p,t) := \sum_{(u \in p) \ \wedge \ (t <_L u)} (coef\ u\ p).u$$

La traducción en Coq, se hace mediante la formalización del siguiente predicado inductivo:

```
Inductive Tpol_Lex3: pol -> pol -> Prop :=
  Tpol_Lex3_v: (m:monom)(p:pol)(full_mon n m)->(full_term_pol p)->
                 (Tpol_Lex3 vpol (cpol m p))
```

```
|Tpol_Lex3_car: (m1,m2:monom)(p1,p2:pol)(full_mon n m1)->(full_mon n m2)->
  (full_term_pol p1)->(full_term_pol p2)->(Ttm (mon_term m1) (mon_term m2))->
                (Tpol_Lex3 (cpol m1 p1) (cpol m2 p2))
|Tpol_Lex3_cdr: (k1,k2:K)(t:term)(p1,p2:pol)(full n t)->
   (full_term_pol p1)->(full_term_pol p2)->(Tpol_Lex3 p1 p2)->
                (Tpol_Lex3 (cpol (k1,t) p1) (cpol (k2,t) p2)).
```

Esta relación juega un papel central en la teoría que se desarrolla en los siguientes capítulos, por eso vamos a describirla explícitamente. Si los términos de los polinomios p y q son iguales entonces $(p \not\prec q)$ y $(q \not\prec p)$. Si tienen términos diferentes, buscamos los términos principales de p y q; si son diferentes, su orden será decisivo $(p \prec q :\Longleftrightarrow hterm\ (p) <_L hterm\ (q))$. Si son iguales, los eliminamos de p y q y repetimos el proceso, comparando el resto de los términos. En caso de que uno de los polinomios se transforme en el polinomio *vpol* antes de alcanzar una decisión, sería el otro polinomio distinto de *vpol* el "ganador". La implementación anterior en Coq tiene tres constructores que reflejan los pasos descritos.

En general la extensión de $<_L$ a $\prec$ no es un orden total. Sin embargo probaremos que es bien fundado, lo cual es fundamental para el algoritmo de Buchberger.

Ejemplo 7.1.1. *Sean $<_L$ el orden lexicográfico definido sobre (T^n) por $(y <_L x)$. Entonces se verifica que*

$$p =_p (9xy^2 + 3xy^2 - y^3 + 7) \prec q =_p (6x^3y + 4x^2y + 4y^3 - 1)$$

pues $hterm\ (p) = 9xy^2 <_L hterm\ (p) = 6x^3y$.

Por otra parte se verifica que

$$p =_p (9x^3y + 3xy^2 - y^3 + 7) \prec q =_p (6x^3y - xy^2 + 4x^2y + 4y^3 - 1)$$

pues $y^3 <_L x^2y$ que son los términos diferentes de mayor grado en los polinomios p y q.

Sin embargo los polinomios $p =_p 4x^3y^2 - x^2y^2 + 2y^3 - 6$, $q =_p x^3y^2 - 6x^2y^2 + 2y^3 - 6$ son distintos y se verifica que $(p \not\prec q)$ y $(q \not\prec p)$, es decir no están relacionados mediante la relación $\prec$.

Comprobamos que *la relación que acabamos de definir, es estable por la relación equpol sobre polinomios canónicos*, que son los que se utilizarán en las demostraciones del teorema de Buchberger. La prueba se hace por recurrencia estructural sobre la relación $Tpol\_Lex3$, generando tres casos, uno por cada constructor. En cada uno de estos casos utilizamos la inducción sobre el polinomio equivalente mediante *equpol*; las seis submetas generadas se resuelven combinado la aplicación de los constructores de la relación, las hipótesis de inducción y resultados probados sobre los monomios de polinomios en forma canónica. A modo de ejemplo ilustrativo, se incluye el código de la prueba.

```
Lemma ext_l_Tpol_Lex3: (p,q:pol)(Tpol_Lex3 p q)->(full_pol p)->(full_pol q)->
                       (r:pol)(equpol p r)->(full_pol r)->(Tpol_Lex3 r q).
```

```
Intros p q.
Induction 1.
Induction m.
Induction r; Auto.
Induction m0; Intros.
Cut (cpol (a0,b0) p1)=vpol; Intros.
Rewrite H7; Auto.
Apply full_eq_vpol; Auto.
Induction m1; Induction m2.
Induction r; Auto.
Induction m; Intros.
Simpl in H4.
Cut b=b1; Intros.
Apply Tpol_Lex3_car; Auto.
Simpl.
Inversion H9; Auto.
Inversion H9; Auto.
Simpl.
Rewrite <- H10; Auto.
Transitivity (hterm (cpol (a,b) p1)).
Symmetry; Auto.
Transitivity (hterm (cpol (a1,b1) p0)); Auto.
Induction r; Auto.
Induction m; Intros.
Cut t=b; Intros.
Rewrite H10.
Apply Tpol_Lex3_cdr; Trivial.
Inversion H9; Auto.
Inversion H9; Auto.
Apply H4.
Inversion H9; Auto.
Inversion H5; Auto.
Inversion H6; Auto.
Rewrite H10 in H8.
Cut (eqK k1 a); Intros.
Apply eq_cpol with (a,b); Auto.
Apply equpol_tran with (cpol (k1,b) p1); Auto.
Apply eqK_trans with (hcoef (cpol (k1,t) p1)); Auto.
Apply eqK_trans with (hcoef (cpol (a,b) p0)); Auto.
Apply eq_hcoef; Auto.
Rewrite H10; Auto.
Inversion H9; Auto.
Transitivity (hterm (cpol (k1,t) p1)).
Symmetry; Auto.
Transitivity (hterm (cpol (a,b) p0)); Auto.


Lemma ext_r_Tpol_Lex3: (p,q:pol)(Tpol_Lex3 p q)->(full_pol p)->(full_pol q)->
                       (r:pol)(equpol q r)->(full_pol r)->(Tpol_Lex3 p r).

Intros p q.
Induction 1.
Induction m.
Induction r.
Intros.
Inversion H3.
```

```
Cut (eqK a OK); Intros.
Elim H9; Auto.
Apply eqK_trans with (hcoef (cpol (a,b) p0)); Auto.
Cut (cpol (a,b) p0)=vpol; Intros.
Rewrite H13.
Apply hcoef_vpol.
Apply full_eq_vpol; Auto.
Induction m0; Intros.
Apply Tpol_Lex3_v.
Simpl.
Inversion H6; Auto.
Inversion H6; Auto.
Induction m1; Induction m2.
Induction r.
Intros.
Simpl in H4.
Cut (cpol (a0,b0) p2)=vpol; Intros.
Rewrite <- H9.
Apply Tpol_Lex3_car; Auto.
Apply full_eq_vpol; Auto.
Induction m; Intros.
Simpl in H4.
Cut b0=b1; Intros.
Rewrite <- H10.
Apply Tpol_Lex3_car; Auto.
Inversion H9; Auto.
Transitivity (hterm (cpol (a0,b0) p2)).
Symmetry; Auto.
Transitivity (hterm (cpol (a1,b1) p0)); Auto.
Induction r.
Intros.
Inversion H6.
Cut (eqK k2 OK); Intros.
Elim H12; Auto.
Apply eqK_trans with (hcoef (cpol (k2,t) p2)); Auto.
Cut (cpol (k2,t) p2)=vpol; Intros.
Rewrite H16.
Apply hcoef_vpol.
Apply full_eq_vpol; Auto.
Induction m; Intros.
Cut t=b; Intros.
Rewrite H10.
Apply Tpol_Lex3_cdr; Auto.
Inversion H9; Auto.
Inversion H9; Auto.
Apply H4.
Inversion H5; Auto.
Inversion H6; Auto.
Rewrite H10 in H8.
Cut (eqK k2 a); Intros.
Apply eq_cpol with (a,b); Auto.
Apply equpol_tran with (cpol (k2,b) p2); Auto.
Apply eqK_trans with (hcoef (cpol (k2,t) p2)); Auto.
Apply eqK_trans with (hcoef (cpol (a,b) p0)); Auto.
Apply eq_hcoef; Auto.
Rewrite H10; Auto.
Inversion H9; Auto.
```

```
Transitivity (hterm (cpol (k2,t) p2)).
Symmetry; Auto.
Transitivity (hterm (cpol (a,b) p0)); Auto.
□
```

7.2. Propiedades

En esta sección verificamos que la relación anterior es un orden estricto y nos interesamos particularmente por una serie de propiedades de la relación $Tpol\_Lex3$ concernientes a polinomios canónicos, que son indisociables de la definición de $\prec$. Así como hemos demostrado una gran cantidad de lemas básicos para formalizar polinomios, los resultados que presentamos sobre las relaciones entre los predicados $full\_pol$ y $Tpol\_Lex3$ son indispensables para la formalización del tipo polinomio canónico, en el sentido de que para utilizar "confortablemente" un nueva noción, a menudo es necesario demostrar propiedades elementales referidas a ella.

Teorema 7.2.1. *El orden $\prec$ sobre $K[x_1, \ldots, x_n]$ verifica la propiedad irreflexiva.*

$$\forall p \in K[x_1, \ldots, x_n];\ p \not\prec p$$

Prueba: En primer lugar hacemos inducción sobre el polinomio p e introducimos la negación en las hipótesis; la primera submeta se resuelve con el constructor $Tpol\_Lex3\_v$, por **Inversion** sobre la hipótesis $(vpol \prec vpol)$. Para resolver la segunda se aplica **Inversion** sobre $(cpol\ m\ p0) \prec (cpol\ m\ p0)$, generando así tres nuevas submetas. La primera se resuelve aplicando la táctica **Discriminate** sobre la hipótesis de dos términos estructuralmente diferentes $(vpol = (cpol\ m\ p0))$ y las otras dos restantes, utilizando la hipótesis de inducción y que el orden $<_L$ sobre términos verifica la propiedad irreflexiva.

Traducción en **Coq**:

```
Theorem Tpol_Lex3_no_refl: (p:pol) ~(Tpol_Lex3 p p).
□
```

Teorema 7.2.2. *El orden $\prec$ sobre $K[x_1, \ldots, x_n]$ verifica la propiedad transitiva.*

$$\forall p, q, r \in K[x_1, \ldots, x_n];\ (p \prec q) \wedge (q \prec r) \Rightarrow p \prec r$$

Prueba: Por recurrencia estructural sobre la relación $(p \prec q)$ obtenemos tres submetas correspondientes a los constructores de $Tpol\_Lex3$:

- Caso $vpol \prec r$: se resuelve por casos respecto al polinomio r
  - Caso $r = vpol$. Por contradicción aplicando la táctica **Inversion** sobre la hipótesis $H1 : ((cpol\ (a, b)\ p0) \prec vpol)$.
  - La meta $(vpol \prec (cpol\ (a0, b0)\ p1))$ se resuelve directamente por el primer constructor de la relación $\prec$.

- Para resolver la segunda y tercera se utiliza **Inversion** sobre $(q \prec r)$. Las metas generadas se resuelven por la transitividad de la relación Ttm sobre las igualdades obtenidas mediante la táctica **Injection**, que corresponden a los constructores de $Tpol\_Lex3$.

Traducción en **Coq**:

```
Theorem Tpol_Lex3_tran: (p,q,r:pol)(Tpol_Lex3 p q)->
                        (Tpol_Lex3 q r)->(Tpol_Lex3 p r).
□
```

A continuación, como cabría esperar, se constata la coherencia entre las formalizaciones de los predicados $full\_pol$ y $Tpol\_Lex3$. Las pruebas se hacen por inversion de dichas formalizaciones y aplicando en cada caso sus constructores así como las propiedades de la relación $<_L$.

Lema 7.2.1.

$$\forall m \in M_{K,n};\ \forall p \in K[x_1, \ldots, x_n];\ [full\_pol\ (m : p)] \Rightarrow p \prec (m : vpol)$$

Traducción en **Coq**:

```
Theorem ext_full_Tpol: (m:monom)(p:pol)(full_pol (cpol m p))->
                       (Tpol_Lex3 p (cpol m vpol)).
□
```

Lema 7.2.2.

$$\forall m \in M_{K,n};\ \forall p \in K[x_1, \ldots, x_n];\ (full\_pol\ p) \wedge (m \neq 0) \wedge (p \prec (m : vpol)) \Rightarrow [full\_pol\ (m : p)]$$

Traducción en **Coq**:

```
Theorem Tpol_cpol_full: (m:monom)(p:pol)(full_mon n m)->(full_pol p)->
  ~(z_monom m)->(Tpol_Lex3 p (cpol m vpol))->(full_pol (cpol m p)).
□
```

El resultado siguiente muestra la "transitividad" entre el orden definido sobre polinomios y el definido sobre términos.

Lema 7.2.3.

$$\forall m, m_1, m_2 \in M_{K,n};\ \forall p, q \in K[x_1, \ldots, x_n];\ (m : p) \prec (m_1 : q) \wedge (mon\_term\ m_1) <_L (mon\_term\ m_2) \Rightarrow (mon\_term\ m) <_L (mon\_term\ m_2)$$

Traducción en **Coq**:

```
Theorem Tpol_tran_mon: (m,m1,m2:monom)(p,q:pol)
(Tpol_Lex3 (cpol m p) (cpol m1 q))->(full_mon n m2)->
  (Ttm (mon_term m1) (mon_term m2))->(Ttm (mon_term m) (mon_term m2)).
□
```

La relación $\prec$ es simplificable por la izquierda con la suma de polinomios canónicos.

Lema 7.2.4. *$\forall x, y, z \in K[X]$ se verifica que:*

$$(full\_pol\ x) \wedge (full\_pol\ y) \wedge (full\_pol\ z) \wedge (x +_p y \prec x +_p z) \Rightarrow y \prec z$$

Prueba: Por inducción sobre el polinomio x, utilizando los constructores de la relación, la hipótesis de inducción y aplicando las reglas de reducción mediante la táctica **Simpl**.

Traducción en **Coq**:

```
Lemma s_1_ord_pol: (x,y,z:pol)(full_pol (suma_app x y))->
        (full_pol (suma_app x z))->
□          (Tpol_Lex3 (suma_app x y) (suma_app x z))->(Tpol_Lex3 y z).
```

Un caso particular del lema anterior, que relaciona el orden $\prec$ sobre polinomios canónicos con el orden $<_L$ sobre términos, es el siguiente corolario.

Corolario 7.2.1.

$$\forall\, b, c \in K;\ \forall\, t, t' \in T^n;\ \forall\, p \in K[x_1, \ldots, x_n];$$
$$\{full\_pol\ [x +_p ((b,t) : vpol)]\} \wedge \{full\_pol\ [x +_p ((c,t') : vpol)]\} \wedge$$
$$[x +_p ((b,t) : vpol)] \prec [x +_p ((c,t') : vpol)] \Rightarrow t <_L t'$$

Traducción en **Coq**:

```
Lemma s_t_ord_p: (b,c:K)(t,t':term)(x:pol)
                  (full_pol (suma_app x (cpol (b,t) vpol)))->
 (full_pol (suma_app x (cpol (c,t') vpol)))->
    (Tpol_Lex3 (suma_app x (cpol (b,t) vpol)) (suma_app x (cpol (c,t') vpol)))->
□       (Ttm t t').
```

Para finalizar esta sección damos un lema técnico de la relación $\prec$ que nos ayudará, más adelante, a descomponer un polinomio en suma de dos polinomios ordenados.

Lema 7.2.5.

$$\forall\, b \in K;\ \forall\, t \in T^n;\ \forall\, p, q \in K[x_1, \ldots, x_n];\ [full\_pol(p +_p q)] \wedge$$
$$[\forall t' \in T^n;\ (t' \in p) \Rightarrow (t <_L t')] \wedge [\forall t' \in T^n;\ (t' \in q) \Rightarrow (t' <_L t)] \Rightarrow$$
$$(p +_p q) \prec [p +_p ((b,t) : vpol)]$$

Prueba: Por inducción sobre los polinomios p, q; las submetas obtenidas se resuelven aplicando los constructores de $Tpol\_Lex3$, propiedades de los términos y coeficientes de los polinomios canónicos así como el teorema (6.6.6).

Traducción en **Coq**:

```
Lemma auxiliar2: (b:K)(t:term)(x,x0:pol)(full_pol (suma_app x x0))->
   (full n t)->((t':term)(term_in_pol x t')->(Ttm t t'))->
      ((t':term)(term_in_pol x0 t')->(Ttm t' t))->
□        (Tpol_Lex3 (suma_app x x0) (suma_app x (cpol (b,t) vpol))).
```

7.3. Polinomios canónicos

En la sección 6.6 hemos definido el predicado $full\_pol$ para caracterizar los polinomios ordenados y sin coeficientes nulos. En esta sección, a partir de dicho predicado, damos una formalización del tipo polinomio canónico. Este tipo será particularmente útil para definir un orden bien fundado sobre polinomios y por tanto el tipo sobre el que se sustentará la formalización de la teoría de las bases de Gröbner.

7.3.1. Formalización

A partir del predicado *full_pol* se define el tipo polinomio canónico *pol_full*, que será el tipo utilizado para probar que el orden definido sobre polinomios, restringido a este tipo, es bien fundado (wellfounded).

Definición 7.3.1. *Un polinomio (pol) se dice canónico si verifica el predicado full_pol, es decir está ordenado en orden decreciente de términos y con todos los coefientes nulos.*

Traducción en **Coq**:

```
Definition pol_full:= {p:pol | (full_pol p)}.
```

Se utilizará $K_c[X]$ para denotar el conjunto de los polinomios canónicos.

Ahora restringimos, mediante la construcción **Case**, el orden $Tpol\_Lex3$ formalizado sobre el tipo *pol*, al nuevo tipo *pol_full*. Esta relación se denotara por $\prec_c$.

Traducción en **Coq**:

```
Definition Tpol_full: pol_full-> pol_full-> Prop:=
   [a,b: pol_full]
    (<Prop> Case a of
      [p: pol][H1: (full_pol p)]
       (<Prop> Case b of [q: pol] [H2: (full_pol q)] (Tpol_Lex3 p q) end)
    end).
```

Ya hemos visto anteriormente (pág 30) que en Coq, una manera de formalizar funciones definidas por casos es utilizar **Cases**. Definimos la función que devuelve la lista de términos de un polinomio cualquiera.

Traducción en **Coq**:

```
Fixpoint pol_Lex_rec [p: pol]: (list term):= Cases p of
   vpol              => (nil ?)
  |(cpol (k,t) p1)   => (cons t (pol_Lex_rec p1))
  end.
```

Esta función será utilizada en la mayoría de las definiciones y pruebas relativas al orden bien fundado sobre polinomios canónicos. A este efecto probamos a continuación algunas propiedades.

Para trabajar con listas de términos aprovechamos las definiciones de orden de listas y listas en orden descendente de la biblioteca *Relation_Operators* de **Coq**, así como la definición del orden lexicográfico sobre listas.

```
Inductive Ltl [A : Set; leA : A->A->Prop]: (list A)->(list A)->Prop :=
      Lt_nil : (a:A; x:(list A))(Ltl A leA (nil A) (cons a x))
    | Lt_hd : (a,b:A)(leA a b)->
              (x,y:(list A))(Ltl A leA (cons a x) (cons b y))
    | Lt_tl : (a:A; x,y:(list A))
              (Ltl A leA x y)->(Ltl A leA (cons a x) (cons a y)).


Inductive Desc [A : Set; leA : A->A->Prop]  : (list A)->Prop :=
```

```
    d_nil : (Desc A leA (nil A))
  | d_one : (x:A)(Desc A leA (cons x (nil A)))
  | d_conc : (x,y:A; l:(list A))(leA x y)->(Desc A leA l^(cons y (nil A)))->
               (Desc A leA (l^(cons y (nil A)))^(cons x (nil A))).


Definition Pow [A:Set; leA:(A->A->Prop)][List:=(list A)]: Set :=
               (sig List (Desc A leA)).


Definition lex_exp [A:Set; leA:(A->A->Prop)][List:=(list A)]: Pow->Pow->Prop:=
  [a,b:(Pow A leA)](Ltl A leA (proj1_sig List (Desc A leA) a)
    (proj1_sig List (Desc A leA) b))
```

A partir de estas definiciones comprobamos, como era de esperar, que el orden dado sobre polinomios se conserva al pasar a sus listas de términos. Utilizamos como orden subyacente en las listas, el definido (Ttm) sobre términos. La demostración es inmediata debido a la correspondencia entre los constructores de los predicados $Tpol\_Lex3$ y Ltl.

Traducción en **Coq**:

```
Lemma pol_rec_Tpol_lex3: (p,q:pol)(Tpol_Lex3 p q)->
                           (Ltl ? Ttm (pol_Lex_rec p) (pol_Lex_rec q)).
```

□

De manera análoga a como hemos hecho en polinomios (lista de monomios), con la ayuda de las definiciones precedentes, introducimos la noción de listas de términos en orden descendente de un polinomio dado.

Traducción en **Coq**:

```
Definition Listterm_desc:= [p:pol] (Desc term Ttm (pol_Lex_rec p)).
```

Probamos resultados técnicos, relativos a propiedades de las listas descendentes de términos. Estos resultados sirven para automatizar un poco las pruebas posteriores. La pruebas se realizan mediante los constructores de los tipos utilizados y aplicando propiedades de listas, obtenidas en las bibliotecas del sistema, así como tácticas de reescritura.

Traducción en **Coq**:

```
Lemma Desc_rev: (l:(list term))(t,b:term)(Ttm b t)->
        (Desc term Ttm (cons b l))->(Desc term Ttm (cons t (cons b l))).


Lemma Desc_rev_pol: (p:pol)(t,b:term)(Ttm b t)->
                      (Desc term Ttm (cons b (pol_Lex_rec p)))->
                        (Desc term Ttm (cons t (cons b (pol_Lex_rec p)))).
```

□

Se comprueba que un polinomio que verifica el predicado $full\_pol$ tiene efectivamente sus términos en orden descendente. Se prueba en primer lugar, por inducción sobre el tipo pol, un resultado relativo al predicado low_pol; a partir de él por medio de la simplificación de objetos y aplicando inducción sobre el polinomio dado se obtiene la prueba.

Traducción en **Coq**:

```
Lemma ayuda_full_imp_lis_desc: (p:pol)(t:term)(full_pol p)->(full n t)->
        (low_pol t p)->(Desc term Ttm (cons t (pol_Lex_rec p))).


Theorem full_imp_lis_desc: (p:pol)(full_pol p)->(Listterm_desc p).
```
□

Con la ayuda de las definiciones y propiedades anteriores, es posible definir formalmente la función auxiliar que dado un polinomio canónico, es decir de tipo *pol_full*, obtiene explícitamente la lista de términos en orden descendente.

Traducción en **Coq**:

```
Definition pol_recpol: pol_full->(Pow term Ttm) :=
   [a: pol_full]
    ( <(Pow term Ttm)> Case a of
     [p: pol][H: (full_pol p)]
     (exist (list term) (Desc term Ttm) (pol_Lex_rec p) (full_imp_lis_desc p H))
      end).
```

Similarmente a lo que hemos hecho con la función *pol_Lex_rec* (pág 157), demostramos que el orden dado sobre polinomios, en este caso canónicos, se conserva al pasar al orden lexicográfico de sus listas de términos, explícitamente ya en orden descendente. La demostración, una vez simplificadas las notaciones y tipos correspondientes, se obtiene por aplicación directa del lema *pol_rec_Tpol_lex*3.

Traducción en **Coq**:

```
Lemma pol_rec_Tpol:(p,q: pol_full)(Tpol_full p q)->
                     (lex_exp term Ttm (pol_recpol p)(pol_recpol q)).
```
□

7.3.2. Subtipos

Introducimos dos funciones que formalizan los tipos *pol_full* y (*list pol_full*) como subtipos de *pol* y (*list pol*), respectivamente. Nos permitirán en los siguientes capítulos trasladar resultados probados sobre *pol* a resultados sobre el tipo *pol_full*.

Traducción en **Coq**:

```
Definition inc: pol_full -> pol :=
   [p:pol_full] (Case p of [q:pol][H:(full_pol q)] q
  end).


Fixpoint inc_list [p:(list pol_full)]: (list pol) := Cases p of
   nil              => (nil ?)
  |(cons p1 p2)    => (cons (inc p1) (inc_list p2))
  end.
```

Los lemas siguientes muestran la "equivalencia" entre el tipo *pol* que verifique el predicado *full_pol* y el tipo *pol_full*. Más adelante tendremos necesidad de estos resultados. Las pruebas se obtienen fácilmente a partir del desdoblamiento del tipo polinomio canónico.

Lema 7.3.1.

$$\forall f \in K_c[X];\ \{\exists\, g \in K[X];\ (full\_pol\ g)\ \wedge\ ((inc\ f) =_p q)\}$$

Traducción en **Coq**:

```
Lemma ex_fpol:(f:pol_full){g:pol|(full_pol g)/\(equpol (inc f) g)}.
□
```

Lema 7.3.2.

$$\forall f \in K[X];(full\_pol\ f)\ \Rightarrow\ \{\exists\, g \in K_c[X];\ [full\_pol\ (inc\ f)]\ \wedge\ [(inc\ f) =_p g]\}$$

Traducción en **Coq**:

```
Lemma ex_fpol_eq:(f:pol)(full_pol f)->{g:pol_full|(equpol f (inc g))}.


Lemma pol_f_impl_f_pol:(f:pol_full)(full_pol (inc f)).
□
```

7.3.3. Bien fundado

Estamos ya en condiciones de probar que el orden definido sobre polinomios canónicos verifica la propiedad de buen orden.

Teorema 7.3.1. *El orden definido sobre polinomios canónicos* ($\prec_c$)*, es bien fundado.*

Prueba: Aplicamos resultados de las librerías del sistema **Coq** sobre listas y órdenes (lemas *wf_incl*, *wf_lex_exp* y *wf_inverse_image* contenidos en los ficheros *Wellfounded.Inclusion*, *Wellfounded.Lexicographic_Exponentation* y *Wellfounded.Inverse_Image*, respectivamente). Una vez aplicados los lemas anteriores, la prueba queda reducida a utilizar el lema anterior y a demostrar que el orden *Ttm* sobre términos es bien fundado, teorema (3.4.2).

Traducción en **Coq**:

```
Theorem Tpol_full_wf: (well_founded pol_full Tpol_full).
□
```

Capítulo 8

Ideales de polinomios

Los polinomios juegan un papel central en computación simbólica y muchos de los problemas relacionados con ellos se pueden formular en términos de *ideales de polinomios*. Es más, se usan diversos tipos particulares de bases de ideales para obtener algoritmos eficientes en computación simbólica, y muchas soluciones satisfactorias a tales problemas utilizan un tipo particular de base de ideales de polinomios llamado *base de Gröbner*, ver [30, 32, 55].

8.1. Definición

Definición 8.1.1. *Un subconjunto I de $K[x_1,\ldots,x_n]$ es un* ideal *si satisface:*

- $0 \in I$
- *Si* $f, g \in I \Rightarrow (f +_p g) \in I$
- *Si* $f \in I$ *y* $h \in K[x_1,\ldots,x_n] \Rightarrow (h \cdot_p f) \in I$

El primer ejemplo de un ideal es el ideal generado, en el anillo de polinomios $K[x_1,\ldots,x_n]$, por un número finito de polinomios.

Definición 8.1.2. *Sean $f_1, f_2, \ldots, f_s$ polinomios de $K[x_1,\ldots,x_n]$. Definimos el conjunto*

$$< f_1,\ldots,f_s >:= \left\{ \sum_{i=1}^{s} h_i \cdot_p f_i \;/\; h_1,\ldots,h_s \in K[x_1,\ldots,x_n] \right\}$$

El siguiente predicado define, por comprensión, el conjunto generado por un conjunto (lista) finito de polinomios como se ha expresado en la definición anterior. En efecto, no se trata de calcular $< f_1,\ldots,f_s >$, sino de expresar el hecho de que los elementos de $< f_1,\ldots,f_s >$ son todas las posibles combinaciones lineales obtenidas con los elementos del conjunto de polinomios inicial y coeficientes polinomicos cualesquiera.

Traducción en **Coq**:

```
Inductive Ideal: pol->(list pol)->Prop:=
   ideal_vpol: (p:pol)(F:(list pol))(full_term_pol p)->(equpol p vpol)->
                  (Ideal p F)
 | ideal_cpol: (p,p',q,q':pol)(F:(list pol))(full_term_pol p)->
               (full_term_pol q')->(Ideal p' F)->
                  (equpol p (suma_app (mult_p q q') p'))->(Ideal p (cons q F)).
```

El hecho crucial a demostrar en este capítulo es que el conjunto de polinomios $< f_1, \ldots, f_s >$ es un ideal (es decir verifica la condiciones dadas en la definición 8.1.1) que llamaremos **ideal generado** por $f_1, \ldots, f_s$. Diremos que el conjunto $P = \{f_1, \ldots, f_s\}$ es una **base** para este ideal. Desafortunadamente, aunque P genera el conjunto $< P >$, los polinomios f_i de P no nos dan muchas pistas sobre la naturaleza de este ideal.

Ejemplo 8.1.1. *Los siguientes polinomios,*

$$p_1 = 3x^3yz - xz^2, \quad p_2 = xy^2z - 2yz, \quad p_3 = x^2y - 4z^2$$

generan un ideal de polinomios en $\mathbb{R}[x, y, z]$ *denotado por,*

$$< p_1, p_2, p_3 >= \{h_1 \cdot_p p_1 + h_2 \cdot_p p_2 + h_3 \cdot_p p_3 \ / \ h_1, h_2, h_3 \in \mathbb{R}[x, y, z]\}$$

No es difícil comprobar que $q = -9x^4y^2z + x^4y^2 + 2x^3y - 10xz^2$ *es un miembro de este ideal, pues los polinomios* $h_1 = 2 - 4xy$, $h_2 = 3x^3$, *y* $h_3 = x^2y + 2x$, *verifican que*

$$q = h_1 \cdot_p p_1 + h_2 \cdot_p p_2 + h_3 \cdot_p p_3$$

En este caso, a fuerza de probar podríamos "encontrar" los polinomios h_1, h_2 *y* h_3. *Sin embargo, como veremos en los próximos capítulos, en general no es fácil saber si un polinomio dado* q, *pertenece a un ideal generado por un conjunto* $\{p_1, \ldots, p_n\}$ *de polinomios.*

8.2. Caracterización

Para enunciar y probar propiedades sobre la formalización de *Ideal* necesitamos un predicado que compruebe si un polinomio dado pertenece a una lista (conjunto) de polinomios.

Traducción en **Coq**:

```
Inductive pol_In_ensemb [p:pol]: (list pol)->Prop:=
   pol_In_hd: (p':pol)(F:(list pol))(full_term_pol p')->
      (full_term_pol p)->(equpol p p')->(pol_In_ensemb p (cons p' F))
 | pol_In_tl: (p':pol)(F:(list pol))(pol_In_ensemb p F)->
      (pol_In_ensemb p (cons p' F)).
```

De la definición de *pol_In_ensemb* se obtienen directamente las pruebas de las siguientes propiedades relativas a este predicado: "*si un polinomio* f_0 *pertenece a un conjunto de polinomios* $\{f\}$, *entonces* $f = f_0$, *con la igualdad definida por el sistema* $(=)$" y "*la pertenencia a una lista de polinomios es estable por la relación equpol*".

Traducción en **Coq**:

```
Lemma pol_In_ensemb_unic: (f,f0:pol)(pol_In_ensemb f0 (cons f (nil pol)))->
                          (f=f0).
```

```
Lemma pol_In_ensemb_ext: (x:pol)(F:(list pol))(y:pol)(full_term_pol y)->
       (equpol x y)->(pol_In_ensemb x F)->(pol_In_ensemb y F).
```
□

Retomando la definición de *Ideal*, en primer lugar comprobamos que *la pertenencia a un ideal es estable por la relación equpol.* La prueba se obtiene por **Inversion** sobre la hipótesis $f \in< F >$; esto genera dos submetas que se resuelven aplicando directamente los constructores del predicado *Ideal* y la transitividad de la relación *equpol*.

Traducción en **Coq**:

```
Lemma pol_eq_id_eg: (f,g:pol)(F:(list pol))(Ideal f F)->(equpol f g)->
                    (full_term_pol g)->(Ideal g F).

Intros f g F i d fg.
Inversion i.
Apply ideal_vpol; Trivial.
Apply equpol_tran with f; Auto.
Apply ideal_cpol with p':=p' q':=q'; Trivial.
Apply equpol_tran with f; Auto.
```
□

Proposición 8.2.1. *Dado un conjunto de polinomios F, se verifica:*

$$\forall g \in K[x_1, \ldots, x_n];\ < F > \subset < g \cup F >$$

Traducción en **Coq**:

```
Lemma ideal_mon : (F:(list pol))(p,g:pol)(Ideal p F)->(Ideal p (cons g F)).
```
□

Veamos que el conjunto $< F >$ verifica la tercera condición (conjunto cerrado respecto a la multiplicación de polinomios) de ideal de la definición (8.1.1). Al igual que los resultados anteriores, y casi todos los lemas de este capítulo referentes al predicado *Ideal*, se prueba utilizando la definición recursiva de dicho predicado. Las submetas generadas se obtienen aplicando los constructores de dicho predicado, la transitividad de *equpol*, así como lemas sobre el número de variables de los polinomios resultantes al aplicar las operaciones algebraicas básicas de polinomios.

Proposición 8.2.2. *Dado un conjunto de polinomios F, se verifica:*

$$\forall f \in< F >;\ \forall g \in K[x_1, \ldots, x_n] \Rightarrow f \cdot_p g \in< F >$$

Traducción en **Coq**:

```
Lemma mult_p_id: (f,g:pol)(F:(list pol))(Ideal f F)->(full_term_pol g)->
                 (Ideal (mult_p f g) F).
```
□

Utilizando un lema auxiliar, la proposición (8.2.1) y la estabilidad de la pertenencia a un ideal por *equpol*, podemos probar que los elementos de la **base** están en el ideal.

Proposición 8.2.3. *Dado un conjunto de polinomios F, se verifica:*

$$\forall f \in K[x_1, \ldots, x_n];\ f \in F \Rightarrow f \in < F >$$

Traducción en **Coq**:

```
Lemma p_idp: (p:pol)(F:(list pol))(full_term_pol p)->(Ideal p (cons p F)).


Lemma pol_F_idF: (f:pol)(F:(list pol))(pol_In_ensemb f F)->(full_term_pol f)->
                   (Ideal f F).
```

□

Como la pertenencia a un ideal es estable por la relación *equpol*, de los teoremas (8.2.2 y 6.3.17) se obtienen directamente los siguientes corolarios.

Corolario 8.2.1. *Dado un conjunto de polinomios F, se verifica:*

$$\forall f \in K[x_1, \ldots, x_n];\ \forall m \in M_{K,n};\ f \in F \Rightarrow (f \cdot_M m) \in < F >$$

Traducción en **Coq**:

```
Lemma pol_F_idmult: (f:pol)(m:monom)(F:(list pol))(pol_In_ensemb f F)->
   (full_term_pol f)->(full_mon n m)->(Ideal (mult_m f m) F).
```

□

Corolario 8.2.2. *Dado un conjunto de polinomios F, se verifica:*

$$\forall f \in < F > \Rightarrow -f \in < F >$$

Traducción en **Coq**:

```
Lemma opp_p_id: (f:pol)(F:(list pol))(Ideal f F)->(Ideal (pol_opp f) F).
```

□

Probamos, de forma análoga a la proposición 8.2.2, que el conjunto $< F >$ verifica la segunda condición ideal (definición 8.1.1).

Proposición 8.2.4. *Dado un conjunto de polinomios F, se verifica:*

$$\forall f, g \in < F > \Rightarrow f +_p g \in < F >$$

Traducción en **Coq**:

```
Lemma suma_p_id: (f:pol)(F:(list pol))(Ideal f F)->(g:pol)(Ideal g F)->
                   (Ideal (suma_app f g) F).
```

□

Los lemas técnicos siguientes nos permitirán pruebas más cómodas, cuando se trate de cálculos sobre la pertenencia a un ideal del polinomio opuesto a uno dado.

Traducción en **Coq**:

```
Lemma opp_p_id2: (f,g:pol)(F:(list pol))(Ideal f F)->(equpol g (pol_opp f))->
```

```
Lemma oppm_idp:(p:pol)(m:monom)(F:(list pol))(full_term_pol p)->(full_mon n m)->
                  (Ideal (pol_opp (mult_m p m)) (cons p F)).
```

```
Lemma pol_F_idmult_opp: (f:pol)(m:monom)(F:(list pol))(pol_In_ensemb f F)->
            (full_term_pol f)->(full_mon n m)->(Ideal (pol_opp (mult_m f m)) F).
```

□

Cuando se prueben, en el capítulo siguiente, propiedades que relacionan la reducción de polinomios con la pertenencia a un ideal, se necesitará lo siguiente.

Proposición 8.2.5. *Dado un conjunto finito de polinomios en varias variables* F, *y* $\forall f, g_0, h_0 \in K[x_1, \ldots, x_n]$, *se verifica:*

$$(h_0 \in < f \cup F >) \wedge (f \in < g_0 \cup F >) \Rightarrow (h_0 \in < g_0 \cup F >)$$

Prueba: Explicitando, mediante **Inversion**, las hipótesis $h_0 \in < f \cup F >$, $f \in < g_0 \cup F >$ y utilizando la proposición 8.2.1, así como propiedades de las operaciones algebraicas de polinomios.

Traducción en **Coq**:

```
Lemma id_trans: (F:(list pol))(f,g0,h0:pol)(Ideal h0 (cons f F))->
                      (Ideal f (cons g0 F))->(Ideal h0 (cons g0 F)).
```

□

Una forma de caracterizar ideales es definir la función *Ideal_ens* que es la función característica de los ideales generados por un conjunto de polinomios. A partir de ella, dos ideales serán iguales si coinciden sus funciones características, lo que se formaliza en Coq por el predicado *eq_Ideal*.

Traducción en **Coq**

```
Definition Ideal_ens:= [F:(list pol)][p:pol](Ideal p F).
```

```
Definition eq_Ideal:= [I1,I2:(pol->Prop)]
       ((p:pol)(I1 p)->(I2 p)) /\ ((p:pol)(I2 p)->(I1 p)).
```

Una propiedad fundamental de ideales, que necesitaremos en otros capítulos, es que si $p \in < F >$, entonces $< p \cup F > = < F >$, es decir que si un elemento del ideal se agrega a la base, el ideal no varía. La prueba se obtiene explicitando las definiciones anteriores y aplicando resultados probados en este capítulo.

Traducción en **Coq**

```
Lemma eq_Ideal_eq_p: (p:pol)(F:(list pol))(Ideal p F)->
        (eq_Ideal (Ideal_ens F) (Ideal_ens (cons p F))).
```

□

Capítulo 9

Reducción (División) de polinomios

9.1. Introducción

El algoritmo de la división para polinomios en una variable asegura que si f y g son polinomios, con $f \neq 0$, existen dos polinomios q y r tal que $g = qf + r$ donde ó $r = 0$ ó $gr(r) < gr(f)$. Esto significa que podemos representar cada polinomio $g \in K[x]$ por un polinomio r con $gr(r) < gr(f)$ ó por 0, módulo el ideal[1] $< f >$. Esta representación es única, ya que si $g = q_1 f + r_1 = q_2 f + r_2$, entonces $r_1 - r_2$ es un múltiplo de f, y esto sólo es posible, si $r_1 - r_2 = 0$. Por consiguiente tenemos una única representación de polinomios módulo el ideal $< f >$. Es la unicidad de representación de polinomios la que nos permite realizar cálculos módulo $< f >$, es decir, en el anillo cociente $K[x]/ < f >$.

Si queremos extender esto al anillo de polinomios de varias variables, nos encontramos con varios obstáculos. El primero es que mientras que en $K[x]$ sólo hay un orden de términos compatible con la multiplicación de términos, a saber $1 < x < x^2 < \ldots$, en $K[x_1, \ldots, x_n]$ hay varias posibilidades. Este obstáculo ya lo hemos salvado introduciendo, en el capítulo 3, el concepto de "orden admisible", y eligiendo para nuestro desarrollo el orden lexicográfico. Un segundo obstáculo es que los ideales en $K[x_1, \ldots, x_n]$ no son todos principales[2]. Esto trae consigo problemas para una buena elección del conjunto generador de un ideal, lo que nos llevará al concepto de bases de Gröbner.

[1] Puesto que $g - r = qf \in < f >$, diremos que g y r están relacionados módulo el ideal $< f >$.

[2] Un ideal I engendrado por un sólo elemento r del anillo, se le llama ideal principal. En el anillo $K[x]$, de una sola variable, todos sus ideales son principales.

9.2. Algoritmo de la división

La meta de esta sección es generalizar el algoritmo de la división de polinomios en una variable a polinomios en varias (n) variables, también llamado proceso de reducción. La principal diferencia es que utilizamos un algoritmo que divide un polinomio por un conjunto de polinomios; este algoritmo es esencial para la obtención de la base de Gröbner de un conjunto de polinomios.

9.2.1. Nociones básicas

Una vez fijado un orden admisible $<_L$, sobre los términos de un polinomio, introducimos en primer lugar la noción de reducción de un polinomio g por un polinomio dado f.

Definición 9.2.1. *Dados $f, g, h \in K[x_1, \ldots, x_n]$ con $f \neq_p vpol$, decimos que* **g se reduce a h módulo f** *en un sólo paso (denotado $g \xrightarrow{f} h$) si, y sólo si, existe un término $t \in g$ con $(hterm\ f)|t$, y*

$$h =_p g -_p f \cdot_M \left(\frac{(coef\ t\ g)}{hcoef\ f}, \frac{t}{hterm\ f}\right)$$

Traducción en **Coq**:

```
Inductive red [f,g,h:pol]: Prop:=
   red1_simpl: (t:term)(term_in_pol g t)->(full_term_pol g)->
      (full_term_pol f)->(full_term_pol h)->~(equpol f vpol)->
         (term_div (hterm f) t)->
(equpol h (suma_app g (pol_opp (mult_m f ((divK (coef g t) (hcoef f)),
                                  (div_term t (hterm f)))))))->(red f g h).
```

Ejemplo 9.2.1. *Sea $<_L$ el orden lexicográfico definido por $(y <_L x)$, y los polinomios $g =_p 6x^3y + 4x^2y + 4y^3 - 1$ y $f =_p 2xy + y^3$. Entonces,*

$$6x^3y + 4x^2y + 4y^3 - 1 \xrightarrow{2xy+y^3} h := g - f.\left(\frac{4x^2y}{2xy}\right) = 6x^3y - 2xy^3 + 4y^3 - 1$$

Aquí el término de g elegido para la reducción es $4x^2y$.

Esta reducción es no determinista, pues si el término elegido es $6x^3y$, obtendríamos

$$6x^3y + 4x^2y + 4y^3 - 1 \xrightarrow{2xy+y^3} h := g - f.\left(\frac{6x^3y}{2xy}\right) = -3x^2y^3 + 4x^2y + 4y^3 - 1$$

Es decir para las mismas entradas, polinomios f y g, se pueden obtener distintos polinomios de salida.

En la definición anterior, el polinomio resultante h se puede pensar como el resto de la división de g por f en un paso, de manera similar a como se realiza en polinomios de una variable, es decir, g y h son equivalentes módulo f.

Otra caracterización de la reducción, muy útil para la simplificación de las pruebas cuando se tienen varias opciones (términos) de reducción, es formalizar la reducción explicitando el término que se simplifica.

Definición 9.2.2. *Dados $f, g, h \in K[x_1, \ldots, x_n]$ y $t \in g$ con $f \neq_p vpol$, decimos que $g \xrightarrow{f;\, t} h$,* **g se reduce a h módulo f por el término t** *en un sólo paso, si, y sólo si, existe un término $u \in T^n$ con $t = u.(hterm\ f)$, $c =_K \frac{(coef\ t\ g)}{hcoef\ f}$, y*

$$h =_p g -_p f \cdot_M (c, u)$$

Traducción en **Coq**:

```
Inductive red_exp [f,g,h:pol;t:term]: Prop:=
  redi_exp_simpl: (term_in_pol g t)->(full_term_pol g)->(full_term_pol f)->
    (full_term_pol h)->~(equpol f vpol)->(u:term)(full n u)->
     t=(term_mult u (hterm f))->(c:K)(eqK c (divK (coef g t) (hcoef f)))->
      (equpol h (suma_app g (pol_opp (mult_m f (c,u)))))->(red_exp f g h t).
```

Ejemplo 9.2.2. *Sea $<_L$ el orden lexicográfico definido por $(y <_L x)$, dados los polinomios $g =_p 6x^3 + 4x^2y + 10xy + 4y^3 - 1$ y $f =_p 2xy + y^3$. Entonces, la reducción por el término xy sería:*

$$6x^3 + 4x^2y + 10xy + 4y^3 - 1 \xrightarrow{2xy+y^3;\, xy} g - f.\left(\frac{10xy}{2xy}\right) = 6x^3 + 4x^2y - y^3 - 1$$

Pero si reducimos por el término x^2y, obtenemos

$$6x^3 + 4x^2y + 10xy + 4y^3 - 1 \xrightarrow{2xy+y^3;\, x^2y} 6x^3 - 2xy^3 + 10xy + 4y^3 - 1$$

La idea básica del algoritmo de la reducción (división) en polinomios de varias variables es la misma que en los polinomios de una variable: cuando hacemos la reducción $g \xrightarrow{f;\, t} h$, queremos cancelar monomios de g utilizando el término principal de f. En el ejemplo anterior cuando reducimos el polinomio $6x^3 + 4x^2y + 10xy + 4y^3 - 1$ por el polinomio $2xy + y^3$ utilizando el término x^2y, se ve que eliminamos de $6x^3 + 4x^2y + 10xy + 4y^3 - 1$ el monomio $(4x^2y)$ y lo sustituimos por monomios con términos estrictamente menores que x^2y. La cancelación del término por el cual se reduce es, precisamente, lo que se prueba en la siguiente proposición.

Proposición 9.2.1. *Dados $f, g, h \in K[x_1, \ldots, x_n]$ y $t \in T^n$, se verifica que*

$$(g \xrightarrow{f;\, t} h) \Rightarrow t \notin h$$

Prueba: La prueba se obtiene mediante **Inversion** de la hipótesis del contexto $H : (red\_exp\ f\ g\ h\ t)$, la definición (9.2.2) y aplicando propiedades relativas a términos y coeficientes de un polinomio, probados en capítulos anteriores.

Traducción en **Coq**:

```
Lemma elim_red_exp: (f,g,h:pol)(t:term)(red_exp f g h t)->
                        (no_term_in_pol h t).
```

□

Presentamos algunos resultados que ilustran claramente sobre las relaciones entre las dos definiciones anteriores. En posteriores especificaciones se utilizará la más adaptada a nuestro desarrollo.

Lema 9.2.1. *Dados* $g, h, f_i \in K[x_1, \ldots, x_n]$ *y* $t \in T^n$, *que verifican* $g \xrightarrow{f_i;\, t} h$, *entonces*

$$\exists q \in K[x_1, \ldots, x_n] \text{ tal que } [(\forall t' \in q) \Rightarrow t <_L t'] \wedge (g -_p q) \xrightarrow{f_i} (h -_p q)$$

Prueba: De la hipótesis $g \xrightarrow{f_i;\, t} h$ se obtiene que $h =_p g -_p f_i \cdot_M (c, u)$, donde $u = \frac{t}{(hterm\ f_i)}$ y $c =_K \frac{(coef\ t\ g)}{(hcoef\ f_i)}$. Como $t \in g$, por el corolario (6.5.1), descomponemos g en la forma $g =_p H(g,t) +_p [(\text{coef}\ (t,g)), t] +_p L(g,t)$. Tomando $q =_p H(g,t)$, por el teorema (6.4.11) ($t \notin q$), con lo cual $t \in (g -_p q)$, utilizando el lema (6.4.3).

Sólo queda demostrar que $[h -_p H(g,t)] =_p [g -_p H(g,t)] -_p f_i \cdot_M (c, u)$, lo cual es consecuencia de las propiedades de las operaciones algebraicas sobre polinomios enunciadas en capítulos anteriores.

Traducción en **Coq**:

```
Lemma red_comp: (g,h,fi:pol)(t:term)(red_exp fi g h t)->
   (EX x | (full_term_pol x)/\
      ((t':term)(term_in_pol x t')->(Ttm t t'))/\
         (red fi (suma_app g (pol_opp x)) (suma_app h (pol_opp x)))).
```

Aunque las definiciones (9.2.1 y 9.2.2) son análogas, comprobamos formalmente su "equivalencia técnica", dado que en la segunda se explicita el término utilizado en la reducción. La prueba se hace por **Inversión** sobre ambas definiciones y, en cada caso, aplicando los respectivos constructores.

```
Lemma red_impl_red_exp: (f,g,h:pol)(red f g h)->
                          (Ex [t:term](red_exp f g h t)).


Lemma red_exp_impl_red: (f,g,h:pol)(t:term)(red_exp f g h t)->(red f g h).
```

□

Un caso particular de la definición 9.2.2, es la reducción de polinomios por el término principal.

Definición 9.2.3. *Dados* $f, g, h \in K[x_1, \ldots, x_n]$ *con* $f, g \neq_p vpol$, *decimos que* **g se reduce a h módulo f por el término principal** *en un sólo paso si, y sólo si,* $(hterm\ f) | (hterm\ g)$, *y*

$$h =_p g -_p f \cdot_M \left(\frac{hcoef\ g}{hcoef\ f}, \frac{hterm\ g}{hterm\ f} \right)$$

Este tipo de reducción se denota por $g \xrightarrow{f;\, hred} h$

Traducción en **Coq**:

```
Inductive hred [f,g,h:pol]: Prop:=
   hred1_simpl: (full_term_pol g)->(full_term_pol h)->(full_term_pol f)->
        ~(equpol g vpol)->~(equpol f vpol)->(term_div (hterm f) (hterm g))->
           (equpol h (suma_app g (pol_opp (mult_m f
           ((divK (hcoef g) (hcoef f)),(div_term (hterm g) (hterm f)))))))->
                              (hred f g h).
```

Ejemplo 9.2.3. *Sean* $<_L$ *el orden lexicográfico definido por* $(y <_L x)$, $g =_p 6x^2y + 4xy - x + 4y^3 - 1$ *y* $f =_p 2xy + y^3$. *Aquí,*

$$g \xrightarrow{f;\, hred} -3xy^3 + 4xy - x + 4y^3 - 1$$

Comprobamos que, efectivamente, *la reducción por el término principal es un caso particular de la reducción.* La prueba es análoga a las anteriores.

```
Lemma red_hred: (f,g,f1:pol)(hred f1 f g)->(red f1 f g).
```
□

Directamente de la formalización de la reducción se obtienen los dos resultados siguientes.

Proposición 9.2.2. *Si un polinomio* g *es reducible por un polinomio* f, *entonces* g *es distinto de vpol.*

$$g \xrightarrow{f} h \;\Rightarrow\; g \neq_p vpol$$

Traducción en **Coq**:

```
Lemma red_n_vpol1: (f,g,h:pol)(red f g h)->~(equpol g vpol).
```
□

Proposición 9.2.3.

$$\forall f \in K[x_1, \ldots, x_n];\; (f \neq_p vpol) \;\Rightarrow\; f \xrightarrow{f} vpol$$

Traducción en **Coq**:

```
Lemma red_f_f: (f:pol)(full_term_pol f)->~(equpol f vpol)->(red f f vpol).
```
□

Antes de extender la reducción a un conjunto de polinomios, demostramos un resultado auxiliar muy importante para nuestro trabajo posterior.

Proposición 9.2.4. *Dados* $f, p \in K[x_1, \ldots, x_n]$, *o bien* f *se reduce por un múltiplo del termino principal de* p $(t = u.(hterm\ p))$, *o bien no se reduce módulo* p.

Prueba: Utilizando la decidibilidad de la presencia de un término en un polinomio tenemos dos casos:

- Si $t \notin f$ entonces $coef(t, f) =_K 0$, es decir el polinomio f no puede reducirse utilizando el término t.
- En el caso $t \in f$, tenemos todas las hipótesis necesarias para la formalización de la reducción (red), utilizando el término t.

Traducción en **Coq**:

```
Lemma red_0_un_step: (f,p:pol)(t,u:term)(full_term_pol f)->
(full_term_pol p)->(full n t)->(full n u)->
(~(equpol p vpol))->t=(term_mult u (hterm p))->
(equpol f (suma_app f (pol_opp (mult_m p ((divK (coef f t) (hcoef p)),u))))))\/
```

```
  (red p f (suma_app f (pol_opp (mult_m p((divK (coef f t) (hcoef p)),u))))).

Intros f p t u.
Elim (dec_term_in_pol f t); Intros.
Right.
Split with t; Trivial.
Apply full_term_suma_app; Auto.
Rewrite H4; Auto.
Apply equpol_ss; Trivial.
Apply opp_comp.
Apply mult_pm_comp_2.
Simpl.
Right.
Split; Auto.
Rewrite H4.
Symmetry.
Replace (term_mult u (hterm p)) with (term_mult (hterm p) u); Auto.
Apply div_mult_mon_inv with n; Auto.
Left.
Apply equpol_tran with (suma_app f vpol).
Auto.
Apply equpol_ss; Trivial.
Apply equpol_tran with (pol_opp vpol); Trivial.
Apply opp_comp.
Apply equpol_sym.
Apply mult_m_z_monom; Auto.
□
```

Probamos resultados de extensionalidad de la reducción respecto a la relación *equpol*. Las pruebas se realizan mediante el constructor *red1_simpl*, y aplicando en cada caso propiedades de los elementos que forman un polinomio, formuladas en capítulos anteriores.

Traducción en **Coq**:

```
Lemma red_ext1: (f,g,g',h:pol)(equpol g g')->(full_term_pol g')->
                  (red h f g)->(red h f g').


Lemma red_ext2: (f,g,f',h:pol)(equpol f f')->(full_term_pol f')->
                  (red h f g)->(red h f' g).


Lemma red_ext3: (f,g:pol)(h,h':pol)(equpol h h')->(full_term_pol h')->
                  (red h f g)->(red h' f g).
□
```

9.2.2. Reducción por un conjunto de polinomios

La siguiente definición da el nombre *for_all_list_pol* al predicado mínimo que determina las listas de polinomios que verifican una propiedad P.

Traducción en **Coq**:

```
Inductive for_all_list_pol [P:pol->Prop]: (list pol)->Prop :=
  For_all_nil: (for_all_list_pol P (nil ?))
| For_all_cs: (F:(list pol))(p:pol)(P p)
              ->(for_all_list_pol P F)->(for_all_list_pol P (cons p F)).
```

Las dos cláusulas, *For_all_nil* y *For_all_cs*, no son más que la traducción en el lenguaje **Coq** de los axiomas matemáticos:

$$\begin{cases} \text{la lista vacía (nil)} & \text{verifica P} \\ \forall F \in (list\ pol);\ \forall p \in K[X] & \text{si } p \text{ verifica } P \text{ y } F \text{ verifica } P \Rightarrow \\ & \{p\} \cup F \text{ verifica } P \end{cases}$$

La condición de minimalidad se expresa en el sistema por el *principio de inducción*. El principio asociado a *for_all_list_pol* es:

```
(P:(pol->Prop); P0:((list pol)->Prop))
        (P0 (nil pol))
        ->((F:(list pol); p:pol)
            (P p)->(for_all_list_pol P F)->(P0 F)->(P0 (cons p F)))
        ->(l:(list pol))(for_all_list_pol P l)->(P0 l)
```

Nota: A cada definición inductiva el sistema Coq asocia un principio análogo al anterior.

El predicado *for_all_list_pol* nos permite definir, como hemos dicho anteriormente, los conjuntos finitos de polinomios no nulos con n variables.

Traducción en **Coq**:

```
Definition full_fam := (for_all_list_pol [p:pol]
                          (full_term_pol p)/\~(equpol p vpol)).
```

Los resultados siguientes comprueban que el predicado *full_fam* es la función característica de los polinomios no nulos con n variables. Las pruebas son inmediatas. Se obtienen, en cada caso, por inducción sobre el parámetro de la meta en curso y la técnica de inversion sobre el predicado *full_fam*.

Traducción en **Coq**:

```
Lemma cons_full_fam: (q:pol)(F:(list pol))(full_fam (cons q F))->
                        (full_fam F).


Lemma nvpol_full_fam: (q:pol)(F:(list pol))(full_fam (cons q F))->
                        (~(equpol q vpol)).


Lemma full_term_full_fam: (q:pol)(F:(list pol))(full_fam (cons q F))->
                            (full_term_pol q).
```

□

Estamos en condiciones de demostrar una propiedad importante del predicado *full_fam*, su decidibilidad respecto a un conjunto de polinomios. Hacemos esta prueba por recurrencia sobre el conjunto de polinomios F. El caso (*nil pol*) lo deduce automáticamente el sistema a partir de la implementación de *full_fam*. Si F toma el valor (*cons a l*); entonces la hipótesis de recurrencia sobre l (lista de polinomios), los lemas de decidibilidad *full_pol_tm_dec* y

vpol_dec_full_term (páginas 87 y 131, respectivamente) así como los resultados anteriores permiten concluir la prueba.

Traducción en **Coq**:

```
Lemma full_fam_dec : (F:(list pol)){(full_fam F)}+{~(full_fam F)}.

Induction F; Intros.
Left; Auto.
Elim H; Intros; Clear H.
Elim (full_pol_tm_dec a); Intros.
Elim (vpol_dec_full_term a); Auto; Intros.
Right.
Red; Intros.
Cut ~(equpol a vpol); Intros; Auto.
Apply nvpol_full_fam with l; Auto.
Right.
Red; Intros.
Cut (full_term_pol a); Auto; Intros.
Apply full_term_full_fam with l; Auto.
Right.
Red; Intros.
Apply b.
Apply cons_full_fam with a; Auto.
□
```

Ahora damos algunos resultados técnicos simples, concernientes a la relación que hay entre la definición anterior y el predicado *pol_In_ensemb* implementado en la sección 8.2.

Lema 9.2.2. *Dada un conjunto de polinomios F que verifica el predicado full_fam, se cumple:*

- $\forall p \in F \Rightarrow p \in K[x_1, \ldots, x_n]$
- $\forall p \in F \Rightarrow p \neq_p vpol$
- $\forall p \in F \Rightarrow (hterm\ p) \in T^n$

Traducción en **Coq**:

```
Lemma elem_full: (p:pol)(F:(list pol))(pol_In_ensemb p F)->(full_fam F)->
                        (full_term_pol p).

Lemma elem_n_vpol: (p:pol)(F:(list pol))(pol_In_ensemb p F)->(full_fam F)->
                      ~(equpol p vpol).

Lemma full_pol_In_hterm: (F:(list pol))(full_fam F)->
□       (p:pol)(pol_In_ensemb p F)->(full n (hterm p)).
```

Con la ayuda del predicado *ful_fam*, podemos extender el proceso de reducción, definición (9.2.1), a un conjunto (lista) de polinomios no nulos de n variables.

Definición 9.2.4. *Sean $g, h \in K[x_1, \ldots, x_n]$, y F un conjunto de polinomios no nulos de n variables. Decimos que* **g se reduce a h módulo F**, *denotado por*

$$g \xrightarrow{F} h$$

si, y sólo sí, existe $f \in F$ *tal que*

$$g \xrightarrow{f} h$$

Traducción en **Coq**:

```
Inductive Red1 [F:(list pol);g,h:pol]: Prop:=
  Red1_simp: (full_fam F)->(f:pol)(pol_In_ensemb f F)->
             (red f g h)->(Red1 F g h).
```

Ejemplo 9.2.4. *Consideremos el conjunto* $F := \{f_1, f_2, f_3\}$, *donde*

$$f_1 := 3x^2y + 9x^2 + 2xy + 5x + y - 3 \tag{9.1}$$

$$f_2 := 2x^3y + 6x^3 - 2x^2 - xy - 3x - y + 3 \tag{9.2}$$

$$f_3 := x^3y + 3x^3 + x^2y + 2x^2 \tag{9.3}$$

Los polinomios f_1, f_2, f_3 *están ordenados respecto al orden lexicográfico* ($y <_L x$). *Sea*

$$g := 2x^2y + 8x^2 + \frac{5}{2}xy + \frac{3}{2}x + 5y^2 + \frac{3}{2}y - \frac{9}{2} \tag{9.4}$$

Ocurre que $g \xrightarrow{F} h$,

$$\textit{Para} \quad h := 2x^2 + \frac{7}{6}xy - \frac{11}{6}x + 5y^2 + \frac{5}{6}y - \frac{5}{2} \tag{9.5}$$

ya que,

$$h = g - f_1 . \left(\frac{2x^2y}{3x^2y}\right)$$

Podemos decir, grosso modo, que un polinomio g se reduce al polinomio h módulo F si, y sólo si, h resulta de eliminar un monomio de g mediante la sustracción de un múltiplo de un polinomio f del conjunto de polinomios F; el polinomio así obtenido resulta (en cierto sentido) "menor".

9.2.3. Proceso de reducción

De forma similar a como se hace en polinomios de una variable se puede pensar en el polinomio h, del ejemplo anterior, como el resto de la división en una etapa. Podemos continuar el proceso sustrayendo de g todos los términos que son divisibles por los términos principales de los polinomios de F.

Definición 9.2.5. *Sean* $g, h \in K[x_1, \ldots, x_n]$, *y* $F = \{f_1, \ldots, f_s\}$ *un conjunto de polinomios no nulos de n variables. Decimos que* **g se reduce a h módulo F en varias etapas**, *denotado por*

$$g \xrightarrow[*]{F} h$$

si, y sólo sí, existe una secuencia de índices $i_1, i_2, \ldots, i_t \in \{1, \ldots, s\}$ *y una secuencia de polinomios* $h_1, \ldots, h_{t-1} \in K[x_1, \ldots, x_n]$ *tal que*

$$g \xrightarrow{f_{i_1}} h_1 \xrightarrow{f_{i_2}} h_2 \xrightarrow{f_{i_3}} \ldots \xrightarrow{f_{i_{t-1}}} h_{t-1} \xrightarrow{f_{i_t}} h$$

Nota: En adelante siempre se utilizará esta reducción y la llamaremos simplemente, reducción módulo un conjunto de polinomios

Así vemos que la reducción implementada anteriormente (*Red*1) puede pensarse como un paso (etapa) en la división generalizada. Por ello definimos la clausura reflexiva y transitiva de $\xrightarrow{F}$, denotada por $\xrightarrow[*]{F}$, formalizada en **Coq** por el predicado *Red*3.

Traducción en **Coq**:

```
Inductive Red3 [F:(list pol)]: pol->pol->Prop:=
 Red3_eq: (g,h:pol)(full_fam F)->(full_term_pol g)->(full_term_pol h)->
            (equpol g h)->(Red3 F g h)
|Red3_step: (g,h:pol)(full_fam F)->(full_term_pol g)->(full_term_pol h)->
              (Red1 F g h)->(Red3 F g h)
|Red3_trans: (g,f,h:pol)(full_fam F)->(full_term_pol f)->(full_term_pol g)->
               (full_term_pol h)->(Red3 F g f)->(Red3 F f h)->(Red3 F g h).
```

Ejemplo 9.2.5. *Consideremos el conjunto* $F := \{f_1, f_2\}$, *donde*

$$f_1 := xy + 1 \tag{9.6}$$

$$f_2 := y + 1 \tag{9.7}$$

Los polinomios f_1, f_2 *están ordenados respecto al orden lexicográfico* ($y <_L x$). *Sea*

$$g := xy^2 + 1 \tag{9.8}$$

Ocurre que $g \xrightarrow[*]{F} 2$, *ya que,*

$$xy^2 + 1 \xrightarrow{f_1} -y + 1 \xrightarrow{f_2} 2$$

y por tanto, tenemos que $g = y \cdot_p f_1 + (-1) \cdot_p f_2 + 2$, *donde* $h = 2$ *sería el resto de la división del polinomio* g *por el conjunto de polinomios* F, *es decir,* $g - 2 \in < f_1, f_2 >$.

Algunos lemas técnicos, fácilmente demostrables por **Inversion**, que se utilizarán sistematicamente en pruebas posteriores, son los siguientes.

Traducción en **Coq**:

```
Lemma Red3_ste: (f,g:pol)(F:(list pol))(Red1 F f g)->(Red3 F f g).
```

```
Lemma Red3_full_l: (f,g:pol)(F:(list pol))(Red3 F f g)->(full_term_pol f).
```

```
Lemma Red3_full_r: (f,g:pol)(F:(list pol))(Red3 F f g)->(full_term_pol g).

Lemma Red3_full_fam: (f,g:pol)(F:(list pol))(Red3 F f g)->(full_fam F).

Lemma Red3_tran: (f,g,h:pol)(F:(list pol))(Red3 F f g)->
□                  (Red3 F g h)->(Red3 F f h).
```

A partir de los lemas de extensionalidad de la reducción por un polinomio $\xrightarrow{f}$ (pág 172), se obtiene fácilmente la extensionalidad respecto a *equpol* de las nuevas reducciones: $\xrightarrow{F}$ y $\xrightarrow[*]{F}$.

Traducción en **Coq**:

```
Lemma Red1_ext1: (F:(list pol))(f,g,g':pol)(full_term_pol g')->
                   (equpol g g')->(Red1 F f g)->(Red1 F f g').

Lemma Red1_ext2: (F:(list pol))(f,g,f':pol)(full_term_pol f')->
                   (equpol f f')->(Red1 F f g)->(Red1 F f' g).

Lemma Red3_ext1: (F:(list pol))(f,g:pol)(Red3 F f g)->
   (g':pol)(equpol g g')->   (full_term_pol g')->(Red3 F f g').

Lemma Red3_ext2: (F:(list pol))(f,g:pol)(Red3 F f g)->
□  (f':pol)(equpol f f')->(full_term_pol f')->(Red3 F f' g).
```

Con la ayuda de las definiciones precedentes, es posible demostrar algunas de las propiedades esperadas.

Proposición 9.2.5. *Dados $f, g, h \in K[x_1, \ldots, x_n]$, con g no nulo, y F un conjunto de polinomios no nulos de n variables, se verifica*

$$f \xrightarrow[*]{F} h \quad \Rightarrow \quad f \xrightarrow[*]{g:F} h$$

Es decir, si un polinomio se reduce por un conjunto de polinomios F, también se reduce por cualquier conjunto de polinomios no nulos que contenga a F.

Prueba: Aplicando la recursión estructural sobre $f \xrightarrow[*]{F} h$ obtenemos tres metas que se resuelven utilizando los constructores de *Red3* y las propiedades del predicado *pol_In_ensemb*. En todos los casos se utiliza como polinomio de $(g : F)$ por el cual se reduce, el mismo que se usa en F, obtenido por la hipótesis de recursión.

Traducción en **Coq**:

```
Lemma red3_fFh2: (f,h,g:pol)(F:(list pol))(Red3 F f h)->~(equpol g vpol)->
□                  (full_term_pol g)->(Red3 (cons g F) f h).
```

En la sección (9.2.1), se había demostrado que cualquier polinomio se reduce por sí mismo a *vpol*. De forma similar la siguiente propiedad afirma lo mismo para *Red3*, en el caso de que se agrege al conjunto por el cual se reduce algún polinomio obtenido en el proceso de reducción.

Proposición 9.2.6. *Dados $f, h \in K[x_1, \ldots, x_n]$, con h no nulo, y F un conjunto de polinomios no nulos de n variables, se verifica*

$$f \xrightarrow[*]{F} h \quad \Rightarrow \quad f \xrightarrow[*]{h:F} vpol$$

Prueba: Para simplificar la prueba probamos en primer lugar el resultado tomando como hipótesis la reducción en un solo paso $\xrightarrow{F}$. Por inversión de la hipótesis $f \xrightarrow{F} h$ obtenemos que $f \xrightarrow{f_0} h$ con $f_0 \in F$, y la prueba por transitividad de la reducción respecto al polinomio h, se sigue sin más que utilizar la proposión anterior y la proposión (9.2.3).

La prueba para $\xrightarrow[*]{F}$ se obtiene de la prueba de la reducción en un solo paso de los constructores de *Red3* y de la proposición anterior.

Traducción en **Coq**:

```
Lemma red3_fFh: (f,h:pol)(F:(list pol))
            (Red1 F f h)->~(equpol h vpol)->(Red3 (cons h F) f vpol).


Lemma red3_fFh_vpol: (f,h:pol)(F:(list pol))
            (Red3 F f h)->~(equpol h vpol)->(Red3 (cons h F) f vpol).
```

□

Proposición 9.2.7. *Dados $f, g \in K[x_1, \ldots, x_n]$, y F un conjunto de polinomios no nulos de n variables, se verifica*

$$\forall c \in K,\ \forall t \in T^n; \quad (f \xrightarrow[*]{F} g) \Rightarrow [f \cdot_M (c,t) \xrightarrow[*]{F} g \cdot_M (c,t)]$$

Es decir, $\xrightarrow[*]{F}$ *es compatible con la multiplicación por un monomio.*

Prueba: Al igual que hemos hecho en la prueba de la proposición anterior, en primer lugar probamos el resultado para la reducción en un solo paso $\xrightarrow{F}$. Comenzamos traduciendo la hipótesis $(Red1\ F\ f\ g)$ y obtenemos la hipótesis $H2 : (red\ f0\ f\ g)$, de la cual se tiene $g =_p f - f_0 \cdot_M m'$. Otra de las hipótesis obtenidas es que el monomio principal de $f_0 \cdot_M m'$, denotado $M(f_0 \cdot_M m')$, está en f. Se sigue fácilmente que $M(f_0 \cdot_M [m'.(c,t)])$ está en $f \cdot_M (c,t)$ y así, se verifica que

$$f \cdot_M (c,t) \xrightarrow{f_0} f \cdot_M (c,t) - f_0 \cdot_M [m'.(c,t)] =_p g \cdot_M (c,t)$$

En la prueba con la hipótesis $\xrightarrow[*]{F}$, distinguimos dos casos, según sea $c =_K 0$ ó no. El primer caso se resuelve trivialmente por el primer constructor de *Red3* y el segundo se sigue de la prueba de la reducción en un solo paso y de los constructores de *Red3*.

Traducción en **Coq**:

```
Lemma Red1_mult_m: (f,g:pol)(F:(list pol))(Red1 F f g)->(c:K)(~(eqK c OK))->
        (t:term)(full n t)->(Red1 F (mult_m f (c,t)) (mult_m g (c,t))).


Lemma Red3_mult_m: (f,g:pol)(F:(list pol))(Red3 F f g)->
   (c:K)(t:term)(full n t)->(Red3 F (mult_m f (c,t)) (mult_m g (c,t))).
□
```

Un caso particular, que se obtiene aplicando la extensionalidad respecto a *equpol*, de la proposición anterior es el corolario siguiente.

Corolario 9.2.1. *Dados $g \in K[x_1,\ldots,x_n]$, y F un conjunto de polinomios no nulos de n variables, se verifica*

$$\forall c \in K,\ \forall t \in T^n;\quad (g \xrightarrow[*]{F} vpol) \Rightarrow (g \cdot_M (c,t) \xrightarrow[*]{F} vpol)$$

Traducción en **Coq**:

```
Lemma Red3_vpol_mult_m: (g:pol)(F:(list pol))(Red3 F g vpol)->
          (c:K)(t:term)(full n t)->(Red3 F (mult_m g (c,t)) vpol).
□
```

Una condición suficiente, utilizada en pruebas posteriores, para la reducción en una sola etapa es la siguiente.

Teorema 9.2.1. *Dados $f, g \in K[x_1,\ldots,x_n]$, y F un conjunto de polinomios no nulos de n variables, se verifica*

$$\forall m \in M_{K,n},\ \textit{no nulo};\ \forall f_i \in F\ \textit{tal que}$$
$$(f =_p g - f_i \cdot_M m) \wedge [(hterm\ (f_i \cdot_M m)) \notin f] \Rightarrow g \xrightarrow{F} f$$

Prueba: Se explicita el monomio m de la forma $(y, y0)$, y elegimos para la reducción el polinomio f_i y el término $hterm\ (f_i \cdot_M (y,y0))$. Tenemos que verificar que estos valores cumplen las hipótesis de la reducción buscada, lo cual se traduce en tres submetas a demostrar:

- $hterm\ (f_i \cdot_M (y,y0)) \in g$, que se resuelve utilizando las dos hipótesis iniciales $hterm\ (f_i \cdot_M (y,y0)) \notin f$ y $f =_p g - f_i \cdot_M (y,y0)$. Resaltar que se necesita introducir en el contexto, la hipótesis $(f_i \cdot_M m) \neq_p vpol$, obtenida por el teorema (6.3.20) a través de las hipótesis $H2 : (pol\_In\_ensemb\ fi\ F)$ y $H4 : \neg(z\_monom\ (a,b))$.
- $(hterm\ f_i)|(hterm\ (f_i \cdot_M m))$, directamente manipulando las propiedades de la función *hterm*.
- $f =_p g - f_i \cdot_M \left(\frac{[coef\ (hterm\ (f_i \cdot_M m))\ g]}{hcoef\ f_i}, \frac{hterm\ (f_i \cdot_M m)}{hterm\ f_i}\right)$, que se prueba mediante resultados de las operaciones algebracias y elementos destacados de un polinomio, así como con propiedades de las operaciones sobre T^n y K.

Traducción en **Coq**:

```
Lemma sufc_Red1: (f,g,fi:pol)(F:(list pol))(full_fam F)->(full_term_pol g)->
   (full_term_pol f)->(pol_In_ensemb fi F)->(m:monom)(full_mon n m)->
      ~(z_monom m)->(equpol f (suma_app g (pol_opp (mult_m fi m))))->
         (no_term_in_pol f (hterm (mult_m fi m)))->(Red1 F g f).
```

□

Enunciamos y probamos a continuación diferentes lemas que relacionan la clausura reflexiva y transitiva de la reducción módulo un conjunto de polinomios, ($\xrightarrow[*]{F}$), con la pertenencia al ideal generado por dicho conjunto. En todos los casos, para simplificar la prueba, demostramos primero el resultado para la reducción en un solo paso ($\xrightarrow{F}$). Las pruebas de todos ellos son similares: se utiliza la hipótesis de inducción obtenida de la reducción por un polinomio del conjunto F y las submetas resultantes se resuelven utilizando la formalización y propiedades de los ideales, probadas en el capítulo anterior, así como algunas otras relativas a operaciones sobre polinomios.

Teorema 9.2.2. *Dados $g, h \in K[x_1, \ldots, x_n]$, y F un conjunto de polinomios no nulos de n variables, se verifica*

$$g \xrightarrow[*]{F} h \Rightarrow (g -_p h) \in < F >$$

Traducción en **Coq**:

```
Lemma Red1_dif_id: (g,h:pol)(F:(list pol))(Red1 F g h)->
        (Ideal (suma_app g (pol_opp h)) F).


Lemma Red3_dif_id: (g,h:pol)(F:(list pol))(Red3 F g h)->
        (Ideal (suma_app g (pol_opp h)) F).
```

□

Un caso particular del teorema anterior es el corolario siguiente.

Corolario 9.2.2. *Dados $g \in K[x_1, \ldots, x_n]$ y F un conjunto de polinomios no nulos de n variables, si g se reduce a vpol entonces g está en el ideal generado por F, es decir*

$$g \xrightarrow[*]{F} vpol \Rightarrow g \in < F >$$

Traducción en **Coq**:

```
Lemma Red1_vpol: (g:pol)(F:(list pol))(Red1 F g vpol)->(Ideal g F).


Lemma Red_vpol: (g:pol)(F:(list pol))(Red3 F g vpol)->(Ideal g F).
```

□

Verificamos dos lemas técnicos simétricos.

Proposición 9.2.8. *Dados $g, h \in K[x_1, \ldots, x_n]$ y F un conjunto de polinomios no nulos de n variables, si g se reduce a h, módulo F, entonces h está en el ideal generado por g y F, es decir*

$$g \xrightarrow[*]{F} h \Rightarrow h \in < g : F >$$

Traducción en **Coq**:

```
Lemma Red1_en_cons: (F:(list pol))(g,h:pol)(Red1 F g h)->(Ideal h (cons g F)).


Lemma Red3_en_cons: (F:(list pol))(g,h:pol)(Red3 F g h)->(Ideal h (cons g F)).
```
□

Proposición 9.2.9. *Dados $g, h \in K[x_1, \ldots, x_n]$ y F un conjunto de polinomios no nulos de n variables, si g se reduce a h, módulo F, entonces g está en el ideal generado por h y F, es decir*

$$g \xrightarrow[*]{F} h \Rightarrow g \in < h : F >$$

Traducción en **Coq**:

```
Lemma Red1_en_cons_inv: (F:(list pol))(g,h:pol)(Red1 F g h)->
                        (Ideal g (cons h F)).


Lemma Red3_en_cons_inv: (F:(list pol))(g,h:pol)(Red3 F g h)->
                        (Ideal g (cons h F)).
```
□

Probamos a continuación la relación que existe entre reducción y pertenencia a un ideal de los polinomios componentes. La prueba de este resultado se sigue de la formalización de la reducción y de las propiedades de los ideales. Para simplificar la prueba, en primer lugar se demuestra para la reducción en un solo paso.

Teorema 9.2.3. *Dados $g, h \in K[x_1, \ldots, x_n]$, y F un conjunto de polinomios no nulos de n variables, se verifica*

$$(g \xrightarrow[*]{F} h) \wedge (g \in < F >) \Rightarrow h \in < F >$$

es decir, el resultado de reducir un polinomio del ideal $< F >$ por el conjunto F está en el ideal.

Traducción en **Coq**:

```
Lemma Red1_en_id: (g,h:pol)(F:(list pol))(Red1 F g h)->
                  (Ideal g F)->(Ideal h F).


Lemma Red3_en_id: (g,h:pol)(F:(list pol))(Red3 F g h)->
                  (Ideal g F)->(Ideal h F).
```
□

Para finalizar, se prueba el teorema recíproco del anterior; es decir, que si el polinomio reducido por $\xrightarrow[*]{F}$ está en el ideal $< F >$, el que se reduce también está. La prueba es simétrica de la anterior.

Teorema 9.2.4. *Dados $g, h \in K[x_1, \ldots, x_n]$, y F un conjunto de polinomios no nulos de n variables, se verifica*

$$(g \xrightarrow[*]{F} h) \wedge (h \in < F >) \Rightarrow g \in < F >$$

Traducción en **Coq**:

```
Lemma Red1_en_id_inv: (g,h:pol)(F:(list pol))(Red1 F g h)->
                      (Ideal h F)->(Ideal g F).


Lemma Red3_en_id_inv: (g,h:pol)(F:(list pol))(Red3 F g h)->
                      (Ideal h F)->(Ideal g F).
```

□

9.3. Conexión entre congruencia y reducción

En esta sección relacionamos la reducción de polinomios en $K[x_1,\ldots,x_n]$, formalizada anteriormente, con la relación de congruencia sobre $K[x_1,\ldots,x_n]$ inducida por los ideales de polinomios. Recordamos que para cada conjunto de polinomios $F \subseteq K[x_1,\ldots,x_n]$, $< F >$ es el ideal generado por F en $K[x_1,\ldots,x_n]$ (ver Definición 8.1.2).

9.3.1. Congruencia módulo un conjunto de polinomios

Definición 9.3.1. *Dados $f,g \in K[x_1,\ldots,x_n]$, y F un conjunto de polinomios no nulos de n variables,*

$$f \equiv_F g \Leftrightarrow f -_p g \in < F >$$

se dice que f es congruente con g módulo F.

Traducción en **Coq**:

```
Inductive equiv_Id [F:(list pol);f:pol]: pol->Prop :=
 equiv_Id_cons: (g:pol)(full_fam F)->(full_term_pol f)->(full_term_pol g)->
                (Ideal (suma_app f (pol_opp g)) F)->(equiv_Id F f g).
```

Ejemplo 9.3.1. *Si $F =< x^2 - y^2, x + y^3 + 1 >$, entonces $f = x^4 + x - y^4$ y $g = x^5 + x^4y^3 + x^4 + x$ son congruentes módulo F ya que*

$$f - g = -x^5 - x^4y^3 - y^4 = (x^2 + y^2)(x^2 - y^2) - (x^4)(x + y^3 + 1)$$

Directamente de su formalización y de las propiedades de ideal comprobamos que *la relación $\equiv_F$ es una relación de equivalencia sobre el anillo de polinomios $K[x_1,\ldots,x_n]$.*

Traducción en **Coq**:

```
Lemma equiv_id_reflex: (F:(list pol))(f:pol)(full_fam F)->(full_term_pol f)->
                       (equiv_Id F f f).
```

```
Lemma equiv_id_sym: (F:(list pol))(f,g:pol)(equiv_Id F f g)->(equiv_Id F g f).


Lemma equiv_id_trans: (F:(list pol))(f,g,h:pol)(equiv_Id F f g)->
                        (equiv_Id F g h)->(equiv_Id F f h).
```

□

Congruencia y reducción están intimamente relacionadas. Para estudiar esta relación utilizamos la clausura reflexiva, simétrica y transitiva de $\xrightarrow{F}$, denotada $\underset{*}{\overset{F}{\longleftrightarrow}}$, es decir

Definición 9.3.2. *Dados* $g, h \in K[x_1, \ldots, x_n]$ *y* $F \subseteq K[x_1, \ldots, x_n]$

$$g \underset{*}{\overset{F}{\longleftrightarrow}} h \Leftrightarrow \textit{existe una secuencia de polinomios verificando que:}$$

$$g = k_0 \overset{F}{\longleftrightarrow} k_1 \overset{F}{\longleftrightarrow} k_2 \overset{F}{\longleftrightarrow} \ldots \overset{F}{\longleftrightarrow} k_n = h$$

en donde

$$g \overset{F}{\longleftrightarrow} h \Leftrightarrow (o \quad g \xrightarrow{F} h, \textit{ o bien } \quad h \xrightarrow{F} g)$$

Traducción en **Coq**:

```
Inductive Red_equiv [F:(list pol)] : pol->pol->Prop :=
 Red_equiv_eq: (g,h:pol)(full_fam F)->(full_term_pol g)->(full_term_pol h)->
                 (equpol g h)->(Red_equiv F g h)
|Red_equiv_step: (g,h:pol)(full_fam F)->(full_term_pol g)->(full_term_pol h)->
                 (Red1 F g h)->(Red_equiv F g h)
|Red_equiv_sym: (g,h:pol)(full_fam F)->(full_term_pol g)->(full_term_pol h)->
                 (Red_equiv F g h)->(Red_equiv F h g)
|Red_equiv_trans: (g,f,h:pol)(full_fam F)-> (full_term_pol g)->
                   (full_term_pol f)->(full_term_pol h)->(Red_equiv F g f)->
                      (Red_equiv F f h)->(Red_equiv F g h).
```

A partir del lema de extensionalidad de la reducción por un conjunto de polinomios $\xrightarrow{F}$ (pág 177) y utilizando la propiedad transitiva de la relación *equpol* se obtiene fácilmente la extensionalidad respecto a *equpol* de la relación anterior $\overset{F}{\longleftrightarrow}$.

Traducción en **Coq**:

```
Lemma Red_equiv_ext: (G : (list pol))(g,h:pol)(Red_equiv G g h)->
        (g':pol)(equpol g g')->(full_term_pol g')->(Red_equiv G g' h).
```

□

Desarrollamos algunos lemas técnicos elementales sobre el predicado anterior. Se utilizarán, especialmente, para agilizar las pruebas de esta sección.

Traducción en **Coq**:

```
Lemma Red3_equiv: (f,g:pol)(F:(list pol))(Red3 F f g)->(Red_equiv F f g).


Lemma Red_equiv_full_1: (f,g:pol)(F:(list pol))(Red_equiv F f g)->
                          (full_term_pol f).
```

```
Lemma Red_equiv_full_r: (f,g:pol)(F:(list pol))(Red_equiv F f g)->
                        (full_term_pol g).
```

```
Lemma Red_equiv_tran: (f,g,h:pol)(F:(list pol))(Red_equiv F f g)->
                                  (Red_equiv F g h)->(Red_equiv F f h).
```

```
Lemma Red_equiv_sy:  (f,g:pol)(F:(list pol))
                       (Red_equiv F f g)->(Red_equiv F g f).
```

□

Queremos probar la equivalencia ($\overset{F}{\underset{*}{\longleftrightarrow}} \iff \equiv_F$) entre la clausura reflexiva, simétrica y transitiva de $\overset{F}{\longrightarrow}$ y la congruencia módulo un conjunto de polinomios.

Proposición 9.3.1. *Dados $f, g \in K[x_1, \ldots, x_n]$ y F un conjunto de polinomios no nulos de n variables, se verifica*

$$f \overset{F}{\underset{*}{\longleftrightarrow}} g \Rightarrow (f -_p g) \in < F >$$

Prueba: Aplicando recursión sobre la formalización de $\overset{F}{\underset{*}{\longleftrightarrow}}$ se obtienen cuatro submetas, una por cada constructor. La primera se resuelve utilizando el primer constructor de *Ideal* y propiedades de la función *suma_app*; la segunda es el teorema 9.2.2; la dos últimas son consecuencia de que la pertenencia a un ideal es estable por la relación *equpol* y de la proposición 8.2.4.

Traducción en **Coq**:

```
Lemma con_red_cong: (f,g:pol)(F:(list pol))(Red_equiv F f g)->
                          (Ideal (suma_app f (pol_opp g)) F).
```

□

Como consecuencia immediata de esta proposición y de la formalización de congruencia obtenemos el resultado siguiente.

Corolario 9.3.1. *Sea $f, g \in K[x_1, \ldots, x_n]$ y F un conjunto de polinomios no nulos de n variables, se verifica*

$$f \overset{F}{\underset{*}{\longleftrightarrow}} g \Rightarrow f \equiv_F g$$

Prueba: Por la proposición anterior $f \overset{F}{\underset{*}{\longleftrightarrow}} g \Rightarrow (f -_p g) \in < F >$, que por definición es $f \equiv_F g$.

Traducción en **Coq**:

```
Lemma conv_congr: (F:(list pol))(f,g:pol)(Red_equiv F f g)->(equiv_Id F f g).
```

□

Nota: Si $g \xrightarrow{f} h$ entonces tenemos que $h =_p g -_p f \cdot_M m$ para algún monomio m verificándose además la cancelación de un término de g y, como veremos más adelante, que g es "mayor" que h. Por el contrario, si $h =_p g -_p f \cdot_M m$ para

algún monomio m, de esto no podemos deducir que ocurra la cancelación de algún término de g, ni que g sea "mayor" que h.

Para completar la prueba de que la clausura reflexiva, simétrica y transitiva de $\xrightarrow{F}$ es equivalente a la congruencia módulo un conjunto de polinomios, se utilizará el resultado $f \in < F > \iff f =_p \sum m_i \cdot_M h_i$ con $m_i \in M_{K,n}$, y $h_i \in F$ (es decir monomio a monomio en vez de polinomio a polinomio). En este resultado aparece una suma finita de productos de polinomios por monomios que es necesario formalizar de forma explícita.

Dado un conjunto de polinomios F, la expresión de que un polinomio f se diferencia de otro g en una suma finita de productos de monomios por polinomios de F, está recogida en **Coq** en el predicado inductivo *suma_pol_mon*. La notación que se utilizará es $f \equiv_F^{m_i} g \iff f -_p g =_p \sum m_i \cdot_M h_i$ con $m_i \in M_{K,n}$, $h_i \in F$.

Traducción en **Coq**:

```
Inductive suma_pol_mon [F:(list pol); f:pol] : pol->Prop :=
   suma_pol_mon1: (f0:pol)(full_fam F)->(full_term_pol f)->
      (full_term_pol f0)->(equpol f f0)->(suma_pol_mon F f f0)
|  suma_pol_mon2: (g0:pol)(g:pol)(full_fam F)->(full_term_pol f)->
      (full_term_pol g0)->(full_term_pol g)->(suma_pol_mon F f g)->
         (m:monom)(full_mon n m)->(fi:pol)(pol_In_ensemb fi F)->
            (equpol g0 (suma_app g (mult_m fi m)))->(suma_pol_mon F f g0).
```

Esta formalización requiere probar que la definición inductiva *suma_pol_mon es estable por equpol y que verifica las propiedades reflexiva, simétrica y transitiva*. Las pruebas se derivan de la definición del predicado *suma_pol_mon*, en combinación con la aplicación de resultados de manipulación de polinomios.

```
Lemma equpol_Id_sum2: (p,q,p':pol)(F:(list pol))(equpol p q)->
   (suma_pol_mon F p p')->(full_term_pol q)->(suma_pol_mon F q p').


Lemma equpol_Id_sum2_op: (p,q,p':pol)(F:(list pol))(suma_pol_mon F p' p)->
                   (full_term_pol q)->(equpol p q)->(suma_pol_mon F p' q).


Lemma reflex_equiv_Id2: (F:(list pol))(f:pol)(full_fam F)->
                           (full_term_pol f)->(suma_pol_mon F f f).


Lemma trans_equiv_Id2: (f,g,h:pol)(F:(list pol))(suma_pol_mon F f g)->
                          (suma_pol_mon F g h)->(suma_pol_mon F f h).


Lemma sym_equiv_Id2: (f,g:pol)(F:(list pol))(suma_pol_mon F f g)->
                        (suma_pol_mon F g f).
□
```

A continuación probamos algunos lemas técnicos sobre la relación $\equiv_F^{m_i}$ que se utilizarán en pruebas posteriores. Las pruebas de los tres primeros se derivan inmediatamente de la definición de *suma_pol_mon* y de la utilización de simplificaciones polinómicas desarrolladas en la sección 6.3.2.

Lema 9.3.1. *Dados $f, g, h \in K[x_1, \ldots, x_n]$, y F un conjunto de polinomios no nulos de n variables, si h se diferencia de $(f -_p g)$ en una suma finita de productos de monomios por polinomios de F, entonces a $(h +_p g)$ le ocurre lo mismo respecto del polinomio f.*

Traducción en **Coq**:

```
Lemma equat_equiv_Id: (f,g,h:pol)(F:(list pol))(full_term_pol f)->
   (full_term_pol g)->(suma_pol_mon F (suma_app f (pol_opp g)) h)->
                        (suma_pol_mon F f (suma_app h g)).
```

□

Lema 9.3.2. *Si un polinomio $f \in K[x_1, \ldots, x_n]$ es una suma de productos de monomios por elementos de un conjunto de polinomios no nulos F, también lo es si ampliamos el conjunto F con un polinomio no nulo[3] p, de n variables.*

Traducción en **Coq**:

```
Lemma equiv_Id_cpol: (f,p:pol)(F:(list pol))(full_term_pol p)->
                        (~(equpol p vpol))->(suma_pol_mon F f vpol)->
                           (suma_pol_mon (cons  p F) f vpol).
```

□

Lema 9.3.3. *La relación suma_pol_mon es monótona respecto a la suma de polinomios.*

Traducción en **Coq**:

```
Lemma equiv_m: (F:(list pol))(f,g:pol)(m:monom)(full_mon n m)->
         (suma_pol_mon F f g)->(suma_pol_mon F (cpol m f) (cpol m g)).


Lemma monot_suma_pol_mon: (F:(list pol))(f,g,h:pol)(full_term_pol h)->
        (suma_pol_mon F f g)->(suma_pol_mon F (suma_app h f) (suma_app h g)).
```

□

Para ilustrar claramente la diferencia que presenta la demostración formal de una propiedad matemática realizada en Coq frente a la demostración usual con lápiz y papel, mostramos la prueba del lema siguiente en las dos versiones.

Lema 9.3.4. *Dados $f, h \in K[x_1, \ldots, x_n]$ y $F \cup \{g\}$ un conjunto de polinomios de n variables no nulos, si $g._p h$ se diferencia de f en una suma finita de productos de monomios por polinomios de $F \cup \{g\}$, entonces f se puede expresar como suma finita de productos de monomios por polinomios de $F \cup \{g\}$.*

$$f \equiv^{m_i}_{F \cup \{g\}} g.h \implies f \equiv^{m_i}_{F \cup \{g\}} vpol$$

Prueba con lápiz y papel: Directamente de la propiedad distributiva del producto de polinomios se obtiene que $g.h = \sum g.m'_i$ con los m'_i monomios recorriendo el polinomio h. Entonces $f \equiv^{m_i}_{F \cup \{g\}} g.h \implies f = g.h + \sum m_i.f_i$, de lo que se deduce $f = \sum g.m'_i + \sum m_i.f_i = \sum m''_i.f'_i$, donde tanto los polinomios f_i como los f'_i recorren $F \cup \{g\}$. Así tenemos demostrado que $f \equiv^{m_i}_{F \cup \{g\}} vpol$.

[3] El predicado *suma_pol_mon* se formalizó sobre un conjunto de polinomios no nulos.

Prueba: Se razona por recurrencia sobre h; el caso $h =_p vpol$ se resuelve utilizando que $(h \cdot_p vpol) =_p vpol$ y que *suma_pol_mon* es estable por *equpol* (pág 185). Por otro lado si $h =_p (m : p)$, introducimos en primer lugar tres nuevas hipótesis,

- $H1 : g._M m \equiv^{m_i}_{F\cup\{g\}} vpol$
- $H2 : (g._M m +_p g._p p) \equiv^{m_i}_{F\cup\{g\}} g._p p$
- $H3 : f \equiv^{m_i}_{F\cup\{g\}} g._p p$

Combinando la hipótesis de recurrencia $H : (f \equiv^{m_i}_{F\cup\{g\}} g._p p) \Rightarrow (f \equiv^{m_i}_{F\cup\{g\}} vpol)$ y $H3$ obtenemos la tesis del lema.

Tenemos que probar ahora, en orden inverso, las tres hipótesis introducidas en el contexto.

- Para demostrar $H3$ aplicamos la transitividad de $\equiv^{m_i}_{F\cup\{g\}}$ respecto a la expresión $(g._M m +_p g._p p)$. Se obtienen así dos nuevas submetas: una es la hipótesis $H2$, y la otra $f \equiv^{m_i}_{F\cup\{g\}} (g._M m +_p g._p p)$ se transforma, mediante la estabilidad de $\equiv^{m_i}_{F\cup\{g\}}$ respecto a *equpol*, en $f \equiv^{m_i}_{F\cup\{g\}} (g._p(m : p))$ que es, precisamente, la hipótesis inicial.
- Utilizando, de nuevo, la estabilidad de $\equiv^{m_i}_{F\cup\{g\}}$ respecto a *equpol*, la meta $H2$ se transforma en $(g._p p +_p g._M m)) \equiv^{m_i}_{F\cup\{g\}} (g._p p +_p vpol)$. Aplicando el lema anterior *monot_suma_pol_mon*, quedaría la meta $g._M m \equiv^{m_i}_{F\cup\{g\}} vpol$ que coincide con $H1$.
- Para probar $H1$ aplicamos el constructor *suma_pol_mon*2, eligiendo como elementos intermedios $g := (g._M m)$, $m := (-m)$ y $fi := g$. Sólo quedaría probar $vpol =_p g._M m +_p g._M(-m)$ y $g._M m \equiv^{m_i}_{F\cup\{g\}} g._M m$, que se demuestran automáticamente pues la primera es una propiedad ya probada sobre polinomios y la segunda es el constructor *suma_pol_mon*1.

Traducción en **Coq**:

```
Lemma split_pol_mon: (F:(list pol))(f,g,h :pol)(full_term_pol g)->
(full_term_pol h)->(full_fam F)->(suma_pol_mon (cons g F) f (mult_p g h))->
                                  (suma_pol_mon (cons g F) f vpol).
```

□

Lema 9.3.5. *Si a una suma de productos de monomios por polinomios no nulos de un conjunto F, se le añade el producto de un polinomio no nulo g por un monomio, se obtiene de nuevo una suma de productos de monomios por polinomios de la familia ampliada $F \cup \{g\}$. Es decir, $f \equiv^{m_i}_F vpol \implies (g._M m +_p f) \equiv^{m_i}_{F\cup\{g\}} vpol$.*

Prueba: Como f se puede poner en la forma $f =_p \sum m_i._M h_i$ donde $m_i \in M_{K,n}$ y $h_i \in F$, por el lema (9.3.2), también podemos escribir $f =_p \sum m_i._M h_i$ donde $m_i \in M_{K,n}$ y $h_i \in F \cup \{g\}$. Así por el lema (9.3.3) y la transitividad de la

relación *suma_pol_mon* respecto a $g._M m$, el polinomio $(g._M m +_p f)$ es suma de producto de monomios por polinomios de la familia ampliada $F \cup \{g\}$.

Traducción en **Coq**:

```
Lemma split_mon_mon: (F:(list pol))(f,g:pol)(m:monom)(full_term_pol g)->
          (~(equpol g vpol))->(full_mon n m)->(suma_pol_mon F f vpol)->
□           (suma_pol_mon (cons g F) (suma_app (mult_m g m) f) vpol).
```

Los dos teoremas siguientes prueban la equivalencia $f \equiv_F^{m_i} g \Longleftrightarrow f \equiv_F g$. Demostramos que dado un conjunto de polinomios F, el hecho de que un polinomio se diferencie de otro en una suma finita de productos de monomios por polinomios del conjunto F, es equivalente a que dichos polinomios sean congruentes módulo F.

Teorema 9.3.1. *Sea $f, g \in K[x_1, \ldots, x_n]$ y F un conjunto de polinomios no nulos de n variables, que verifican el predicado suma_pol_mon, entonces $f \equiv_F g$.*

Prueba: Por recurrencia sobre la definición inductiva de *suma_pol_mon*, se obtienen dos metas. La primera meta, cuando $f =_p g$, es trivial utilizando el constructor *ideal_vpol* de la definición de ideal; en cuanto a la segunda $f \equiv_F (g_1 +_p fi._M m)$, esto es $(f -_p g_1 -_p fi.m) \in < F >$, dado que $f \equiv_F g_1$ por hipótesis de recursión y $fi.m \in < F >$ por el corolario 8.2.1, se sigue de las propiedades de *Ideal*.

Traducción en **Coq**:

```
Lemma equiv_Id2_equiv_Id: (f,g:pol)(F:(list pol))(suma_pol_mon F f g)->
□                           (equiv_Id F f g).
```

Para el recíproco se utilizan, entre otros, los lemas técnicos sobre la relación *suma_pol_mon* dados anteriormente.

Teorema 9.3.2. *Sea $f, g \in K[x_1, \ldots, x_n]$ y F un conjunto de polinomios no nulos de n variables, se verifica que si $f \equiv_F g$, entonces f y g se diferencian en una suma finita de monomios por polinomios de F, $(f \equiv_F^{m_i} g)$.*

Prueba: En primer lugar aplicamos recurrencia sobre $(f -_p g) \in < F >$; hay que hacer notar que antes se debe transformar la meta inicial utilizando el lema 9.3.1 en la nueva meta $(f -_p g) \equiv_F^{m_i} vpol$. La primera submeta es consecuencia immediata del constructor *suma_pol_mon1*; la segunda es, $p =_p \sum m_i._M h_i$ con $m_i \in M_{K,n}$ y $h_i \in F \cup \{q\}$, bajo las hipótesis $p =_p q._p q' +_p p'$ y $p' =_p \sum m_i._M h_i$ donde $m_i \in M_{K,n}$ y $h_i \in F$. Esta nueva meta se resuelve por inducción sobre el polinomio q'; el caso $q' =_p vpol$ es consecuencia del constructor *suma_pol_mon1*; el caso $q' =_p (m : q'')$ se resuelve aplicando los lemas (9.3.2, 9.3.3, 9.3.4 y 9.3.5). Notemos que, para finalizar las pruebas, debemos utilizar resultados acerca del número de variables de los términos.

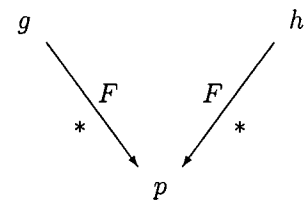
Traducción en **Coq**:

```
Lemma ideal_suma_pol_mon: (F:(list pol))(f,g:pol)(full_term_pol f)->
   (full_term_pol g)->(full_fam F)->(Ideal (suma_app f (pol_opp g)) F)->
      (suma_pol_mon F f g).


Lemma equiv_Id_equiv_Id2: (f,g:pol)(F:(list pol))(equiv_Id F f g)->
□                           (suma_pol_mon F f g).
```

9.3.2. Sucesor común

Para la caracterización de la clausura por la congruencia módulo un conjunto de polinomios, objetivo último de esta sección, hemos definido el predicado auxiliar *suma_pol_mon*. Por el mismo motivo, definimos la propiedad de que dos polinomios tengan un sucesor común, mediante el proceso de reducción de dos polinomios por un conjunto de polinomios no nulos.

Definición 9.3.3. *Sean $g, h, p \in K[x_1, \ldots, x_n]$ y F un conjunto de polinomios no nulos de n variables. Llamamos a p* ***sucesor común*** *de g y h respecto a $\xrightarrow[*]{F}$ si,*



Denotaremos dicha relación por $g \downarrow_*^F h$, y en este caso decimos que los polinomios g y h tienen un sucesor común.

Traducción en **Coq**:

```
Inductive common_succ [F:(list pol);g,h:pol]: Prop:=
  common_succ_red: (p:pol)(Red3 F g p)->(Red3 F h p)->(common_succ F g h).
```

Formalizamos resultados elementales sobre la definición *common_succ* y su relación con la definición $\xleftrightarrow[*]{F}$. Las pruebas se basan en la utilización del constructor *common_succ_red* y la transitividad de $\xleftrightarrow[*]{F}$.

Lema 9.3.6. *Dados $f, g, h \in K[x_1, \ldots, x_n]$, y F un conjunto de polinomios no nulos de n variables, se verifica*

1. $f \downarrow_*^F g \Rightarrow f \xleftrightarrow[*]{F} g.$
2. $f \xleftrightarrow[*]{F} g \wedge g \downarrow_*^F h \Rightarrow f \xleftrightarrow[*]{F} h.$

Traducción en **Coq**:

```
Lemma common_is_Red_equiv: (f,g:pol)(F:(list pol))(common_succ F f g)->
                           (Red_equiv F f g).


Lemma trans_equiv_common: (f,g,h:pol)(F:(list pol))(Red_equiv F f g)->
                          (common_succ F g h)->(Red_equiv F f h).
```

□

Explicitando la definición de reducción se puede elegir el polinomio adecuado para ser el sucesor común buscado en cada caso.

Lema 9.3.7. *Dados $f, g, h, h' \in K[x_1, \ldots, x_n]$, y F un conjunto de polinomios no nulos de n variables, se verifica*

1. *La relación $\downarrow_*^F$ es simétrica.*
2. $(h \xrightarrow{F} f) \wedge (g \downarrow_*^F f) \Rightarrow g \downarrow_*^F h.$
3. $(f \xrightarrow{F} h) \wedge (g \xrightarrow{F} h') \wedge (h =_p h') \Rightarrow f \downarrow_*^F g.$
4. $f \xrightarrow{F} g \Rightarrow f \downarrow_*^F g.$

Traducción en **Coq**:

```
Lemma conmt_common_succ: (f,g:pol)(F:(list pol))(common_succ F f g)->
                          (common_succ F g f).


Lemma trans_common_red: (f,g,h:pol)(F:(list pol))(Red1 F h f)->
                         (common_succ F g f)->(common_succ F g h).


Lemma eq_Red_equiv_common: (f,g:pol)(F:(list pol))(h:pol)(Red1 F f h)->
        (h':pol)(Red1 F g h')->(equpol h h')->(common_succ F f g).


Lemma Red1_is_common: (f,g:pol)(F:(list pol))(Red1 F f g)->
                       (common_succ F f g).
```

□

A partir de la extensionalidad de $\xrightarrow[*]{F}$ (pág 177) respecto a *equpol*, se obtiene la extensionalidad de *common_succ*.

Traducción en **Coq**:

```
Lemma comp_common_succ: (p,q,r,s:pol)(F:(list pol))(full_term_pol q)->
        (full_term_pol s)->(equpol p q)->(equpol r s)->
                (common_succ F p r)->(common_succ F q s).
```

□

Utilizando el teorema 9.2.1, se demuestra que los polinomios g y $g -_p f_{i \cdot M} m$ tienen sucesor común. Este resultado que enunciamos y demostramos a continuación será clave tanto para la prueba del teorema siguiente (9.3.4), como para la prueba de la semicompatibilidad de la reducción respecto de la adición de polinomios (lema 9.7.1).

Teorema 9.3.3. *Dados $f, g \in K[x_1, \ldots, x_n]$, y F un conjunto de polinomios no nulos de n variables, se verifica*

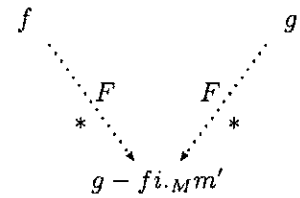
$$\forall m \in M_{K,n};\ \forall f_i \in F;\ (f =_p g -_p f_{i \cdot M} m) \Rightarrow f \downarrow_*^F g$$

Prueba: Se explicita el monomio m de la forma $(y, y0)$ y se procede en dos etapas.

1. Cuando $(y, y0)$ es un monomio no nulo, distinguimos dos casos.
   a) $hterm\ [fi \cdot_M (y, y0)] \in f$, procedemos de nuevo por casos:

1) Si $hterm\ [fi\ ._M\ (y,y0)] \notin g$, como $g =_p f - [fi._M(-m)]$, aplicando el teorema (9.2.1) se tiene que $f \xrightarrow{F} g$. Luego, utilizando la propiedad 4) del lema (9.3.7), se deduce $f \downarrow_*^F g$.

2) Suponiendo que $hterm\ [fi\ ._M\ (y,y0)] \in g$, de la definición *red*, se tiene que $g \xrightarrow{F} g - fi._M \left(\frac{(coef\ \{hterm\ [fi\ ._M\ (y,y0)]\}\ g)}{hcoef\ fi}, y0\right)$. Por otro lado de la hipótesis $g =_p f +_p fi._M(y,y0)$ deducimos que el coeficiente de $hterm\ [fi\ ._M\ (y,y0)]$ en g es igual al coeficiente de $hterm\ [fi\ ._M\ (y,y0)]$ en f más $(y\ ._K\ [hcoef\ fi])$. Procediendo de la misma manera que en la reducción anterior tenemos que $f \xrightarrow{F} g - fi._M \left(\frac{(coef\ \{hterm\ [fi\ ._M\ (y,y0)]\}\ g)}{hcoef\ fi}, y0\right)$. Con lo cual tenemos efectivamente que



donde $m' =_M \left(\frac{(coef\ \{hterm\ [fi\ (y,y0)]\}\ g)}{hcoef\ fi}, y0\right)$

b) Si $hterm\ [fi\ (y,y0)] \notin f$, estamos en la condiciones del teorema 9.2.1; por consiguiente tenemos que $g \xrightarrow{F} f$, y mediante el lema 9.3.7, se deduce $f \downarrow_*^F g$.

2. Si $(y,y0)$ es el monomio nulo, es suficiente, mediante las operaciones de polinomios, obtener que $f =_p g$, y así de la formalización de *common_succ*, se deduce que $f \downarrow_*^F g$, lo que acaba la prueba.

Traducción en **Coq**:

```
Lemma sufc_common1: (f,g,fi:pol)(F:(list pol))(full_fam F)->
(full_term_pol f)->(full_term_pol g)->(pol_In_ensemb fi F)->
(m:monom)(full_mon n m)->~(z_monom m)->
   (equpol f (suma_app g (pol_opp (mult_m fi m))))->(common_succ F f g).


Lemma sufc_common: (f,g,fi:pol)(F:(list pol))(full_fam F)->(full_term_pol f)->
  (full_term_pol g)->(pol_In_ensemb fi F)->(m:monom)(full_mon n m)->
     (equpol f (suma_app g (pol_opp (mult_m fi m))))->(common_succ F f g).
```

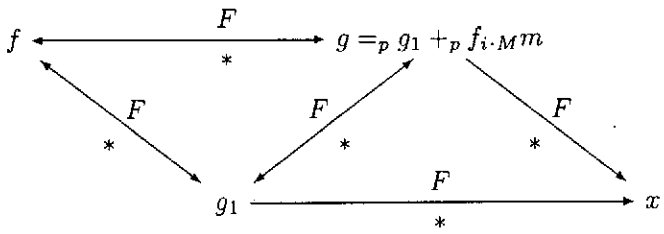
□

Ahora estamos en condiciones de demostrar el recíproco del corolario 9.3.1, cerrando así la equivalencia ($\xleftarrow[*]{F} \iff \equiv_F$). Hay que hacer notar que se utiliza la relación *suma_pol_mon* la cual nos permite expresar de otra forma la pertenencia de la diferencia de dos polinomios a un *ideal*.

Teorema 9.3.4. *Sea $f, g \in K[x_1, \ldots, x_n]$ y F un conjunto de polinomios no nulos de n variables, se verifica*

$$f \equiv_F g \Leftrightarrow f \equiv_F^{m_i} g \Rightarrow f \overset{F}{\underset{*}{\longleftrightarrow}} g$$

Prueba: Por recurrencia sobre la definición de *suma_pol_mon*. El caso del primer constructor, cuando $f =_p g$, es trivial. La segunda submeta $f \overset{F}{\underset{*}{\longleftrightarrow}} g$, en donde $g =_p g_1 +_p f_{i \cdot M} m$, se resuelve aplicando en primer lugar la transitividad de *Red_equiv* respecto a g_1. En efecto, $f \overset{F}{\underset{*}{\longleftrightarrow}} g_1$ es una de la hipótesis obtenidas por recurrencia, mientras que $g_1 \overset{F}{\underset{*}{\longleftrightarrow}} g$ se obtiene combinando la aplicación de la propiedad 1 del lema 9.3.6 junto con el teorema anterior, según el siguiente esquema:



Traducción en **Coq**:

```
Lemma congr_conv : (F:(list pol))(f,g:pol)(suma_pol_mon F f g)->
                   (Red_equiv F f g).
```

□

9.4. Noetherianidad de la reducción

La noetherianidad [102, 66, 99] es sin duda, entre las nociones clásicas del álgebra, la que tiene un tratamiento constructivo más delicado.

Examinando las exigencias de las matemáticas constructivas sobre la noetherianidad se aprecia que no se trata simplemente de proporcionar algoritmos; hace falta que las pruebas de su terminación sean constructivas. Se trata de una exigencia que roza la epistemología, y que es difícil de definir si no es por la práctica.

En nuestro caso, si queremos obtener el algoritmo que calcule la forma normal de un polinomio módulo un conjunto de polinomios, se necesita la propiedad de noetherianidad[4] de la reducción para garantizar la terminación de los cálculos.

Definición 9.4.1. *Una relación R definida en un conjunto A es* **noetheriana** *si no existe una sucesión infinita de la forma $a_0\ R\ a_1\ R\ a_2\ R\ \ldots$.*

[4]La noetherianidad garantiza la terminación de los cálculos sea cual sea la estrategia de reducción adoptada.

Otra definición posible para la noetherianidad es: **una relación R es noetheriana si, y sólo si, R^{-1} es bien fundada, siendo R^{-1} la relación simétrica de R** $(a\ R^{-1} b \Leftrightarrow b\ R\ a)$.

Pensamos que esta última definición de la noetherianidad es la más apropiada en nuestro caso y la formalizamos en **Coq** por:

Traducción en **Coq**:

```
Definition simetrico [A:Set;R:A -> A -> Prop; a,b:A] : Prop:= (R b a).


Definition noetheriano := [A:Set][R:A -> A -> Prop]
                (well_founded A (simetrico A R)).
```

Para demostrar que la relación de reducción por un polinomio dado es noetheriana, se necesita que el orden definido sobre polinomios sea bien fundado. Como esta propiedad sobre polinomios se ha probado sobre el tipo *pol_full* (ver pág 156), necesitamos formalizar la notación de la reducción por un polinomio para dicho tipo.

Traducción en **Coq**:

```
Definition inc_red: pol_full -> pol_full -> pol_full -> Prop :=
            [p,q,r:pol_full] (red (inc p)  (inc q) (inc r)).
```

Ya hemos visto que el algoritmo de reducción de un polinomio f módulo un conjunto de polinomios F es no determinista. Sin embargo, siempre termina: probaremos que $\xrightarrow{F}$ es noetheriana. Esto se debe esencialmente al hecho que cuando en el paso de reducción se elimina un término t de g, entonces los términos t' de g que verifican $t <_L t'$, no cambian.

Para probar que en el paso de reducción, el polinomio obtenido es menor que el inicial, se necesita el siguiente lema técnico.

Lema 9.4.1.

$$\begin{array}{l}\forall a \in K;\ \forall b, t \in T^n;\ \forall p, q, r \in K[x_1, \ldots, x_n];\ [full\_pol\ ((a,b) : p)] \\ \wedge\ [t \in ((a,b) : p)]\ \wedge (q, r \neq_p vpol) \wedge \\ \{((a,b) : p) =_p q +_p [(coef\ ((a,b) : p), t) : vpol] +_p r\} \wedge \\ (\forall t' \in q \Rightarrow t <_L t')\ \wedge\ (\forall t' \in r \Rightarrow t' <_L t) \\ \Longrightarrow\ t \in p\end{array}$$

Prueba: Comenzamos utilizando la decidibilidad de términos $(t \neq b) \vee (t = b)$; así tenemos dos casos:

- El caso $t \neq b$, una vez desdoblado la notación *term_in_pol*, es consecuencia del teorema 6.4.5 y de la hipótesis $t \in ((a,b) : p)$.
- Si $t = b$. De la hipótesis $(\forall t' \in q \Rightarrow t <_L t')$ obtenemos que $t <_L (hterm\ q)$, utilizando las reglas de reescritura y el lema 6.8.8 dos veces tenemos que $(hterm\ q) = (hterm\ (q +_p [(coef\ ((a,t) : p), t) : vpol] +_p r)$. Combinado estos dos últimos resultados, por la transitividad de términos y los lemas de

la página 140, deducimos que $t <_L (hterm\ ((a,t) : p))$. Como el polinomio $((a,b) : p)$ es canónico, se obtiene que $(t <_L t)$ lo que es contradictorio con la propiedad irreflexiva del orden $<_L$.

Traducción en **Coq**:

```
Lemma aux1_red_menor2: (a:K)(b,t:term)(p,x:pol)(full_pol (cpol (a,b) p))->
(term_in_pol (cpol (a,b) p) t)->~(equpol x vpol)->(full_term_pol x)->
(equpol (cpol (a,b) p)(suma_app x (cpol ((coef (cpol (a,b) p) t),t) vpol)))->
       ((t':term)(term_in_pol x t')->(Ttm t t'))->(term_in_pol p t).


Lemma aux2_red_menor2: (a:K)(b,t:term)(p,x,x0:pol)(full_pol (cpol (a,b) p))->
(term_in_pol (cpol (a,b) p) t)->~(equpol x vpol)->~(equpol x0 vpol)->
 (full_term_pol x)->(full_term_pol x0)->(equpol (cpol (a,b) p)
  (suma_app x (suma_app (cpol ((coef (cpol (a,b) p) t),t) vpol) x0)))->
   ((t':term)(term_in_pol x t')->(Ttm t t'))->
   ((t':term)(term_in_pol x0 t')->(Ttm t' t))->(term_in_pol p t).
□
```

Teorema 9.4.1. *Si* $f, g, f_i \in K_c[X]$, *se verifica que*

$$f \xrightarrow{f_i} g \Rightarrow g \prec_c f$$

Prueba: Simplificamos las notaciones relativas al tipo *pol_full* para trabajar con el tipo usual *pol* verificando el predicado *full_pol*. Procedemos por inducción sobre el polinomio f: el caso $f = vpol$ se resuelve por inversion sobre la hipótesis $vpol \xrightarrow{f_i} g$ (contradice la definición de *red*). Para resolver el segundo caso aplicamos inducción sobre g. Cuando $g = vpol$ la prueba se obtiene mediante el primer constructor ($Tpol\_Lex3\_v$) del orden definido sobre polinomios.

Ahora la meta es $((a0,b0) : p0) \prec ((a,b) : p)$, con ambos polinomios no nulos por ser canónicos y de tipo $(cpol\ ((c,t) : q))$ (teorema 6.6.8). Aplicando la táctica de inversion sobre la hipótesis $((a,b) : p) \xrightarrow{f_i;t} ((a0,b0) : p0)$ se obtienen tres nuevas hipótesis: $t \in ((a,b) : p)$, $t \notin ((a0,b0) : p0)$ y

$$((a0,b0) : p0) =_p ((a,b) : p) -_p f_i \cdot_M \left(\frac{[\text{coef } t\ ((a,b) : p)]}{h\text{coef } f_i}, \frac{t}{hterm\ f_i} \right)$$

Por el Corolario 6.5.1 podemos descomponer el polinomio inicial de la forma:

$$((a,b) : p) =_p x +_p ((\text{coef } (t, [(a,b) : p])), t) +_p x_0$$

donde $x = H[((a,b) : p), t]$ y $x_0 = L[((a,b) : p), t]$, notaciones de dicho corolario.

Utilizando la decidibilidad de la relación $=_p$, respecto a *vpol*, de los polinomios x y x_0, distinguimos cuatro casos:

1. Cuando $x =_p x_0 =_p vpol$, entonces $((a,b) : p) =_p ((\text{coef } (t, [(a,b) : p])), t)$, utilizando diversos resultados sobre los términos principales de polinomios canónicos obtenemos la igualdad entre los términos $b0$ y

$$hterm \left[((\text{coef}(t, [(a,b) : p])), t) -_p f_i \cdot_M \left(\frac{[\text{coef } t\ ((a,b) : p)]}{\text{hcoef } f_i}, \frac{t}{hterm\ f_i} \right) \right]$$

así, como $b = hterm\ ((\text{coef}(t, [(a,b) : p])), t)$. Aplicando el lema (6.8.14) y la transitividad de términos se obtiene que $b0 <_L b$; se concluye este caso directamente del segundo constructor de $Tpol\_Lex3$.

2. Si $x =_p vpol \wedge x_0 \neq_p vpol$. Así $((a,b) : p) =_p ((\text{coef}\ (t, [(a,b) : p])), t) +_p x_0$; como todos los términos del polinomio x_0 son menores que t, podemos proceder de forma análoga al caso anterior a través del lema (6.8.14) y obtener $b0 <_L b$. Ello nos permite utilizar el segundo constructor de $Tpol\_Lex3$ para probar la submeta.

3. Si $x \neq_p vpol \wedge x_0 =_p vpol$. Así $((a,b) : p) =_p x +_p ((\text{coef}\ (t, [(a,b) : p])), t)$, utilizando lemas técnicos sobre los coeficientes y términos principales, de forma análoga a los dos casos precedentes tenemos

$$b0 = hterm\left[((a,b) : p) -_p f_i \cdot_M \left(\frac{[\text{coef}\ t\ ((a,b) : p)]}{\text{hcoef}\ f_i}, \frac{t}{hterm\ f_i}\right)\right]$$

$$hterm\left[f_i \cdot_M \left(\frac{[\text{coef}\ t\ ((a,b) : p)]}{\text{hcoef}\ f_i}, \frac{t}{hterm\ f_i}\right)\right] <_L hterm((a,b) : p)$$

y $b = hterm((a,b) : p)$. Usando estas propiedades, el lema 6.8.8 deduce que $b = b0$. Combinando esta nueva hipótesis con el tercer constructor de $Tpol\_Lex3$ nos queda mostrar que $p \prec p_0$. Aplicando la hipótesis obtenida por la inducción, transformamos la meta anterior en $p \xrightarrow{f_i;t} p_0$. Es suficiente aplicar el lema anterior con a, b, x y $x_0 =_p vpol$, para deducir que $t \in p$. Las demás precondiciones de la reducción se consiguen aplicando propiedades de las operaciones algebraicas sobre polinomios.

4. Caso $(x =_p \neq_p vpol) \wedge (x_0 \neq_p vpol)$. Se prueba de forma simétrica al caso anterior, obteniendo $b = b0$ y aplicando el constructor $Tpol\_Lex3\_cdr$, pues tenemos que $((a,b) : p) =_p x +_p ((\text{coef}\ (t, [(a,b) : p])), t) +_p x_0$.

Nota: En todos los casos debemos utilizar muchos resultados técnicos sobre la longitud de los términos y las formalizaciones de los elementos destacados de un polinomio que alargan bastante la prueba.

Traducción en **Coq**:

```
Lemma red_menor2: (f,g,fi:pol_full)(inc_red fi f g)->(Tpol_full g f).
□
```

Como consecuencia del lema anterior y del hecho de que $\prec_c$ sobre $K_c[X]$ está bien fundada (teorema 7.3.1), obtenemos que la reducción por un polinomio es una relación noetheriana.

Teorema 9.4.2. *Para cada $f_i \in K_c[X]$, la relación $\xrightarrow{f_i}$ es noetheriana.*

Traducción en **Coq**:

```
Lemma buen_ord_red: (fi:pol_full)(noetheriano ? (inc_red fi)).
```
□

Ahora nos queda probar que la reducción por un conjunto de polinomios canónicos también verifica la propiedad de noetherianidad. Para ello, al igual que en el caso de la reducción por un polinomio, antes debemos formalizar dicha reducción sobre el tipo polinomio canónico, *pol_full*.

Traducción en **Coq**:

```
Definition inc_Red1: (list pol_full)->pol_full->pol_full->Prop:=
   [F:(list pol_full)][p,q:pol_full] (Red1 (inc_list F)  (inc p) (inc q)).
```

Como consecuencia del teorema (9.4.1) y de la extensionalidad de la reducción de polinomios respecto a la relación *equpol* se obtiene inmediatamente el siguiente resultado.

Teorema 9.4.3. *Sean $f, g \in K_c[X]$ y F un conjunto de polinomios canónicos. Se verifica que*

$$f \xrightarrow{F} g \Rightarrow g \prec_c f$$

Traducción en **Coq**:

```
Lemma Red1_menor2: (F:(list pol_full))(f,g:pol_full)(inc_Red1 F f g)->
                   (Tpol_full g f).
```
□

Al igual que hemos hecho para probar la noetherianidad de $\xrightarrow{f_i}$, demostramos la noetherianidad de $\xrightarrow{F}$ a partir del teorema anterior.

Teorema 9.4.4. *Para cada conjunto de polinomios canónicos F, la relación $\xrightarrow{F}$ es noetheriana.*

Traducción en **Coq**:

```
Lemma buen_ord_red1: (F:(list pol_full))(noetheriano ? (inc_Red1 F)).
```
□

9.5. Forma normal

En esta sección, formalizamos el algoritmo para el cálculo de la forma normal de un polinomio módulo un conjunto de polinomios F. La terminación de este algoritmo está garantizada por la noetherianidad de $\xrightarrow{F}$, probada en la sección anterior. Es importante resaltar que esta forma normal no es única en general, y la idea central en la teoría de la bases de Gröbner es "completar" (alargar) F para que la forma normal sea única.

9.5.1. Definiciones y propiedades elementales

Definición 9.5.1. *Un polinomio $f \in K[x_1, \ldots, x_n]$ está en* **forma normal** *(o forma reducida) módulo F si f no es reducible módulo F, es decir no existe un polinomio $g \in K[x_1, \ldots, x_n]$ tal que $f \xrightarrow{F} g$. Se denotará por f_F.*

Traducción en **Coq**:

```
Definition normal: (list pol)->pol->Prop:= [F:(list pol)][f:pol]
                    (g:pol)(~(Red1 F f g)).
```

Algunos autores llaman irreducible módulo F a un polinomio que está en forma normal módulo F; no emplearemos está terminología para evitar confusión con el concepto de "no tener factorización propia".

Ejemplo 9.5.1. *El polinomio h en el ejemplo 9.2.4 está en forma normal módulo $F := \{f_1, f_2, f_3\}$: ningún término de dicho polinomio es múltiplo de los términos principales de los polinomios contenidos en F. Por lo tanto, no se puede reducir por ningún polinomio de F.*

Por otro lado, si $g := 6x^3y + 8x^2 + 5xy + 3x + 5y^2 + 3y - 9$, reduciendo por el polinomio $f_1 \in F$ y por el término $6x^3y$ obtenemos,

$$g \xrightarrow{f_1} g' = -18x^3 - 4x^2y - 2x^2 + 3xy - 3x + 5y^2 + 3y - 9$$

a su vez g' se puede reducir por f_1 y por el término $-4x^2y$,

$$g' \xrightarrow{f_1} g_1 = -18x^3 + 10x^2 + \frac{17}{3}xy + \frac{11}{3}x + 5y^2 + \frac{13}{3}y - 13$$

Ahora g_1 no puede reducirse por ningún polinomio de F, es decir está en forma normal módulo F; por lo tanto g_1 es una forma normal de g módulo F. Decimos una forma normal debido a que si hacemos otra reducción posible del polinomio inicial g utilizando el polinomio f_2 y el término $6x^3y$, se obtiene,

$$g \xrightarrow{f_2} g_2 = -18x^3 + 14x^2 + 8xy + 12x + 5y^2 + 6y - 18$$

g_2 es también una forma normal módulo F y sin embargo $g_1 \neq g_2$

De este ejemplo se infiere que la forma normal de un polinomio módulo un conjunto de polinomios no es única. La elección de tales conjuntos F para que la forma normal de un polinomio sea única juega un papel fundamental en la solución de problemas de ideales polinomiales y nos lleva al concepto de **base de Gröbner.**

Al igual que hemos hecho con otras definiciones anteriores comprobamos la *extensionalidad de la relación normal con respecto a equpol.* La prueba es trivial a partir de la extensionalidad de $\xrightarrow{F}$.

Traducción en **Coq**:

```
Lemma normal_ext: (p,q:pol)(F:(list pol))(normal F p)->(full_term_pol p)->
                                (equpol p q)->(normal F q).
```

□

Un resultado técnico esperado y que induce la notación (quizás desafortunada) $f \xrightarrow[*]{F} f$, utilizada por algunos autores [87, 88], para expresar que el polinomio f está en forma normal, es el siguiente.

Teorema 9.5.1. *Dados $f, g \in K[x_1, \ldots, x_n]$, y F un conjunto de polinomios no nulos de n variables, se verifica que*

$$(f \xrightarrow[*]{F} g) \wedge f_{-F} \Rightarrow (f =_p g) \wedge g_{-F}$$

Prueba: Se razona por recurrencia sobre la hipótesis $f \xrightarrow[*]{F} g$. Esto genera tres casos: el primero se resuelve directamente por el lema anterior; el segundo por contradicción entre las hipótesis obtenidas por recurrencia (g_{0_F}) y $(g_0 \xrightarrow{F} h)$ y; la tercera aplicando la transitividad de *equpol* y de nuevo las hipótesis de recurrencia.

Traducción en **Coq**:

```
Lemma normal_eq_aux: (F:(list pol))(f:pol)(g:pol)(Red3 F f g)->
                     (normal F f)->(equpol f g)/\(normal F g).
```
□

Un resultado útil que se obtiene a partir de la implementación de *polinomio en forma normal* es que *el polinomio vpol está en forma normal*[5]. Se prueba por introducción de la negación: supuesto que $vpol \xrightarrow{F} g$, por inversión se obtiene que existe un término t tal que $t \in vpol$. Desdoblando la formalización de *term_in_pol* se obtiene como meta $(\text{coef } t\ vpol) =_K 0$, que coincide con la definición de la función coef de la página 109.

Traducción en **Coq**:

```
Lemma vpol_normal: (G:(list pol))(normal G vpol).
```
□

A partir del resultado anterior y del teorema 9.5.1 se obtiene el siguiente corolario.

Corolario 9.5.1. *Dados $f \in K[x_1, \ldots, x_n]$, y F un conjunto de polinomios no nulos de n variables, se verifica*

$$(vpol \xrightarrow[*]{F} f) \Rightarrow vpol =_p f$$

Traducción en **Coq**:

```
Lemma Red3_vpol_normal: (F:(list pol))(f:pol)(Red3 F vpol f)->(equpol vpol f).
```
□

9.5.2. Procedimiento de normalización.

Demostramos resultados calculatorios para obtener el algoritmo del cálculo de una forma normal de un polinomio módulo un conjunto de polinomios.

Necesitamos previamente probar algunos lemas de decidibilidad de la reducción, que simplificarán la prueba final.

Lema 9.5.1. *Dados $f, g \in K[x_1, \ldots, x_n]$, se verifica*

$$(full\_pol\ g) \wedge (f \neq_p vpol) \Rightarrow \{\exists\, h \in K[x_1, \ldots, x_n]; g \xrightarrow{f} h\} \vee \{g_{-\{f\}}\}$$

[5] Algunos autores suelen introducirlo como axioma.

Prueba: Este resultado se prueba por recurrencia estructural sobre el polinomio g. En el caso del primer constructor, es decir cuando $g =_p vpol$, se verifica automáticamente la afirmación por el lema *vpol_normal*. Si $g =_p ((y, y0) : p)$ utilizamos la hipótesis de recurrencia para el polinomio p; la existencia de un polinomio x que verifica $p \xrightarrow{f;\, t} x$ permite construir $h =_p ((y, y0) : x)$. La prueba de $g \xrightarrow{f} ((y, y0) : x)$ se hace por el término t, ya que el coeficiente de t en p y en $((y, y0) : p)$ es el mismo por la hipótesis de que g está en forma canónica. En la otra rama de la hipótesis de recurrencia se obtiene que p esta en forma normal módulo f, denotado por $(p_{-\{f\}})$. Mediante la decidibilidad de la divisibilidad de términos, teorema (3.5.5), estudiamos los dos casos posibles,

- $hterm\ (f) \mid y0$. Se tiene que el polinomio buscado es

$$((y, y0) : p) -_p f \cdot_M \left(\frac{y}{\text{hcoef}\ (f)}, \frac{y0}{hterm\ (f)} \right)$$

  es decir que g se reduce por el término de cabeza. Utilizamos aquí de nuevo que g es canónico y diversos resultados sobre términos y coeficientes probados en capítulos anteriores.

- $\neg(hterm\ (f) \mid y0)$. Se prueba por reducción al absurdo: supuesto que existe un polinomio $g0$ tal que $((y, y0) : p) \xrightarrow{f} g0$, por inversión se obtiene que existe un término t tal que $t \in ((y, y0) : p)$, por el cual se reduce $((y, y0) : p)$ módulo f, verificándose que $(hterm\ (f) \mid t)$. Si $t = y0$ contradice el hecho $\neg(hterm\ (f) \mid y0)$, si $t \in p$ encontramos un polinomio h tal que $p \xrightarrow{f;\, t} h$ lo que contradice la hipótesis de recurrencia $(p_{-\{f\}})$.

Traducción en **Coq**:

```
Lemma prev_calcula_fn2: (g,f:pol)(full_pol g)->(full_term_pol f)->
□ ~(equpol f vpol)->{h:pol | (red f g h)}+{(normal (cons f (nil pol)) g)}.
```

Generalizamos el lema anterior a la reducción por un conjunto de polinomios canónicos. La prueba se hace por inducción sobre el conjunto de polinomios canónicos F, aplicando el lema anterior y resultados sobre la reducción a cada una de las submetas obtenidas.

Lema 9.5.2. *Dados $f \in K_c[X]$, y F un conjunto de polinomios canónicos no nulos, se verifica*

$$\{\, \exists\, g \in K_c[X];\ f \xrightarrow{F} g\} \ \vee\ \{f_{-F}\}$$

Traducción en **Coq**:

```
Lemma prev_calcula_fn3bis: (F:(list pol_full))(f:pol_full)
(full_fam (inc_list F))->{g:pol_full|
□ (Red1 (inc_list F) (inc f) (inc g))} + {(normal (inc_list F) (inc f))}.
```

Para obtener el algoritmo que calcule una forma normal de un polinomio módulo un conjunto de polinomios, trabajamos sobre tipo *pol_full* debido a que, como se ha comentado anteriormente, se necesita la propiedad de noetherianidad de la reducción (teorema 9.4.4), para garantizar la terminación de los cálculos.

Teorema 9.5.2. *Dados $f \in K_c[X]$ y F un conjunto de polinomios canónicos no nulos, se verifica*

$$\{\exists\, g \in K_c[X];\ (f \xrightarrow[*]{F} g) \wedge g_{-F}\}$$

Prueba: Utilizando la recurrencia sobre la noetherianidad de la reducción por un conjunto de polinomios ($\xrightarrow{F}$) se obtiene la hipótesis de recurrencia noetheriana $(\forall\, y;\ f \xrightarrow{F} y \Rightarrow \{\exists\, y_0;\ (y \xrightarrow{F} y_0) \wedge y_{0-F}\})$. Aplicando el lema anterior al polinomio f tenemos dos casos,

- $\exists\, p;\ f \xrightarrow{F} p$. Aplicando la hipótesis de recursión noetheriana al polinomio p, mediante la táctica **Elim**, se obtiene un polinomio en forma normal g, que verifica $p \xrightarrow[*]{F} g$. Así el polinomio buscado es g. Al obtener por la transitividad de *Red*3 respecto al polinomio p, la meta $f \xrightarrow[*]{F} g$, deducimos que el polinomio buscado es g.
- f está en forma normal. El polinomio elegido es el propio f por la clausura reflexiva de $\xrightarrow{F}$.

Traducción en **Coq**:

```
Lemma calcula_fnbis: (F:(list pol_full))(f:pol_full)
(full_fam (inc_list F))->{g:pol_full|(normal (inc_list F) (inc g))/\
                           (Red3 (inc_list F) (inc f) (inc g))}.
```
□

A igual que hemos hecho con la función que obtiene el polinomio canónico (página 129) a partir de un polinomio cualquiera, asociamos a un nombre, *for_norm*, la construcción de la forma normal de un polinomio canónico módulo un conjunto de polinomios canónicos no nulos. Asociamos el valor *vpol* al caso en el que el conjunto de polinomios canónicos (*list pol*) contenga algún polinomio nulo. La forma normal de un polinomio p, obtenida mediante esta función se denotará $(for\_norm\ F\ p)$.

Traducción en **Coq**:

```
Definition for_norm: (list pol_full)->pol_full->pol_full.
```
□

Aplicando la herramienta de extracción del sistema a la asignación *for_norm* se obtiene el siguiente código **OCaml**, que calcula una forma normal de un polinomio canónico, módulo un conjunto de polinomios canónicos no nulos.

```
let for_norm f g =
  match full_fam_dec (inc_list f) with
    | true -> calcula_fnbis f g
    | false -> Vpol
```

Los lemas siguientes prueban que el polinomio obtenido al aplicar la función *for_norm* a un polinomio f canónico, está en forma normal, tiene el mismo número de variables que el inicial y verifica que $f \xrightarrow[*]{F} (for\_norm\ F\ f)$. Las pruebas se obtienen directamente de la aplicación del lema *calcula_fnbis*.

Traducción en **Coq**:

```
Lemma norm_corr: (F:(list pol_full))(f:pol_full)(full_fam (inc_list F))->
                  (normal (inc_list F) (inc (for_norm F f))).


Lemma Red3_norm: (F:(list pol_full))(f:pol_full)(full_fam (inc_list F))->
                  (Red3 (inc_list F) (inc f) (inc (for_norm F f))).


Lemma full_term_norm: (F:(list pol_full))
(f:pol_full)(full_fam (inc_list F))->(full_term_pol (inc (for_norm F f))).
□
```

Como aplicación de estos resutados, se comprueba que el resultado de aplicar la función *for_norm* al polinomio nulo es el mismo *vpol*; ocurre lo mismo con un polinomio canónico que está en forma normal.

Traducción en **Coq**:

```
Lemma fn_equpol_vpol:(F:(list pol_full))(f:pol_full)(full_fam (inc_list F))->
              (equpol (inc f) vpol)->(equpol (inc (for_norm F f)) vpol).


Lemma equpol_fn: (F:(list pol_full))(f:pol_full)(full_fam (inc_list F))->
      (normal (inc_list F) (inc f))->(equpol (inc (for_norm F f)) (inc f)).
□
```

A partir de la construcción de la función *for_norm*, y respectivamente, del corolario 9.2.2 y de las proposiciones 9.2.8 y 9.2.9, son inmediatos los tres siguientes teoremas.

Teorema 9.5.3. *Dados $g \in K_c[X]$ y F un conjunto de polinomios canónicos no nulos, si la forma normal de g es vpol entonces g está en el ideal generado por F, es decir*

$$[(for\_norm\ F\ g) =_p vpol] \Rightarrow g \in < F >$$

Traducción en **Coq**:

```
Lemma pol_fn_id2: (F:(list pol_full))(g:pol_full)(full_fam (inc_list F))->
       (equpol (inc (for_norm F g)) vpol)->(Ideal (inc g) (inc_list F)).
□
```

Teorema 9.5.4. *Dados $g \in K_c[X]$ y F un conjunto de polinomios canónicos no nulos, entonces la forma normal de g, módulo F, está en el ideal generado por g y F, es decir*

$$(for\_norm\ F\ g) \in < g : F >$$

Traducción en **Coq**:

```
Lemma fn_p_id: (F:(list pol_full))(g:pol_full)(full_fam (inc_list F))->
                (Ideal (inc (for_norm F g)) (cons (inc g) (inc_list F))).
□
```

Teorema 9.5.5. *Dados $g \in K_c[X]$ y F un conjunto de polinomios canónicos no nulos, entonces g está en el ideal generado por la forma normal de g y el conjunto de polinomios F, es decir*

$$g \in < (for\_norm\ F\ g) : F >$$

Traducción en **Coq**:

```
Lemma pol_fn_id: (F:(list pol_full))(g:pol_full)(full_fam (inc_list F))->
                 (Ideal (inc g) (cons (inc (for_norm F g)) (inc_list F))).
```

□

Comprobamos que efectivamente, cualquier término de la forma normal módulo F de un polinomio dado, no es divisible por el término principal de ningún polinomio del conjunto F.

Teorema 9.5.6. *Dados $g \in K_c[X]$ y F un conjunto de polinomios canónicos no nulos, se verifica*

$$(\forall f \in F) \wedge (\forall t \in (for\_norm\ F\ g)) \Rightarrow \neg\ [(hterm\ f)|t]$$

Prueba: Introduciendo en el contexto de hipótesis la negación de la conclusión se obtiene como hipótesis que $(hterm\ f)|t$. Utilizando esta nueva hipótesis en combinación con la formalización de la reducción, se puede deducir que el polinomio $(for\_norm\ F\ g)$ se puede reducir módulo f por el término t, con $f \in F$, lo que contradice la definición de $(for\_norm\ F\ g)$.

Traducción en **Coq**:

```
Lemma no_div_term_fn: (F:(list pol_full))(g,f:pol_full)
(full_fam (inc_list F))->(pol_In_ensemb (inc f) (inc_list F))->
(t:term)(term_in_pol (inc (for_norm F g)) t) -> ~(term_div (hterm (inc f)) t).
```

□

La decidibilidad de la forma normal de un polinomio canónico módulo un conjunto de polinomios, respecto del polinomio nulo (*vpol*) se obtiene automáticamente de la aplicación de la decidibilidad, respecto a *equpol*, de polinomios cualesquiera.

Traducción en **Coq**:

```
Lemma for_norm_dec_vpol: (F:(list pol_full))(g:pol_full)
(full_fam (inc_list F))->
 {(equpol (inc (for_norm F g)) vpol)}+{~(equpol (inc (for_norm F g)) vpol)}.
```

□

9.6. Relaciones básicas entre reducción y términos principales

Si expresamos un polinomio de varias variables como una lista de monomios ordenados, en orden decreciente, entonces la reducción en un paso **red** (definición 9.2.1) se puede interpretar de la manera siguiente.

$$\begin{array}{rcl} c_1.t_1 + \ldots + c_i.t_i + & c.t & +(\text{monomios menores}) - \\ \{ & (\frac{c}{hcoef\ f_j}, \frac{t}{hterm\ f_j}).M(f_j) & +(\text{monomios menores})\} = \\ c_1.t_1 + \ldots + c_i.t_i + & & +(\text{monomios menores}). \end{array}$$

De este esquema podemos inferir informalmente que, en el caso de la reducción por un término distinto del término de mayor grado (término principal), el polinomio resultante de la reducción tiene el mismo término principal que el del polinomio inicial. Y en el caso de la reducción por el término de cabeza, el polinomio resultante tiene el término principal menor que el inicial.

El teorema siguiente, muy útil en posteriores resultados, formaliza la inferencia anterior.

Teorema 9.6.1. *Dados* $f, g, f_j \in K[x_1, \ldots, x_n]$, *se verifica*

$$(f \xrightarrow{f_j} g) \quad \Rightarrow \quad hterm\ (g) \leq_L hterm\ (f)$$

Prueba: Por casos sobre la decidibilidad de los polinomios f y g respecto a *vpol*. Cuando $f =_p g =_p vpol$, por la extensionalidad de *hterm* respecto a *equpol* y la transitividad de términos, se obtiene fácilmente la igualdad de los términos principales. Si $g =_p vpol$ y $f \neq_p vpol$ se utiliza la decidibilidad de términos (teorema 3.3.1), respecto a $(hterm\ f)$ y $(x_1^0 x_2^0 \ldots x_n^0)$. En el subcaso $(hterm\ f) = (x_1^0 x_2^0 \ldots x_n^0)$ por la proposición 6.8.5 se alcanza la igualdad $(hterm\ f) = (x_1^0 x_2^0 \ldots x_n^0) = (hterm\ g)$; si $(hterm\ f) \neq (x_1^0 x_2^0 \ldots x_n^0)$ se resuelve utilizando la condición de minimalidad de $(x_1^0 x_2^0 \ldots x_n^0)$ para el orden lexicográfico (teorema 3.4.8).

En el caso $g \neq_p vpol$, explicitando la definición de reducción en $f \xrightarrow{f_j} g$, tenemos que existe un término $t \in f$ con $(hterm\ f_j)|t$, tal que

$$g =_p f -_p f_j \cdot_M \left(\frac{(coef\ t\ f)}{hcoef\ (f_j)}, \frac{t}{hterm\ (f_j)} \right)$$

A partir de la existencia de un término $t \in f$, introducimos en el contexto dos nuevas hipótesis: $f \neq_p vpol$ y $t \leq_L hterm\ (f)$, que se deducen de la formalización de la reducción y de la presencia de un término en un polinomio. Estudiamos los dos casos posibles obtenidos a partir de esta última hipótesis:

- $t = hterm\ (f)$. Reescribiendo esta igualdad de términos y utilizando resultados sobre términos y cuerpos probados en capítulos anteriores, deducimos que $M(f) =_M M\left(f_j \cdot_M \left(\frac{(coef\ t\ f)}{hcoef\ (f_j)}, \frac{t}{hterm\ (f_j)}\right)\right)$[6]. Combinando esta nueva hipótesis con $g \neq vpol$ y la igualdad polinómica anterior, obtenemos mediante el lema 6.8.14 (*hterm_eq_suma_Ttm*), que $hterm\ (g) <_L hterm\ (f)$.

- Caso $t < hterm\ (f)$. Procediendo de la misma forma que en el caso anterior se obtiene $hterm\left[f_j \cdot_M \left(\frac{(coef\ t\ f)}{hcoef\ (f_j)}, \frac{t}{hterm\ (f_j)}\right)\right] = t$. De este hecho y de la hipótesis $t < hterm\ (f)$, mediante el lema 6.8.8 (*hterm_suma_Ttm*), se obtiene directamente que $hterm\ (g) = hterm\ (f)$.

Traducción en **Coq**:

[6] Es la reducción por el término de cabeza, es decir el monomio cancelado es el principal de f.

```
Lemma aux_no_hterm_Ttm_in_red: (f,g,fj:pol)(red fj f g)->
                        (Ttm (hterm g) (hterm f)) \/ (hterm g)=(hterm f).
```
□

Como consecuencia inmediata de este teorema y de las propiedades del orden lexicográfico sobre términos, se obtienen los dos corolarios siguientes.

Corolario 9.6.1. *Dados $f, g, f_j \in K[x_1, \ldots, x_n]$, y $t \in T^n$, se verifica*

$$(f \xrightarrow{f_j} g) \wedge (hterm\,(f) <_L t) \quad \Rightarrow \quad hterm\,(g) <_L t$$

Traducción en **Coq**:

```
Lemma no_hterm_Ttm_in_red: (f,g,fj:pol)(t:term)(full n t)->
                      (Ttm (hterm f) t)->(red fj f g)->(Ttm (hterm g) t).
```
□

Corolario 9.6.2. *Dados $f, g, f_j \in K[x_1, \ldots, x_n]$, y $t \in T^n$, se verifica*

$$(f \xrightarrow{f_j} g) \wedge (\forall t_1 \in f;\ t_1 <_L t) \Rightarrow (\forall t_2 \in g;\ t_2 <_L t)$$

Traducción en **Coq**:

```
Lemma no_hterm_Ttm_in_red_bis: (f,g,fj:pol)(t:term)(full n t)->
      ((t1:term)(term_in_pol f t1)->(Ttm t1 t))->(red fj f g)->
            ((t2:term)(term_in_pol g t2)->(Ttm t2 t)).
```
□

9.7. Semicompatibilidad respecto de la adición

Una de las razones por las que los teoremas sobre reducción de polinomios son difíciles de probar (en comparación con pruebas análogas en teoría de reescritura) es que no se verifica en general la compatibilidad de la suma de polinomios respecto de la reducción. Es decir, no es cierto que $f \xrightarrow{F} g \Rightarrow (f +_p h) \xrightarrow{F} (g +_p h)$. Aún así se va a verificar bajo ciertas restricciones; estos resultados serán pieza clave para demostrar la caracterización de que la reducción de polinomios satisface la propiedad de Church-Roser.

Ejemplo 9.7.1. *En el ejemplo 9.2.1 vimos que siendo $f =_p 6x^3y+4x^2y+4y^3-1$ y $f_i =_p 2xy + y^3$, entonces,*

$$6x^3y + 4x^2y + 4y^3 - 1 \xrightarrow{2xy+y^3;\ hred} g =_p -3x^2y^3 + 4x^2y + 4y^3 - 1$$

Tomando $h =_p -6x^3y + 3xy + 1$, se tiene que,

$$f +_p h \xrightarrow{2xy+y^3;\ hred} 3xy + 4y^3 - 2xy^3 \neq_p -6x^3y - 3x^2y^3 - 4x^2y + 4y^3 =_p g +_p h$$

También, tomando $h =_p 6x^2y + 3xy + 1$, se tiene que,

$$f +_p h \xrightarrow{2xy+y^3;\ hred} -3x^2y^3 + 10x^2y + 3xy + 4y^3 =_p g +_p h$$

La clave de estos ejemplos es la relación que hay entre los términos principales de los polinomios f y h; en el primer caso se verifica que $hterm(h) = hterm(f)$ y en el segundo $hterm(h) <_L hterm(f)$

Nota: Las observaciones de este ejemplo se generalizan en los dos siguientes teoremas. Por razones de aplicabilidad se formalizan en Coq de dos formas equivalentes.

Teorema 9.7.1. *Dados $f, g, h, f_i \in K[x_1, \ldots, x_n]$, se verifica*

$$(f \xrightarrow{f_i;\ hred} g) \wedge (hterm\ (h) <_L hterm\ (f)) \Rightarrow (f +_p h) \xrightarrow{f_i;\ hred} (g +_p h)$$

Prueba: De la hipótesis $hterm\ (h) <_L hterm\ (f)$, por el lema (6.8.8) deducimos que $hterm\ (f +_p h) = hterm\ (f)$, y asímismo por el lema (6.8.9) sabemos que $hcoef\ (f +_p h) =_K hcoef\ (f)$. Combinando estas nuevas propiedades con la reducción por el término $t = hterm\ (f)$; de la hipótesis $f \xrightarrow{f_i;\ hred} g$ se obtiene directamente que $(f +_p h) \xrightarrow{f_i} (g +_p h)$. Se finaliza la prueba demostrando las igualdades sobre polinomios que se necesitan en la definición de *red* así como las propiedades referentes al número de variables de los polinomios dados y a la presencia de términos en polinomios.

Traducción en **Coq**:

```
Lemma hred_suma: (f,g,fi:pol)(hred fi f g)->(h:pol)(full_term_pol h)->
        (Ttm (hterm h) (hterm f))->(red fi (suma_app f h) (suma_app g h)).

Lemma hred_suma_bis: (f,g,fi:pol)(hred fi f g)->(h:pol)(full_term_pol h)->
       ((t:term)(term_in_pol h t)->(Ttm t (hterm f)))->
                (red fi (suma_app f h) (suma_app g h)).
```

□

Teorema 9.7.2. *Dados $f, g, h, f_i \in K[x_1, \ldots, x_n]$, se verifica*

$$(f \xrightarrow{f_i} g) \wedge [\forall t \in h \Rightarrow (hterm\ (f) <_L t)] \Rightarrow (f +_p h) \xrightarrow{f_i} (g +_p h)$$

Prueba: Como $f \xrightarrow{f_i} g$, sabemos que existe un término $t \in f$ por el cual el polinomio f se reduce a g módulo f_i. Utilizando el mismo razonamiento que en la prueba precedente de la hipótesis $[\forall t \in h \Rightarrow (hterm\ (f) <_L t)]$ y los lemas (6.8.10 y 6.8.11), se obtiene que, $t \in (f +_p h)$ y $(coef\ t\ f) =_K (coef\ t\ (f +_p h))$. Con lo cual disponemos de todas las condiciones necesarias para poder aplicar el constructor *red1_simpl* de *red* y así probar la meta: $(f +_p h) \xrightarrow{f_i} (g +_p h)$ mediante el término t.

Traducción en **Coq**:

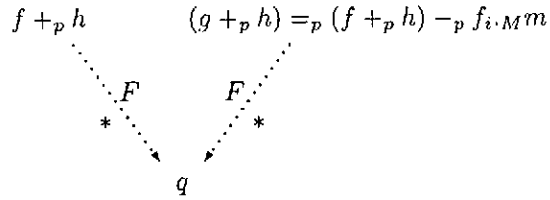
```
Lemma basic2: (f,g,fi:pol)(red fi f g)->(h:pol)(full_term_pol h)->
     ((t:term)(term_in_pol h t)->(t':term)(term_in_pol f t')->(Ttm t' t))->
                (red fi (suma_app f h) (suma_app g h)).
```

```
Lemma basic2_bis: (f,g,fi:pol)(red fi f g)->(h:pol)(full_term_pol h)->
        ((t:term)(term_in_pol h t)->(Ttm (hterm f) t))->
                (red fi (suma_app f h) (suma_app g h)).
```
□

Lema 9.7.1. (Semicompatibilidad) *Dados $f, g, h \in K[x_1, \ldots, x_n]$ y un conjunto de polinomios no nulos de n variables F, se verifica*

$$f \xrightarrow{F} g \Rightarrow (f +_p h) \downarrow_*^F (g +_p h)$$

Prueba: Explicitando la hipótesis $f \xrightarrow{F} g$ se obtiene que $g =_p f -_p f_i.m$ donde $f_i \in F$ y $m \in M_{K,n}$. Por consiguiente $g +_p h =_p (f +_p h) -_p f_i.m$; de esta igualdad y del teorema (9.3.3) se deduce directamente que $(f +_p h) \downarrow_*^F (g +_p h)$, de acuerdo con el siguiente diagrama.



Traducción en **Coq**:

```
Lemma basic3: (h,f,g:pol)(F:(list pol))(Red1 F f g)->(full_term_pol h)->
                (common_succ F (suma_app f h) (suma_app g h)).
```
□

Terminamos esta sección con dos resultados técnicos de mucha utilidad en la caracterización de las bases de Gröbner.

Lema 9.7.2. *Dados $f, g, h, h' \in K[x_1, \ldots, x_n]$ y un conjunto de polinomios no nulos de n variables F, se verifica que si, $(h \xrightarrow[*]{F} h') \wedge (h =_p f -_p g) \Rightarrow$*

$$\exists f', g' \text{ tal que } (f \xrightarrow[*]{F} f') \wedge (g \xrightarrow[*]{F} g') \wedge (h' =_p f' -_p g')$$

Prueba: Al igual que hemos hecho en pruebas anteriores, en primer lugar probamos el resultado bajo la hipótesis de la reducción en un solo paso $\xrightarrow{F}$. De la hipótesis $h \xrightarrow{F} h'$ se obtiene que existe un término $t \in h$ que verifica $h' =_p h -_p f_i \cdot_M m$ donde $f_i \in F$ y $m = \left(\frac{(coef\ t\ h)}{hcoef\ (f_i)}, \frac{t}{hterm\ (f_i)}\right)$. Dependiendo de la presencia o no del término t en los polinomios f y g tenemos los siguientes casos:

1. $t \in f$ y $t \in g$. Procedemos de nuevo por casos, según:

   a) $(coef\ t\ f) =_K (coef\ t\ g)$. Esta nueva hipótesis junto con $h =_p f -_p g$, nos permite deducir que $t \notin h$, lo cual contradice la hipótesis $t \in h$, obtenida anteriormente.

b) $(coef\ t\ f) \neq_K (coef\ t\ g)$. Eligiendo:

$$f' :=_p f -_p f_i \cdot_M \left(\frac{(coef\ t\ f)}{hcoef\ (f_i)}, \frac{t}{hterm\ (f_i)}\right)$$
$$g' := g -_p f_i \cdot_M \left(\frac{(coef\ t\ g)}{hcoef\ (f_i)}, \frac{t}{hterm\ (f_i)}\right)$$

y siguiendo el siguiente diagrama

$$\begin{array}{ccccc} f & - & g & = & h \\ \Big\downarrow f_i;\ t & & \Big\downarrow f_i;\ t & & \Big\downarrow f_i;\ t \\ f' & - & g' & = & h' \end{array}$$

se obtiene el resultado

$$h =_p (f -_p g) \xrightarrow[*]{F} h' =_p f' -_p g' =_p (f -_p g) -_p f_i \cdot_M m'$$

donde,

$m' =_M \left(\frac{(coef\ t\ f) -_K (coef\ t\ g)}{hcoef\ (f_i)}, \frac{t}{hterm\ (f_i)}\right) =_M \left(\frac{(coef\ t\ h)}{hcoef\ (f_i)}, \frac{t}{hterm\ (f_i)}\right)$

2. $t \in f$ y $t \notin g$. Tomando:

$$f' :=_p f -_p f_i \cdot \left(\frac{(coef\ t\ f)}{hcoef\ (f_i)}, \frac{t}{hterm\ (f_i)}\right), \quad \text{y} \quad g' :=_p g$$

de $(coef\ t\ f) =_K (coef\ t\ h)$ se deduce $f \xrightarrow{f_i;\ t} f'$. Como por la clausura reflexiva tenemos que $g \xrightarrow[*]{F} g =_p g'$, se finaliza la prueba al verificar que $h =_p (f -_p g) \xrightarrow[*]{F} h' =_p f' -_p g'$.

3. $t \notin f$ y $t \in g$, es simétrico al anterior. Ahora:

$$f' :=_p f, \quad \text{y} \quad g' := g -_p f_i \cdot \left(\frac{(coef\ t\ g)}{hcoef\ (f_i)}, \frac{t}{hterm\ (f_i)}\right)$$

4. $t \notin f$ y $t \notin g$. Al igual que el caso 1.*a*), obtenemos la contradicción dada por $t \notin h$ y $t \in h$.

En los casos donde la prueba no se obtiene por contradicción, debemos utilizar igualdades sobre polinomios, términos y elementos del cuerpo, así como las propiedades de los términos de n variables, demostrados en capítulos anteriores, que hacen un poco farragosa la prueba.

La demostración para $\xrightarrow[*]{F}$ se obtiene de la correspondiente prueba de la reducción en un solo paso y de los constructores de *Red*3.

Traducción en **Coq**:

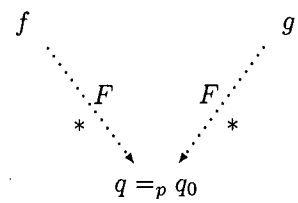
```
Lemma pre_basic4_Red1: (h,h':pol)(F:(list pol))(Red1 F h h')->
   (f:pol)(full_term_pol f)->(g:pol)(full_term_pol g)->
      (equpol h (suma_app f (pol_opp g)))->(EX f' | (Red3 F f f') &
         (EX g' | (Red3 F g g') & (equpol h' (suma_app f' (pol_opp g'))))).


Lemma pre_basic4_Red3: (h,h':pol)(F:(list pol))(Red3 F h h')->
   (f:pol)(full_term_pol f)->(g:pol)(full_term_pol g)->
   (equpol h (suma_app f (pol_opp g)))-> (EX f':pol | (Red3 F f f') &
        (EX g':pol | (Red3 F g g') & (equpol h' (suma_app f' (pol_opp g'))))).
□
```

Teorema 9.7.3. *Dados $f, g \in K[x_1, \ldots, x_n]$ y un conjunto de polinomios no nulos de n variables F, se verifica*

$$(f -_p g) \xrightarrow[*]{F} vpol \Rightarrow f \downarrow_*^F g$$

y, así en particular $f \overset{F}{\underset{*}{\longleftrightarrow}} g$.

Prueba: Aplicando, mediante la táctica **Elim**, el lema anterior con la hipótesis $(f -_p g) \xrightarrow[*]{F} vpol$, se obtienen dos nuevos polinomios, q y q_0, que verifican $f \xrightarrow[*]{F} q$, y $g \xrightarrow[*]{F} q_0$, tales que $q =_p q_0$, pues $(vpol =_p q -_p q_0)$, obteniendo así un sucesor común de f y g.



Traducción en **Coq**:

```
Lemma basic4: (f,g:pol)(F:(list pol))(full_term_pol f)->(full_term_pol g)->
        (Red3 F (suma_app f (pol_opp g)) vpol)->(common_succ F f g).
□
```

Capítulo 10

Bases de Gröbner

10.1. Relevancia práctica

En 1926 Hermann propuso una cuestion [64], más tarde conocida como **el problema de pertenencia a un ideal**; dicha cuestión ha sido presentada de diferentes formas y estudiada en varias áreas.

En 1965 Bruno Buchberger encontró una solución [29] a este problema. Su descubrimiento, llamado bases de Gröbner en honor al director de su tesis (Wolfgang Gröbner), es un conjunto especial de generadores para ideales en un anillo de polinomios en varias variables.

Pasaron alrededor de 10 años antes de que este concepto fuese conocido por las comunidades de investigadores en Matemática y en la Teoría de Ciencias de la Computación. Ahora está ampliamente reconocido y las técnicas que utilizan bases de Gröbner son capaces de resolver muchas cuestiones interesantes de Álgebra, ver [53, 58, 33], con aplicaciones en Física, Química e Ingenieria. Algunas son:

- **Especificación de un ideal**: ¿Todo ideal $I \subset K[x_1, \ldots, x_n]$ tiene un conjunto generador finito? (Teorema de la base de Hilbert, 1.888).
- **Pertenencia a un ideal**: Dado un polinomio $f \in K[x_1, \ldots, x_n]$ y un ideal $I =< f_1, \ldots, f_s >$, determinar si $f \in I$.
- **Resolución de ecuaciones polinomiales**: encontrar todas las soluciones comunes en K^n de un sistema de ecuaciones polinomiales en n variables.
- **Problema de la igualdad**: ¿Son dos polinomios f, f' iguales en el anillo cociente $K[X]/ < F >$? o equivalentemente ¿es $f - f'$ un miembro de $< F >$?
- **Problema de la intersección**: ¿Qué ideal es la intersección en $K[X]$ de dos ideales $< F >$ y $< F' >$?

- **Problemas de simplificación**: Simplificación de términos con respecto a relaciones polinomiales. Este problema tiene dos aspectos: obtener *objetos equivalentes pero más simples* y calcular *representaciones únicas para objetos equivalentes.*

- **Matrices de polinomios**: Dados $F = \{f_1, \ldots, f_n\}$ y $G = \{g_1, \ldots, g_m\}$ dos conjuntos de polinomios, siendo G una base de Gröbner para F. Encontrar la matriz de polinomios Y con m filas y n columnas donde $f_j = \sum_{1 \leq i \leq m} g_i.Y_{i,j}$ (para $j = 1, \ldots, n$).

En $K[x]$ no se plantea ninguno de estos problemas al tratarse de un dominio de ideales principales. Esto ocurre siempre que el anillo es (como en le caso de $\mathbb{Z}$ y de $K[x]$) un dominio euclideo.

La teoría de las bases de Gröbner es una herramienta fundamental en la moderna geometría computacional algebraica y es parte importante del álgebra computacional. En el caso conmutativo está incluida en los mejores y más importantes programas de sistemas de computación simbólica y se aplica en una amplia variedad de áreas de investigación aparentemente no relacionadas entre sí. Por nombrar algunas: aplicaciones en robótica [104, 116], geometría computacional [110], análisis estadístico, probadores de teoremas geométricos [74], técnicas de reescritura [27, 65], etc. Además, se están empezando a aplicar en el modelado de superficies y criptografia. Una conferencia conmemorando los 33 años de las bases de Gröbner fue celebrada en R.I.S.C. en Linz, y las actas [36] contienen artículos sobre diferentes aspectos de las bases de Gröbner que están siendo motivo de investigación. Como dice Barkee en [11] "es inevitable que al igual que la teoría de Galois, la teoría de Buchberger se convierta en una herramienta usada por matemáticos en pruebas".

Resumiendo, la teoría de las bases de Gröbner continua su desarrollo y genera "un interés creciente por su utilidad como herramienta computacional, la cual se aplica en un amplio abanico de problemas en matemáticas, ciencia e ingenieria" [4].

10.2. Nociones básicas. Definición

Los dos primeros problemas, propuestos en la sección anterior, sobre ideales, tienen distintas soluciones en los casos de una y de varias variables.

1. **Pertenencia a un ideal**: Dado el anillo de polinomios $K[x_1, \ldots, x_n]$ y un ideal $I =< f_1, \ldots, f_s >$,

$$DATO: \quad f \in K[x_1, \ldots, x_n]$$

$$PROBLEMA: \quad f \in I$$

es decir, existen polinomios $p_1, \ldots, p_s \in K[x_1, \ldots, x_n]$ de modo que se verifique que $f = p_1.f_1 + \ldots + p_s.f_s$

En el caso de una variable, a partir del algoritmo de la división, dado un polinomio $f \in K[x]$ y un ideal $I =< g >$, dividiendo f por g, obtenemos

$$f = q \cdot g + r$$

en donde $q, r \in K[x]$ y $r = 0$ ó $grad(r) < grad(g)$. Entonces, $f \in I$ si, y sólo si $r = 0$; y así tenemos un test algorítmico para resolver este problema. Utilizando la notación que se ha introducido en las secciones anteriores, para polinomios de varias variables sería: $f \in I =< g > \Leftrightarrow f \xrightarrow[*]{g} 0$.

2. **Especificación de un ideal**: Dado un ideal $I \subset K[x_1, \ldots, x_n]$ ¿existen polinomios $f_i \in K[x_1, \ldots, x_n]$ que verifiquen $I =< f_1, \ldots, f_s >$? En otras palabras, dadas dos familias distintas de polinomios $f_1, \ldots, f_s \in K[x_1, \ldots, x_n]$, y $f'_1, \ldots, f'_t \in K[x_1, \ldots, x_n]$, ¿se verifica la igualdad de ideales $< f_1, \ldots, f_s >=< f'_1, \ldots, f'_t >$?

   En el caso de una variable, $K[x]$, por ser dominio euclideo se verifica que $< f_1, \ldots, f_s >=< mcd(f_1, \ldots, f_s) >$.

Por el contrario, en el anillo $K[x_1, \ldots, x_n]$, si después de hacer la reducción del polinomio f mediante la familia de polinomios $F = (f_1, \ldots, f_s)$ obtenemos un resto nulo, entonces

$$f = a_1 f_1 + \ldots a_s f_s$$

y así $f \in < f_1, \ldots, f_s >$. Por consiguiente $r = 0$ es una condición suficiente para resolver el problema de la pertenencia a un ideal. Sin embargo, no es una condición necesaria como se ve en el ejemplo siguiente.

Ejemplo 10.2.1. *Sean los polinomios $f = x^2y - y$; $f_1 = xy - x$; $f_2 = x^2 - y$ en $\mathbb{Q}[x, y]$, la familia $F = \{f_1, f_2\}$ y el ideal $I =< F >\subset \mathbb{Q}[x, y]$. Utilizando el orden graduado lexicográfico con $x > y$, se obtiene*

$$f \xrightarrow{f_1;\, x^2y} (x^2 - y) \xrightarrow{f_2;\, x^2} 0, \quad \textit{es decir} \quad f \xrightarrow[*]{F} 0$$

Luego $f = xf_1 + f_2$, y así $f \in I$. Sin embargo, si en la reducción, utilizamos primero el polinomio f_2, tenemos que $f \xrightarrow{f_2;\, x^2y} (y^2 - y)$, mientras que $y^2 - y$ no es reducible módulo F. Así, el resto con esta otra reducción (división) de f por F no es cero, pese a que f está en el ideal I.

Incluso en el caso de polinomios de una variable siguiendo el procedimiento del ejemplo, se tiene la misma dificultad. Por ejemplo, siendo $f = x^2$; $f_1 = x^3 + x$; $f_2 = x^3 - x^2 + x$, entonces f no es reducible módulo f_1, f_2, y sin embargo tenemos que $f = f_1 - f_2 \in I =< f_1, f_2 >$. Este problema se resuelve encontrando, como hemos dicho anteriormente, un "buen" conjunto generador para I. Para tal conjunto el resto de la división debería ser único y así la condición $r = 0$ sería equivalente a la pertenencia al ideal. En $K[x]$, por ser dominio de ideales principales, se tiene que $< f_1, f_2 >=< x >=< mcd(x^3 + x, x^3 - x^2 + x) >$. En el anillo $K[x_1, \ldots, x_n]$, que no es un dominio de ideales principales, serán las bases de Gröbner los "buenos" conjuntos generadores.

Definición 10.2.1. *Un conjunto de polinomios no nulos de n variables F, se dice que es una* **base de Gröbner** *(para el ideal generado por F) si se verifica que:*

$$(G1) \quad \forall f \in K[x_1,\ldots,x_n], \quad f \in < F > \iff (f \xrightarrow[*]{F} vpol)$$

En otras palabras, si F es una base de Gröbner para el ideal $< F >$, entonces cualquier polinomio no nulo de $< F >$ es reducible módulo F.

Cuando F es una base de Gröbner y $f \in < F >$ es irreducible, entonces se verifica $f \xrightarrow[*]{F} vpol$ sólo si $f =_p vpol$. Inversamente, si encontramos un h para cualquier $f \in < F >$ tal que $f \xrightarrow[*]{F} h$, siendo h irreducible módulo F, tenemos claramente que $h \in < F >$; así pues h debe ser vpol y entonces $f \xrightarrow[*]{F} vpol$. Esto es, F es una base de Gröbner precisamente cuando la reducción módulo F es un "simplificador" de la forma normal para $K[x_1,\ldots,x_n]/ < F >$.

Nota: La formulación anterior de base de Gröbner es la inicialmente dada por Bruno Buchberger en [31, 6]. Más tarde, como veremos en las siguientes secciones, se dieron otras formulaciones alternativas dirigidas especialmente a proporcionar un modelo constructivo de dichas bases.

Ejemplo 10.2.2. *Sea $< F >$ el ideal generado por los polinomios*

$$g_1 = x^3yz - xz^2,$$

$$g_2 = xy^2z - xyz,$$

$$g_3 = x^2y^2 - z$$

en $K[x,y,z]$.
Una base de Gröbner de este ideal (con respecto al orden lexicográfico, tomando $x > y > z$) es el conjunto

$$G = \{g_2, g_3, x^2yz - z^2, yz^2 - z^2, x^2z^2 - z^3\}$$

Ya que $< F >=< G >$, además para todo $f \in < F >$ se verifica que $f \xrightarrow[*]{G} vpol$, independientemente de la secuencia de reducciones seguida. Como se puede ver en este ejemplo, una base de Gröbner puede contener, y contiene normalmente, más polinomios que el conjunto inicial.

Traducción en **Coq**:

```
Definition Groebner1_full:= [F:(list pol_full)]((full_fam (inc_list F))/\
  (f:pol_full)(Ideal (inc f) (inc_list F))->(Red3 (inc_list F) (inc f) vpol)).
```

Nota: Por las razones dadas anteriormente (ver sección 7.3), implementamos esta formalización de base de Gröbner, al igual que haremos más adelante con las demás formalizaciones alternativas, sobre el tipo *pol_full*.

Esta formulación de base de Gröbner resuelve el problema de la pertenencia a un ideal en polinomios de varias variables. Sin embargo, no nos dice cómo construir la base de Gröbner para un ideal I de una base F. La idea del método de Buchberger es "completar" la base F añadiendo (un número finito) de polinomios. La primera contribución de Buchberger fue demostrar que esta complección sólo necesita una cantidad finita, para un número finito de polinomios de F.

Sea $I =< f_1, \ldots, f_s >$ un ideal de $K[x_1, \ldots, x_n]$, y $F = \{f_1, \ldots, f_s\}$, siendo $f_i \neq 0 (1 \leq i \leq s)$. De la definición 10.2.1 y de la dada para la reducción por un conjunto de polinomios ($\xrightarrow{F}$), se deduce que F es una base de Gröbner sí, y sólo si, $\forall f \in I$, existe $i \in \{1, \ldots, s\}$ tal que $hterm\ (f_i)$ divide a algún término del polinomio f; en el caso particular de la reducción por el término de cabeza sí, y sólo si, divide a $hterm\ (f)$.

En el caso anterior, la dificultad surge con los elementos de I cuyos términos no son divisibles por ninguno de los $hterm\ (f_i)$. Esto ocurre porque aunque f esté en I, es decir $f = \sum_{i=1}^{s} h_i f_i$ para algúnos $h_i \in K[x_1, \ldots, x_n]$, pueden cancelarse términos de estos productos de polinomios, vease el siguiente ejemplo.

Ejemplo 10.2.3. *Sea el ideal* $I =< x^3 - 2xy, x^2y + x - 2y^2 >$, *utilizando el orden lexicográfico en* $K[x, y]$, *tenemos que*

$$x \,.\, (x^2y + x - 2y^2) - y \,.\, (x^3 - 2xy) \;=\; x^2,$$

con lo cual $x^2 \in I$, *sin embargo* x^2 *no es divisible por* x^3 *ni por* x^2y.

De manera similar a lo que se hace en los sistemas de reescritura [5], para evitar estas eliminaciones de términos (pares críticos, ver [34]) se introducen unas combinaciones especiales (resolventes) de elementos del ideal de polinomios I, llamadas normalmente S-polinomios, que se estudian en la sección siguiente.

10.3. S-Polinomios

El cómputo de una base de Gröbner de un conjunto de polinomios no nulos de n variables está íntimamente relacionado con el de **S-polinomio**[1] $S(p, q)$ de dos polinomios $p, q \in K[x_1, \ldots, x_n]$.

De manera intuitiva el **S-polinomio** de dos polinomios p y q se obtiene con el siguiente proceso: hallamos el mínimo común múltiplo L de los términos principales de p y q, multiplicamos p y q por los factores apropiados p' y q', de modo que los términos principales de $(p' \,.\, p)$ y $(q' \,.\, q)$ sean ambos L y, con el mismo coeficiente (el 1 del cuerpo). Entonces se forma el polinomio $(p'.p - q'.q)$. Este polinomio es lo que denominaremos S-polinomio de p y q. Resaltar que el término L sólo "aparece" en la definición formal de S-polinomio.

Definición 10.3.1. *Sean* $vpol \neq p, q \in K[x_1, \ldots, x_n]$. *Entonces se define el*

[1] Es una abreviación de "syzygy polynomial".

S-polinomio *de p y q, como*

$$S(p,q) =_p p \cdot_M \left(\frac{1}{hcoef\ (p)}, \frac{L}{hterm\ (p)}\right) - q \cdot_M \left(\frac{1}{hcoef\ (q)}, \frac{L}{hterm\ (q)}\right)$$

donde $L = mcm[hterm\ (p), hterm\ (q)]$. En el caso que p o q sean el polinomio vpol el S-polinomio resultante es también vpol.

Traducción en **Coq**:

```
Definition S_pol :=[p,q:pol] Cases p q of
     vpol x   =>   vpol
  |  x vpol   =>   vpol
  |  p q   =>
         (suma_app
     (mult_m p
            ((divK unK (hcoef p)),
             (div_term (lcm (hterm p) (hterm q)) (hterm p))))
     (pol_opp (mult_m q
                      ((divK unK (hcoef q)),
                       (div_term (lcm (hterm p) (hterm q)) (hterm q))))))
   end.
```

De la definición de S-polinomio se deduce que el polinomio construido $S(p,q)$ cancela el $mcm[hterm\ (p), hterm\ (q)]$, siendo este el primer término (el término más pequeño) que puede ser reducido módulo p y módulo q, aunque la reducción de $mcm[hterm\ (p), hterm\ (q)]$ puede diferir según sea el polinomio utilizado en la reducción.

Ejemplo 10.3.1. *Sean $f_1 = 2xy - y$, $f_2 = 3x^2 - y \in \mathbb{Q}[x,y]$, con el orden lexicográfico definido sobre términos por $(y <_L x)$. Entonces $L = x^2y$, $S(f_1,f_2) = f_1\frac{x^2y}{2xy} - f_2\frac{x^2y}{3x^2} = f_1\frac{1}{2}x - f_2\frac{1}{3}y = x^2y - \frac{1}{2}xy - x^2y + \frac{1}{3}y^2 = -\frac{1}{2}xy + \frac{1}{3}y^2$*

Se ve que $hterm\ (f_1 \cdot \frac{1}{2}x) = x^2y = hterm\ (f_2 \cdot \frac{1}{3}y)$, se cancela en $S(f_1,f_2)$.

Existe otra manera de considerar los S-polinomios. En la reducción del polinomio f por el conjunto $F = \{f_1, f_2, \ldots, f_s\}$, puede ocurrir que el término t de algún monomio m que figure en f sea divisible por $hterm\ (f_i)$ y $hterm\ (f_j)$ para $i \neq j$. Por lo tanto, dicho término t es divisible por $L = mcm[hterm\ (f_i), hterm\ (f_j)]$. Si reducimos f utilizando f_i, obtenemos el polinomio $h_1 = f - (\frac{m}{M(f_i)} \cdot f_i)$; en cambio si lo hacemos utilizando f_j, obtenemos $h_2 = f - (\frac{m}{M(f_j)} \cdot f_j)$. La ambigüedad que se introduce es precisamente un múltiplo del S-polinomio $S(f_i,f_j)$, más concretamente $h_2 - h_1 = (\frac{m}{M(f_i)} \cdot f_i - \frac{m}{M(f_j)} \cdot f_j) = (\frac{m}{L}) \cdot S(f_i,f_j)$.

Ejemplo 10.3.2. *Sean $f = x^2y + 1$, $f_1 = xy - y$, $f_2 = x^2 - y \in \mathbb{Q}[x,y]$, con el orden lexicográfico definido sobre términos por $(y <_L x)$. Eligiendo el monomio $m = x^2y$ tenemos que $f \xrightarrow{f_1} f - xf_1 = xy + 1$, y $f \xrightarrow{f_2} f - yf_2 = y^2 + 1$. Vemos así que $t = L = x^2y$, y que la ambigüedad introducida es $-xy + y^2 = xf_1 - yf_2 = S(f_1,f_2)$. Hay que hacer notar que $S(f_1,f_2) \in < f_1, f_2 >$, y que*

se puede reducir $S(f_1, f_2) \xrightarrow{f_1} y^2 - y$, *pero no por* f_2. *El polinomio* $y^2 - y$ *es ahora normal con respecto a* $\{f_1, f_2\}$, *pero no es nulo.*

Desafortunadamente, no siempre se elimina la "ambigüedad" con la primera construcción del S-polinomio correspondiente; puede comprobarse en este mismo ejemplo tomando $f = 5x^3y^2 + 1$.

Se introduce el concepto de S-polinomio como una manera de "cancelar" términos principales y para justificar la ambigüedad en la reducción de polinomios. Precisamente en esto va a basarse la estrategia del algoritmo de Buchberger para el cálculo de bases de Gröbner.

A partir de la formalización anterior de S-polinomio en el sistema Coq damos un resultado técnico, para simplificar pruebas posteriores, que comprueba que dicha formalización es igual sintácticamente a la expresión deseada.

Traducción en **Coq**:

```
Lemma S_pol_n_vpol: (p,q:pol)(~(equpol p vpol))->(~(equpol q vpol))->
   (S_pol p q) = (suma_app (mult_m p  ((divK unK (hcoef p)),
            (div_term (lcm (hterm p) (hterm q)) (hterm p))))
                    (pol_opp (mult_m q ((divK unK (hcoef q)),
                      (div_term (lcm (hterm p) (hterm q)) (hterm q)))))).
□
```

Al igual que hemos hecho con otras operaciones sobre polinomios se comprueba que la función *S_pol* es interna en polinomios de n variables y su extensionalidad respecto a la relación *equpol*. Las pruebas son análogas a las de las operaciones sobre polinomios.

Traducción en **Coq**:

```
Lemma full_term_S_pol: (f,g:pol)(full_term_pol f)->(full_term_pol g)->
                         (full_term_pol (S_pol f g)).


Lemma S_pol_ext: (f,x,g,y:pol)(full_term_pol f)->(full_term_pol x)->
(full_term_pol g)->(full_term_pol y)->~(equpol f vpol)->~(equpol g vpol)->
□       (equpol f x)->(equpol g y)->(equpol (S_pol f g) (S_pol x y)).
```

En el resultado que damos a continuación se resumen algunas de las propiedades de los S-polinomios que se necesitarán para la demostración del teorema de Buchberger. Se prueban por inducción sobre los polinomios dados y son immediatas a partir de las propiedades de las operaciones algebraicas de los polinomios, probadas en capítulos anteriores, y de la definición de S-polinomio.

Proposición 10.3.1. *Dados los polinomios* f *y* g *se verifica*

- $S(f, f) = vpol$
- $S(f, g) = -S(g, f)$

Traducción en **Coq**:

```
Lemma S_pol_ff: (f:pol)(equpol (S_pol f f) vpol).


Lemma S_pol_sym: (f,g:pol)(equpol (S_pol f g) (pol_opp (S_pol g f))).
□
```

10.4. Bases de Gröbner. Caracterizaciones alternativas

Comprobaremos que, de igual manera que en el caso de los "sistemas de reescritura", la confluencia de polinomios puede decidirse considerando un número finito de "situaciones críticas", formalizadas anteriormente como S-polinomios.

10.4.1. Caracterización por la confluencia

En primer lugar definimos y formalizamos la noción de confluencia[2] para la relación $\xrightarrow[F]{}$. Se utilizará esta definición para dar una caracterización de las bases de Gröbner.

Definición 10.4.1. *Dado un conjunto F de polinomios no nulos de n variables, la relación $\xrightarrow[F]{}$ se dice* **confluente** *si, y sólo si, verifica la propiedad del* **Diamante**, *es decir:*

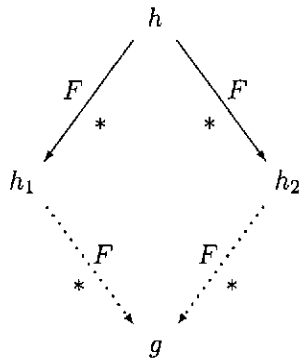


Figura 10.1: Propiedad de Confluencia

$$\forall h, h_1, h_2;\ (h \xrightarrow[*]{F} h_1) \wedge (h \xrightarrow[*]{F} h_2) \Rightarrow \exists g;\ (h_1 \xrightarrow[*]{F} g) \wedge (h_2 \xrightarrow[*]{F} g)$$

A esta propiedad la denotamos por G3CR.

Traducción en **Coq**:

```
Inductive common_succ_full [F:(list pol_full);g,h:pol_full]:Prop:=
  common_succ_red_full: (p:pol_full)(Red3 (inc_list F) (inc g) (inc p))->
     (Red3 (inc_list F) (inc h) (inc p))->(common_succ_full F g h).
```

[2]La noción de confluencia para $\longrightarrow$ significa que el resultado de una computación (reducción por cualquier término posible) es independiente de la estrategia de evaluación.

```
Definition confluente_full :=
   [F:(list pol_full)][f:pol_full](g:pol_full)(h:pol_full)
      (Red3 (inc_list F) (inc f) (inc g))->
         (Red3 (inc_list F) (inc f) (inc h))->(common_succ_full F g h).


Definition Groebner3_CR_full:= [F:(list pol_full)]
            ((full_fam (inc_list F))/\(h:pol_full)(confluente_full F h)).
```

Nota: La noción de confluencia se define a partir de la definición (9.3.3) de sucesor común de dos polinomios f y g ($f \downarrow_*^F g$). Como vamos a trabajar con polinomios canónicos, tipo *pol_full*, necesitamos extender la formalización de *common_succ* a dicho tipo.

Definición 10.4.2. *Dado un conjunto F de polinomios no nulos de n variables, la relación* $\xrightarrow[*]{F}$ *se dice* **localmente confluente** *si, y sólo si, verifica que:*

$$\forall f,g,h;\ (f \xrightarrow{F} g) \wedge (f \xrightarrow{F} h) \Rightarrow \exists z;\ (g \xrightarrow[*]{F} z) \wedge (h \xrightarrow[*]{F} z)$$

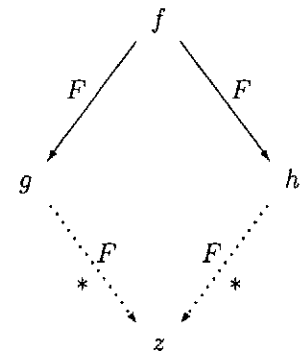


Figura 10.2: Confluencia local

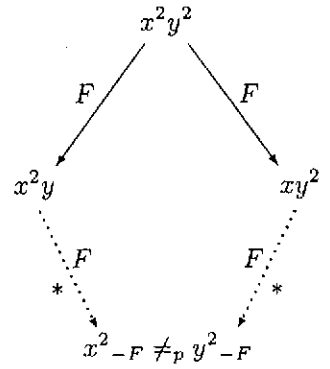
A partir de los lemas de extensionalidad de la reducción por un conjunto de polinomios $\xrightarrow[*]{F}$ (pág 177), obtenemos la extensionalidad respecto a la relación *equpol* de $\downarrow_*^F$. De la misma forma, a partir de los constructores de *Red*3 y del predicado anterior *common_succ_full*, restringimos las propiedades 1 y 2 del lema (9.3.7) a polinomios canónicos.

Traducción en **Coq**:

```
Lemma comp_common_succ_full: (p,q,r,s:pol_full)(F:(list pol_full))
        (equpol (inc p) (inc q))->(equpol (inc r) (inc s))->
           (common_succ_full F p r)->(common_succ_full F q s).


Lemma trans_common_red_full:(f,g,h:pol_full)(F:(list pol_full))
```

```
        (Red1 (inc_list F) (inc h) (inc f))->(common_succ_full F g f)->
           (common_succ_full F g h).


Lemma conmt_common_succ_full: (p,q:pol_full)(F:(list pol_full))
        (common_succ_full F p q)->(common_succ_full F q p).
□
```

Ejemplo 10.4.1. *Sea* $F = \{f_1 = x^2y - x^2, f_2 = xy^2 - y^2\}$, *con el orden lexicográfico definido sobre términos por* $(y <_L x)$. *Se verifica que:*



Esto muestra que, en general, para un conjunto F *de polinomios cualesquiera, la reducción* $\xrightarrow{F}$ *no es localmente confluente, ni, por supuesto, confluente. Será confluente, precisamente, cuando* F *sea una base de Gröbner.*

10.4.2. Caracterización por S-polinomios

Comenzamos utilizando los S-polinomios para dar otra caracterización de las bases de Gröbner.

Definición 10.4.3. *Un conjunto* F *de polinomios no nulos de* n *variables, verifica la propiedad* $(G2)$ *si:*

$$(G2) \quad \forall f, g \in F \implies (S(f,g) \xrightarrow[*]{F} vpol).$$

En otras palabras, todos los S-polinomios de elementos del conjunto F *se reducen a cero módulo* F.

Traducción en **Coq**:

```
Definition Groebner2_full:= [F:(list pol_full)] ((full_fam (inc_list F))/\
   (f,g:pol_full)(pol_In_ensemb (inc f) (inc_list F))->
      (pol_In_ensemb (inc g) (inc_list F))->
         (Red3 (inc_list F) (S_pol (inc f) (inc g)) vpol)).
```

Ejemplo 10.4.2. *Por lo visto en el ejemplo (10.3.2) y, utilizando la definición anterior, podemos decir que el conjunto de polinomios* $\{f_1 = xy - y,\ f_2 = x^2 - y\}$, *con el orden lexicográfico definido sobre términos por* $(y <_L x)$, *no verifica la propiedad anterior puesto que* $S(f_1, f_2) = -xy + y^2$, $S(f_1, f_2) \xrightarrow{f_1} y^2 - y$; *ahora este polinomio está en forma normal con respecto al conjunto* $\{f_1, f_2\}$ *y, sin embargo no es el polinomio cero.*

Los algoritmos para la construcción de bases de Gröbner que generen el mismo ideal se basan en la propiedad anterior $(G2)$.

Queremos probar que la propiedad $(G2)$ es equivalente a la condición dada en la definición de base de Gröbner (ver definición 10.2.1). La prueba de que la definición $(G1)$ implica la propiedad $(G2)$ es directa, ya que si $f, g \in F$, entonces por la propia definición de S-polinomio, se verifica que $S(f,g) \in\!< F >$. Así, utilizando $(G1)$ se obtiene la meta $S(f,g) \xrightarrow[*]{F} vpol$. En la prueba se utilizan resultados sobre polinomios canónicos y propiedades que cumplen los elementos de un ideal.

Teorema 10.4.1.

$$(G1)[\forall f \in\!< F > \Rightarrow\ (f \xrightarrow[*]{F} vpol)] \Rightarrow\ [\forall f, g \in F\ \Rightarrow\ (S(f,g) \xrightarrow[*]{F} vpol)](G2)$$

Traducción en **Coq**:

```
Lemma G1_impl_G2_full: (G:(list pol_full))(Groebner1_full G)->
                       (Groebner2_full G).
```

□

Para probar la implicación inversa $(G2) \Rightarrow (G1)$, utilizamos algunos resultados intermedios y, para ello, demostraremos, paso a paso, su equivalencia. Algunas de estas implicaciones son resultados generales que, nosotros restringimos al contexto de nuestra formalización de polinomios (por ejemplo el *lema de Newman*).

10.4.3. Caracterización por la forma normal

Podríamos haber escogido como definición de base de Gröbner, la propiedad de unicidad de la forma normal. Demostraremos que, dado un conjunto F de polinomios no nulos de n variables, la confluencia de la relación $\xrightarrow[*]{F}$ implica la unicidad de la forma normal, es decir:

$$(G3)\quad \forall h, h_1, h_2;\quad h_{1\_F} \wedge h_{2\_F} \wedge (h \xrightarrow[*]{F} h_1) \wedge (h \xrightarrow[*]{F} h_2) \Rightarrow h_1 =_p h_2$$

donde h_F significa que h es normal respecto a F.

Traducción en **Coq**:

```
Definition Groebner3_full:= [F:(list pol_full)]((full_fam (inc_list F))/\
(h,h1,h2:pol_full)(normal (inc_list F) (inc h1))->
   (normal (inc_list F) (inc h2))->
      (Red3 (inc_list F) (inc h) (inc h1))->
       (Red3 (inc_list F) (inc h) (inc h2))->(equpol (inc h1) (inc h2))).
```

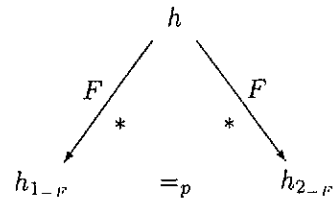


Figura 10.3: Unicidad de la forma normal

Teorema 10.4.2.

$$(G3CR) \quad \Rightarrow \quad (G3)$$

Prueba: Supuesto que $(h \xrightarrow[*]{G} h_1)$ y $(h \xrightarrow[*]{G} h_2)$, para h_1 y h_2 en forma normal respecto a G, la hipótesis de confluencia permite deducir que existe un polinomio q tal que $(h_1 \xrightarrow[*]{G} q)$ y $(h_2 \xrightarrow[*]{G} q)$. Directamente del teorema (9.5.1), se obtiene que $h_1 =_p q =_p h_2$.

Traducción en **Coq**:

```
Lemma G3_CR_impl_G3_full: (G:(list pol_full))(Groebner3_CR_full G)->
                          (Groebner3_full G).
```

□

10.5. Equivalencias entre las caracterizaciones

Veamos ahora las demostraciones que nos conducen a probar la **equivalencia** de tres de las definiciones dadas anteriormente.

Teorema 10.5.1.

$$(G3CR) \quad \Rightarrow \quad (G1)$$

Antes de probar este resultado, necesitamos una formulación intermedia, que consiste en una definición y un lema preliminar.

Definición 10.5.1. *Una reducción* $\xrightarrow{F}$ *se dice que verifica la propiedad de* **Church-Rosser**[3] *si (Figura 10.4)*

$$f \xleftrightarrow[*]{F} g \Rightarrow f \downarrow_*^F g$$

Lema 10.5.1. *Sea F un conjunto de polinomios canónicos no nulos de n variables que verifican la condición $(G3CR)$ y sean $h_1, h_2 \in K_c[X]$ tal que $h_1 \xleftrightarrow[*]{F} h_2$. Entonces, se tiene que:*

$$\exists g \in K_c[X];\ (h_1 \xrightarrow[*]{F} g) \wedge (h_2 \xrightarrow[*]{F} g)$$

[3]Alonzo Church y J. Barkley Rosser probaron que el λ−cálculo tiene esta propiedad.

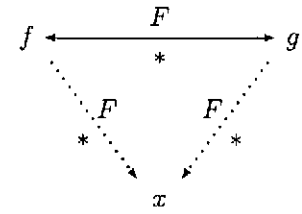
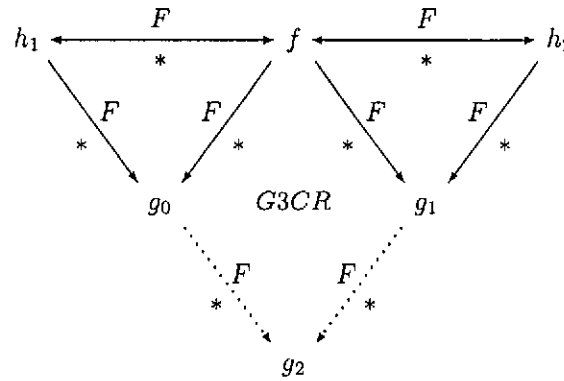


Figura 10.4: Propiedad de Church-Rosser

Prueba: Se razona por recurrencia sobre la hipótesis $h_1 \overset{F}{\underset{*}{\longleftrightarrow}} h_2$. Esto genera cuatro submetas (una por cada constructor de *Red_equiv*, pág 183). En el primer caso, $(h_1 =_p h_2)$, el polinomio común g al cual se reducen, puede ser cualquiera de ellos. Para el caso $\overset{F}{\longrightarrow}$, el polinomio cuya existencia se predica es aquel, de entre h_1 y h_2, el cual es final en un paso de reducción. En el tercer caso, la simetría de la definición *Red_equiv* nos permite afirmar que el polinomio al que se reducen h_1 y h_2 es el mismo al que se reducen h_2 y h_1. En el caso del cuarto constructor, *Red_equiv_trans*, obtenemos las hipótesis de recurrencia $h_1 \overset{F}{\underset{*}{\longleftrightarrow}} f$, con g_0 su sucesor común de f y h_1, y $f \overset{F}{\underset{*}{\longleftrightarrow}} h_2$, siendo g_1 el sucesor común de f y h_2. La propiedad de *confluencia* aplicada a f, g_0 y g_1 garantiza la existencia de g_2, al que se reducen h_1 y h_2, como puede verse en el siguiente diagrama.



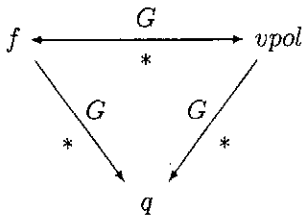
Traducción en **Coq**:

```
Lemma cr_full: (G:(list pol_full))(h1,h2:pol_full)
   (Groebner3_CR_full G)->(Red_equiv (inc_list G) (inc h1) (inc h2))->
      (Ex2 [g:pol_full](Red3 (inc_list G) (inc h1) (inc g))
           [g:pol_full](Red3 (inc_list G) (inc h2) (inc g))).
```

□

Ahora ya tenemos todas las condiciones requeridas para poder demostrar el resultado $(G3CR) \Rightarrow (G1)$ (teorema 10.5.1).

Prueba: Al explicitar las definiciones $G3CR$ y $G1$ la meta es $f \xrightarrow[*]{G} vpol$. De la hipótesis $f \in < G >$, y, de la formalización de la congruencia módulo G, se deduce que $f \equiv_G vpol$; a su vez, mediante el teorema (9.3.4) del capítulo anterior, tenemos que $f \xleftarrow[*]{G} vpol$. Aplicando el lema anterior, con los parámetros G,f y $vpol$, se obtiene que:



Como $vpol$ está en forma normal módulo G, se deduce la meta en curso, es decir $f \xrightarrow[*]{G} q =_p vpol$, por la extensionalidad de $\xrightarrow[*]{G}$ respecto a la relación $equpol$ (pág 177) y por el teorema (9.5.1).

Traducción en **Coq**:

```
Lemma G3_CR_impl_G1_full: (G:(list pol_full))(Groebner3_CR_full G)->
                          (Groebner1_full G).
```

□

Para probar que $(G2) \Rightarrow (G3CR)$ se procede en tres etapas.

- Demostramos un teorema intermedio más débil, utilizando la **confluencia local**, en lugar de la confluencia.
- Como hemos probado que la reducción $\xrightarrow{F}$ es noetheriana, podemos obtener el lema de Newman: *la confluencia local implica la confluencia.*
- Se concluye componiendo los dos resultados anteriores.

Nota: Como estamos trabajando con polinomios canónicos ($K_c[X]$) necesitamos restringir el lema (9.7.1) y el teorema (9.7.3) a dichos polinomios. Para ello utilizamos el teorema (6.6.9), la extensionalidad de $Red3$ mediante $equpol$ y la construcción de un polinomio canónico a partir de un tipo pol que verifica el predicado $full\_pol$.

Traducción en **Coq**:

```
Lemma basic3_full: (h,f,g:pol)(F:(list pol_full))(Red1 (inc_list F) f g)->
        (full_term_pol h)->(x,y:pol_full)(equpol (inc x) (suma_app f h))->
           (equpol (inc y) (suma_app g h))->(common_succ_full F x y).


Lemma basic4_full: (f,g:pol_full)(F:(list pol_full))
        (Red3 (inc_list F) (suma_app (inc f) (pol_opp (inc g))) vpol)->
           (common_succ_full F f g).
```

□

Teorema 10.5.2. *Sea F un conjunto de polinomios canónicos no nulos de n variables que verifican la propiedad (G2) y sean $f, g, h \in K_c[X]$. Se cumple que si $(f \xrightarrow{F} g)$ y $(f \xrightarrow{F} h)$, entonces $g \downarrow_*^F h$ (confluencia local).*

Prueba: Al explicitar las reducciones $(f \xrightarrow{F} g)$ y $(f \xrightarrow{F} h)$, se obtiene la existencia de los términos $t, t_0 \in f$ y los polinomios $f_i, f_j \in F$ que satisfacen las igualdades polinómicas siguientes: $g =_p f -_p f_i \cdot_M \left(\frac{(coef\ t\ f)}{hcoef\ f_i}, \frac{t}{hterm\ f_i}\right)$ y $h =_p f -_p f_j \cdot_M \left(\frac{(coef\ t_0\ f)}{hcoef\ f_j}, \frac{t_0}{hterm\ f_j}\right)$.

Utilizando que el orden lexicográfico de términos $<_L$ es total (teorema 3.4.7), distinguimos tres casos.

1. $t_0 <_L t$

   A partir de la hipótesis $t \in f$, mediante el corolario (6.5.1) descomponemos el polinomio f en tres sumandos $f =_p f_1 +_p ((coef\ t\ f), t) +_p f_2$, siendo $f_1 =_p \sum_{t' \in f \wedge t <_L t'} ((coef\ t'\ f), t')$ y $f_2 =_p \sum_{t' \in f \wedge t' <_L t} ((coef\ t'\ f), t')$. De las hipótesis del contexto y, aplicando resultados de capítulos anteriores, se obtienen los siguientes resultados, que introducimos como hipótesis en el contexto mediante la táctica **Cut**.

   i) $((coef\ t\ f), t) \xrightarrow{f_i;\, t} ((coef\ t\ f), t) -_p f_i \cdot_M \left(\frac{(coef\ t\ f)}{hcoef\ f_i}, \frac{t}{hterm\ f_i}\right)$. Esta reducción se consigue a partir de la formalización de la reducción (red), $t \in ((coef\ t\ f), t)$, y, con la ayuda de las propiedades probadas sobre las operaciones algebraicas de los polinomios.

   ii) Como $t_0 <_L t$ y $t_0 \in f$ entonces $t_0 \in f_2$. De este modo, vemos que la reducción $f_2 \xrightarrow{f_j;\, t_0} f_2 -_p f_j \cdot_M \left(\frac{(coef\ t_0\ f_2)}{hcoef\ f_j}, \frac{t_0}{hterm\ f_j}\right)$ es posible, debido a que sabemos por hipótesis que $((hterm\ f_j) \mid t_0)$.

   iii) Se obtiene $h \xrightarrow{f_i;\, t} h -_p f_i \cdot_M \left(\frac{(coef\ t\ f)}{hcoef\ f_i}, \frac{t}{hterm\ f_i}\right)$ al aplicar sucesivamente los teoremas (9.7.1 y 9.7.2) a la hipótesis obtenida en i).

   iv) De la descomposición del polinomio f, se deduce la igualdad de coeficientes $(coef\ t_0\ f_2) =_K (coef\ t_0\ f)$. A partir del paso de reducción obtenido en ii) y, utilizando el lema (9.7.1) de semicompatibilidad de la reducción respecto de la adición de polinomios canónicos, obtenemos un polinomio que verifica la condición de ser el sucesor común $(\downarrow_*^F)$ de g y $\left[h -_p f_i \cdot_M \left(\frac{(coef\ t\ f)}{hcoef\ f_i}, \frac{t}{hterm\ f_i}\right)\right]$. Para ello añadimos en ambos lados de dicho paso de reducción el polinomio $f_1 +_p ((coef\ t\ f), t) -_p f_i \cdot_M \left(\frac{(coef\ t\ f)}{hcoef\ f_i}, \frac{t}{hterm\ f_i}\right)$, y realizamos las simplificaciones polinómicas oportunas.

   Por consiguiente, de los apartados *iii)* y *iv)* y la propiedad 2 del lema (9.3.7), sobre polinomios canónicos, se resuelve directamente la meta inicial $g \downarrow_*^F h$. La figura (10.5) ilustra el caso.

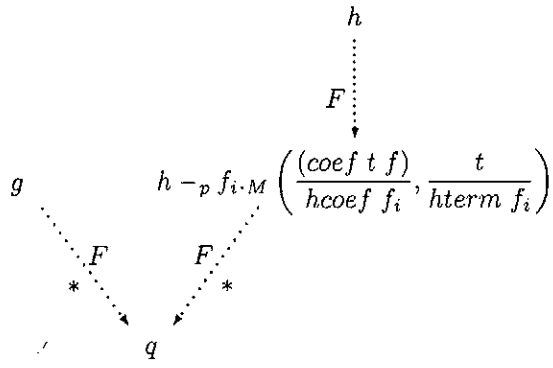


Figura 10.5: Caso $t_0 <_L t$

2. $t <_L t_0$

   Es totalmente simétrica a la prueba del caso anterior, obteniéndose de la misma manera que $h \downarrow_*^F g$ (ver la figura 10.6).

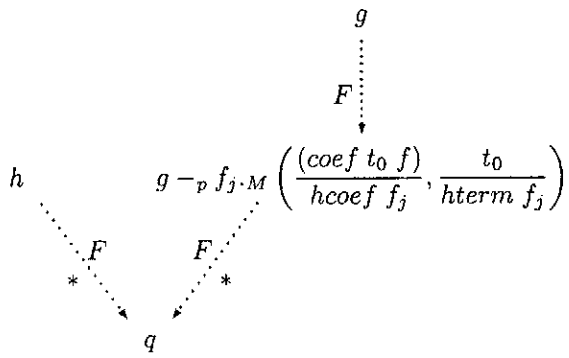


Figura 10.6: Caso $t <_L t_0$

3. $t_0 =_L t$

   Generalizamos lo que hemos visto en el ejemplo (10.3.2). La ambigüedad que se tiene al reducir un polinomio f por $f_i, f_j \in F$ (cuando un término de f es divisible por $hterm\ (f_i)$ y también lo es por $hterm\ (f_j)$ para $i \neq j$) se elimina utilizando los S-polinomios.

   De $t_0 =_L t$ podemos obtener por reescritura que $hterm\ (f_i)$ y $hterm\ (f_j)$ dividen a t. Directamente del teorema (3.6.5), deducimos que su mínimo común múltiplo $t_m = mcm[hterm\ (f_i), hterm\ (f_j)$ también divide a t. Por consiguiente, existe un término t' tal que $t = t_m.t'$. Si t'_i, t'_j denotan los términos tal que $(hterm\ (f_i)).t'_i = t_m = (hterm\ (f_j)).t'_j$,

entonces utilizando las igualdades de polinomios, obtenemos $S(f_i, f_j) =_p \left(\frac{1}{hcoef\ (f_i)}, t'_i\right) \cdot_M f_i - _p \left(\frac{1}{hcoef\ (f_j)}, t'_j\right) \cdot_M f_j$. Cabe resaltar que, para probar la igualdad polinómica anterior, debemos demostrar las siguientes igualdades de términos $(hterm\ (f_i)).t_i = t = t_m.t' = (hterm\ (f_i)).t'_i.t'$. Ello implica que $t_i = t'_i.t'$, obteniendo t_i de la condición de divisibilidad de t por $hterm\ (f_i)$. La igualdad $t_j = t'_j.t'$ se obtiene de manera similar.

Combinando la igualdad polinómica anterior con las ya obtenidas en el contexto de hipótesis para g y h de la reducción de f por f_i y f_j, respectivamente, probamos que

$$(g -_p h) =_p S(f_i, f_j).{}_M \left((coef\ t\ f), \frac{t}{mcm[hterm\ (f_i), hterm\ (f_j)]}\right)$$

La prueba se finaliza utilizando hipótesis y resultados demostrados anteriormente para polinomios canónicos, reproduciendo en **Coq** el siguiente esquema.

$$\left.\begin{array}{c} f_i, f_j \in F \\ (G2) \end{array}\right\} \Rightarrow \left.\begin{array}{c} S(f_i, f_j) \xrightarrow[*]{F} vpol \\ \text{corolario (9.2.1) } (Red3\_vpol\_mult\_m) \end{array}\right\} \Rightarrow$$

$$\left.\begin{array}{c} S(f_i, f_j) \cdot_M \left[(coef\ t\ f), \frac{t}{mcm[hterm\ (f_i), hterm\ (f_j)]}\right] \xrightarrow[*]{F} vpol \\ \text{extensionalidad de la reducción y la igualdad obtenida arriba} \end{array}\right\} \Rightarrow$$

$$\left.\begin{array}{c} (g -_p h) \xrightarrow[*]{F} vpol \\ \text{teorema (9.7.3)} \quad \text{(basic4)} \end{array}\right\} \Rightarrow g \downarrow^F_* h$$

Traducción en **Coq**:

```
Lemma G2_impl_G3_LCR_full: (F:(list pol_full))(Groebner2_full F)->
   ((f,g,h:pol_full)(Red1 (inc_list F) (inc f) (inc g))->
□     (Red1 (inc_list F) (inc f) (inc h))->(common_succ_full F g h)).
```

Para obtener ahora la prueba de que $(G2) \Rightarrow (G3CR)$, es suficiente demostrar la equivalencia entre la confluencia y la confluencia local. Es inmediato demostrar que la confluencia de $\xrightarrow{F}$ implica la **confluencia local** de dicha reducción.

Lema 10.5.2. *Sea F un conjunto de polinomios no nulos de n variables que verifican la propiedad $(G3CR)$ y sean $f, g, h \in K[x_1, \ldots, x_n]$. Se cumple que si $(f \xrightarrow{F} g)$ y $(f \xrightarrow{F} h)$, entonces $g \downarrow^F_* h$ (confluencia local).*

Traducción en **Coq**:

```
Lemma G3CR_impl_G3CR_L: (F:(list pol))(Groebner3_CR_full F)->
□                ((f,g,h:pol)(Red1 F f g)->(Red1 F f h)->(common_succ F g h)).
```

10.5.1. Lema de Newman

En la prueba de la implicación inversa del resultado anterior, que llamaremos **lema de Newman** [91] (Newman 1942), se necesita la noetherianidad de $\xrightarrow{F}$ (ver ejemplos en [5]), descrita en términos de la relación de accesibilidad y probada en el teorema (9.4.4). La prueba del lema de Newman es una prueba ya clásica, tanto en matemáticas como en sistemas de reescritura. Adaptamos la prueba hecha en ([5], [13], [106], [68], [99]) a nuestra implementación. Un guión de la prueba en **Coq**, generalizada a un relación cualquiera puede verse en [13].

Comenzamos probando resultados parciales sobre la fractura de la clausura reflexiva transitiva de la reducción en la reducción en un solo paso ($\xrightarrow{F}$).

Lema 10.5.3. *Sean F un conjunto de polinomios no nulos de n variables y polinomios $g, h \in K[x_1, \ldots, x_n]$, con $g \neq_p h$. Se verifica que:*

$$g \xrightarrow[*]{F} h \Rightarrow \exists f \in K_c[X]; \quad g \xrightarrow{F} f \xrightarrow[*]{F} h$$

Prueba: Por recurrencia sobre la definición de *Red3* se obtienen tres subcasos, que se corresponden a sus constructores.

- Si $g =_p h$, se contradice la hipótesis $g \neq_p h$.
- Cuando $g \xrightarrow{F} h$, es suficiente elegir como polinomio de fractura f, al polinomio h.
- Caso $g \xrightarrow[*]{F} f_0 \xrightarrow[*]{F} h$. Tenemos dos nuevos casos, comparando g y f_0:
  - Si $g =_p f_0$, entonces $f_0 \neq_p h$. Adaptando la hipótesis de recurrencia a $f_0 \xrightarrow[*]{F} h$, obtenemos un polinomio q que verifica $g \xrightarrow{F} q \xrightarrow[*]{F} f_0$. Es fácil ver que el polinomio q cumple la meta.
  - Si $g \neq_p f_0$ adaptando la hipótesis de recurrencia a $g \xrightarrow[*]{F} f_0$, obtenemos un polinomio q que verifica $g \xrightarrow{F} q \xrightarrow[*]{F} f_0$. El polinomio buscado es precisamente q, sin más que aplicar la transitividad de *Red3*.

Traducción en **Coq**:

```
Lemma Red3_nuevo_Red1_bis_full: (F:(list pol))(g,h:pol)(Red3 F g h)->
      (~(equpol g h))->(Ex [f:pol_full](Red1 F g (inc f))/\(Red3 F (inc f) h)).
```
□

Formalizamos la definición de localmente confluente para polinomios canónicos de manera más práctica, para simplificar las pruebas referentes al lema de Newman.

Traducción en **Coq**:

```
Definition local_confluente_full :=[F:(list pol_full)][f:pol_full](g:pol_full)
            (h:pol_full)(Red1 (inc_list F) (inc f) (inc g))->
               (Red1 (inc_list F) (inc f) (inc h))->(common_succ_full F g h).
```

En las próximas demostraciones necesitamos otro esquema de recurrencia: el principio de **inducción bien fundada**[4] sobre polinomios canónicos en Coq, expresado por la siguiente regla de inferencia:

$$\frac{\forall g \in K_c[X],\ (\forall h \in K_c[X],\ g \xrightarrow{F} h \Rightarrow P(h)) \Rightarrow P(g)}{\forall g \in K_c[X],\ P(g)} \quad \textbf{(WFIR)}$$

donde F es un conjunto de polinomios canónicos y P es una propiedad definida sobre dichos polinomios.

En otras palabras: si se desea probar $P(g)$ para todo polinomio g, es suficiente probar $P(g)$ bajo la hipótesis de que $P(h)$ es cierto para todos los polinomios h que se obtienen como sucesores de g mediante la $\xrightarrow{F}$.

Hemos puesto el esquema de inducción sin hacer explícito el "caso base", debido a que la premisa de **WFIR** incluye dicho caso. Como $\xrightarrow{F}$ termina (no hay una cadena descendente infinita de polinomios, teorema 9.4.4), el "caso base" de la inducción consiste en probar que $P(g)$ es cierto para todos los polinomios sin sucesor, es decir en **forma normal**. Por consiguiente, la hipótesis $(\forall h \in K_c[X],\ g \xrightarrow{F} h \Rightarrow P(h))$ es trivialmente verdad y la premisa de **WFIR** degenera a $P(g)$, como era de esperar.

Traducción en **Coq**:

```
Definition noether_Red1_full :=
             [F:(list pol_full)](x:pol_full)(P:pol_full->Prop)
((y:pol_full)((z:pol_full)(Red1 (inc_list F) (inc y) (inc z))->(P z))->(P y))->
               (P x).
```

WFIR no es cierto para una reducción cualquiera, pero sí lo es, para una reducción noetheriana como se prueba a continuación.

Teorema 10.5.3. *Para cada conjunto de polinomios canónicos F, si $\xrightarrow{F}$ termina (es noetheriana), entonces se verifica* **WFIR**.

Prueba: Una vez explicitadas las definiciones: *noetheriano*, *well_founded* y *noether_Red1_full*, la prueba se obtiene directamente de la aplicación del esquema de inducción bien fundada a la propiedad de accesibilidad.

Traducción en **Coq**:

```
Lemma noeth_noether2: (F:(list pol_full))(noetheriano ? (inc_Red1 F))->
   (noether_Red1_full F).
```

□

Probamos el inverso del teorema anterior como una aplicación del esquema de inducción **WFIR**.

Teorema 10.5.4. *Para cada conjunto de polinomios canónicos F, si se verifica* **WFIR**, *entonces $\xrightarrow{F}$ termina (es notheriana).*

[4] Es una generalización de la inducción de $(\mathbb{N}, >)$ a cualquier sistema de reducción que no tenga una cadena descendente infinita $(K_c[X], \xrightarrow{F})$.

Prueba: Tenemos que probar **WFIR** donde $P(g) :=$ "no existe una cadena infinita comenzando en g". El paso de inducción es sencillo: si no existe una cadena infinita que comience de cualquier sucesor de g, entonces tampoco existe una cadena infinita comenzando en g. Por consiguiente $P(g)$ es cierto para cualquier polinomio g, es decir $\xrightarrow{F}$ termina.

Traducción en **Coq**:

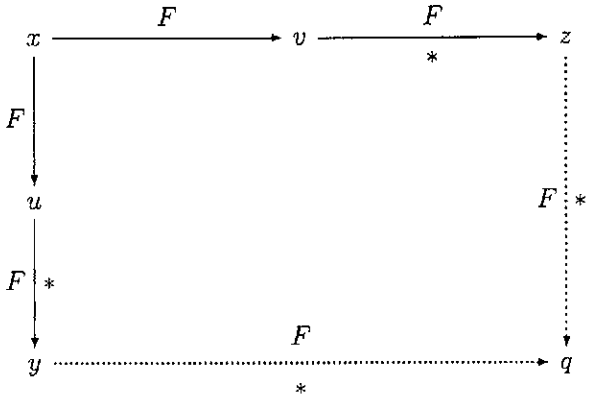
```
Lemma noeth_noether2_inv: (F:(list pol_full))(noether_Red1_full F)->
                          (noetheriano ? (inc_Red1 F)).
```

□

Para demostrar el lema de Newman demostramos algunos resultados intermedios. Para ello utilizamos el mecanismo de **Section** que implementa el sistema Coq (descrito ya en el capítulo de introducción al sistema (veáse pág 19)). Este mecanismo permite organizar la prueba estructurada en secciones. Para alcanzar dicha prueba utilizamos la prueba genérica descrita por Bruno Barras en [13].

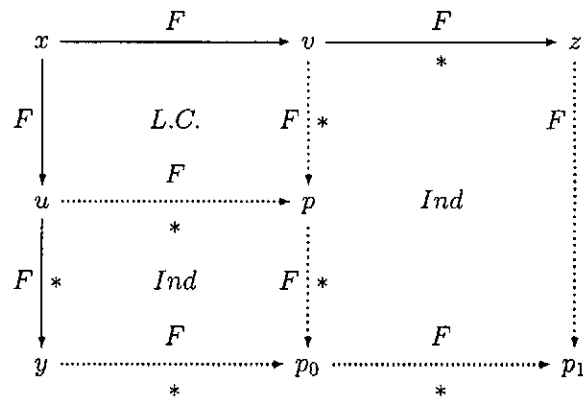
Teorema 10.5.5. *Sea F un conjunto de polinomios canónicos no nulos que verifican la confluencia local y sean $x, y, z, u, v \in K_c[X]$ se cumple que:*

$$(x \xrightarrow{F} u \xrightarrow[*]{F} y) \wedge (x \xrightarrow{F} v \xrightarrow[*]{F} z) \Rightarrow y \downarrow_*^F z$$



Prueba: Aplicando la hipótesis de confluencia local a las reducciones $(x \xrightarrow{F} u)$ y $(x \xrightarrow{F} v)$, se obtiene un polinomio p que verifica $(u \xrightarrow[*]{F} p)$ y $(v \xrightarrow[*]{F} p)$. Por la hipótesis de inducción a partir de $(x \xrightarrow{F} u \xrightarrow[*]{F} y)$ y $(x \xrightarrow{F} v \xrightarrow[*]{F} p)$, encontramos un p_0 tal que $(p \xrightarrow[*]{F} p_0)$ e $(y \xrightarrow[*]{F} p_0)$. De manera análoga, de $(x \xrightarrow{F} u \xrightarrow[*]{F} p_0)$ y $(x \xrightarrow{F} v \xrightarrow[*]{F} z)$, obtenemos un p_1 tal que $(p_0 \xrightarrow[*]{F} p_1)$ y $(z \xrightarrow[*]{F} p_1)$. Por último, a partir de los polinomios obtenidos anteriormente y aplicando la transitividad de $\xrightarrow[*]{F}$, probamos que el polinomio p_1, obtenido

como sucesor común de p_0 y de z, es el sucesor común buscado $(y \downarrow_*^F z)$, como se muestra en el siguiente diagrama.



Traducción en **Coq**:

```
Section Newman_pol.

Variable F:(list pol_full).

Hypothesis Hyp1:(noether_Red1_full F).
Hypothesis Hyp2:(x:pol_full)(local_confluente_full F x).

Section Induccion.

Variable x:pol_full.

Hypothesis hyp_ind:((u:pol_full)(Red1 (inc_list F) (inc x) (inc u))->
                    (confluente_full F u)).

Variables y,z:pol_full.

Hypothesis h1:(Red3 (inc_list F) (inc x) (inc y)).
Hypothesis h2:(Red3 (inc_list F) (inc x) (inc z)).

Section Newman_f.

Variable u:pol_full.

Hypothesis t1:(Red1 (inc_list F) (inc x) (inc u)).
Hypothesis t2:(Red3 (inc_list F) (inc u) (inc y)).


Theorem Diagrama: (v:pol_full)(u1:(Red1 (inc_list F) (inc x) (inc v)))
          (u2:(Red3 (inc_list F) (inc v) (inc z))) (common_succ_full F y z).
□
```

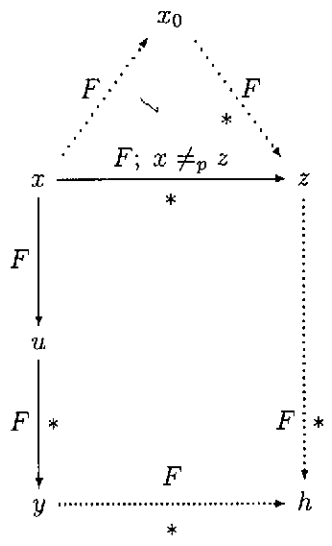
Nota: La sección abierta con el nombre de *Newman_f* la cerramos después de demostrar el siguiente lema.

Teorema 10.5.6. *Sea F un conjunto de polinomios canónicos no nulos que verifican la confluencia local y sean $x, y, z, u \in K_c[X]$. Se cumple que:*

$$(x \xrightarrow[]{F} u \xrightarrow[*]{F} y) \wedge (x \xrightarrow[*]{F} z) \Rightarrow y \downarrow_*^F z$$

Prueba: Distinguimos dos casos:

- Si $x =_p z$, el sucesor común buscado ($y \downarrow_*^F z$), es, lógicamente, y.
- En el caso ($x \neq_p z$) escindimos, mediante el lema (10.5.3), la reducción $x \xrightarrow[*]{F} z$ en $x \xrightarrow{F} x_0 \xrightarrow[*]{F} z$. Con ello, estamos en las condiciones del teorema anterior y, así, obtenemos que $y \downarrow_*^F z$, es decir, podemos construir el diagrama siguiente:



Traducción en **Coq**:

```
Theorem caseRed1xy: (common_succ_full F y z).
```

```
End Newman_f.
```
□

Ahora tenemos todas las condiciones necesarias, dentro de la sección anterior *Newman_pol*, para deducir el lema de Newman.

Teorema 10.5.7. (Lema de Newman)

Sea F un conjunto de polinomios canónicos no nulos. Dado que $\xrightarrow{F}$ es una relación noetheriana, si $\xrightarrow[]{F}$ es localmente confluente, entonces es confluente.

Prueba: Para demostrar la confluencia utilizamos la inducción bien fundada con la siguiente propiedad sobre polinomios canónicos:

$$P(x) := \quad \forall y, z \in K_c[X] \quad y \xleftarrow[*]{F} x \xrightarrow[*]{F} z \Rightarrow y \downarrow_*^F z$$

Obviamente $\xrightarrow{F}$ es confluente si el predicado $P(x)$ es cierto para todo polinomio canónico x. El esquema de recurrencia **WFIR** de la inducción bien fundada (pág 227) requiere probar $P(x)$ bajo la hipótesis de que es cierto $P(y)$ para cualquier polinomio y tal que $x \xrightarrow{F} y$, recogido en la hipótesis *hyp_ind* de la sección *Induccion*. Probamos $P(x)$ por casos:

1. Si $x =_p y$, el sucesor común de y y de z, trivialmente sería el polinomio z.
2. En el caso $x \neq_p y$, mediante el lema (10.5.3), escindimos la hipótesis $x \xrightarrow[*]{F} y$ en $x \xrightarrow{F} x_0 \xrightarrow[*]{F} y$. Con ello, estamos en las condiciones del teorema anterior (en Coq denotado por *caseRed1xy*) y, así, obtenemos la meta buscada $y \downarrow_*^F z$.

Nota: La prueba de este teorema se hace dentro de las secciones *Induccion* y *Newman_pol* utilizando como hipótesis la noetherianidad de $\xrightarrow{F}$ para simplificar y modularizar los pasos a seguir. Una vez demostrado esto y, dado que ya hemos demostrado, en nuestra formalización, la noetherianidad de $\xrightarrow{F}$ en el teorema (9.4.4), probamos el lema de Newman sin necesidad de incluir dicha noetherianidad en el contexto de hipótesis.

Traducción en **Coq**:

```
Theorem Ind_prueba: (common_succ_full F y z).


End Induccion.


Theorem Newmansi: (x:pol_full)(confluente_full F x).


End Newman_pol.


Lemma G3CR_L_impl_G3CR_full: (F:(list pol_full))(full_fam (inc_list F))->
((f,g,h:pol_full)(Red1 (inc_list F) (inc f) (inc g))->
   (Red1 (inc_list F) (inc f) (inc h))->(common_succ_full F g h))->
      (Groebner3_CR_full F).

Intros.
Red; Split.
Assumption.
Apply Newmansi; Auto.
Apply noeth_noether2; Auto.
Apply buen_ord_red1; Auto.
```

Ahora ya tenemos todas la herramientas necesarias para probar la implicación inversa del teorema (10.4.1) y, así cerrar el ciclo de la demostración del llamado **teorema de Buchberger**.

Teorema 10.5.8.

$$(G2)[\forall f,g \in F \Rightarrow (S(f,g) \xrightarrow[*]{F} vpol)] \Rightarrow [\forall f \in < F > \Rightarrow (f \xrightarrow[*]{F} vpol)](G1)$$

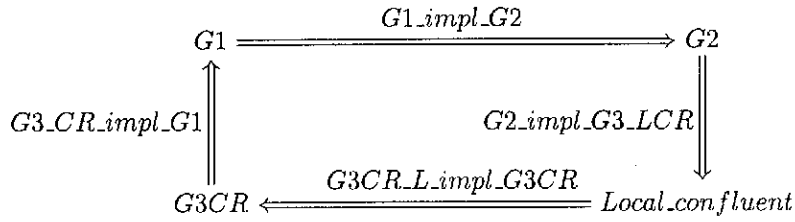
Prueba: Se prueba utilizando teoremas demostrados en este capítulo, reproduciendo en Coq el siguiente esquema,

$$(G2) \Rightarrow (Local\_confluente) \Rightarrow (G3CR) \Rightarrow (G1)$$

Traducción en **Coq**:

```
Lemma G2_impl_G1_full: (G:(list pol_full))(Groebner2_full G)->(Groebner1_full G).

Intros.
Cut (f,g,h:pol_full)(Red1 (inc_list G) (inc f) (inc g))->
   (Red1 (inc_list G) (inc f) (inc h))->(common_succ_full G g h); Intros.
Cut (Groebner3_CR_full G); Intros.
Apply G3_CR_impl_G1_full; Trivial.
Apply G3CR_L_impl_G3CR_full; Trivial.
Red in H.
Elim H; Trivial.
Elim G2_impl_G3_LCR_full with G f g h; Auto; Intros.
Split with p; Trivial.
□
```

El resultado probado ($G1 \Longleftrightarrow G2$), para polinomios canónicos corresponde al siguiente esquema:



Capítulo 11

Conclusión

En primer lugar hemos formalizado en Coq el anillo de polinomios sobre un cuerpo, dada la falta de bibliotecas en este sentido. De hecho, en [9, 10], hemos desarrollo parte de dicha formalización.

A través de esta memoria se puede confirmar que **Coq** es un sistema de ayuda a la prueba que obliga a hacer un esfuerzo hacia la formalización y especificación de pruebas, muy estricta, lo cual permite transcribirlas de modo más preciso en lenguaje matemático. A la hora de hacer balance del uso de Coq en la teoría de polinomios, resaltamos los siguiente hechos.

- Importancia del hecho de disponer de estructuras cocientes (Setoid).
- La formalización de una relación de equivalencia que refleja la igualdad de polinomios cualesquiera.
- La necesidad de formalizar cuidadosamante los polinomios canónicos por razones de eficiencia.
- Distintas caracterizaciones de la noción de reducción para su utilización en las pruebas.
- En la prueba matemática empleamos muchos lemas intermedios sin reparar en ellos. El hecho de que, en Coq es necesario probarlos, profundiza nuestro conocimiento de las teorías en que estamos trabajando.
- Se evidencia la necesidad de establecer un orden bien fundado para garantizar la terminación del algoritmo de normalización.

Es de resaltar que la extracción produce programas legibles, reutilizables y eficaces. De todo ello podemos deducir que la verificación formal de programas no triviales podría seguir una metodología análoga.

Globalmente, el balance es positivo respecto a lo esperado al principio en una prueba tan voluminosa. Se ha alcanzado el objetivo de probar un teorema puramente matemático en el cual se base el Algoritmo de Buchberger, utilizando

un sistema de ayuda a la prueba. Esto nos permite determinar los eventuales límites del sistema utilizado. En nuestro caso particular, hemos podido confirmar que este sistema de ayuda a la prueba está bien adaptado al desarrollo de pruebas de teorías matemáticas de talla y complejidad considerable.

Esperamos que la presentación, a veces bastante técnica, de nuestra formalización ayude a comprender mejor el sistema Coq, en particular el estudio de los tipos inductivos.

11.1. Aspectos prácticos

Hemos comprobado que el sistema Coq permite tratar con objetos matemáticos de manera casi natural, y la mayor parte de las veces, las pruebas formales siguen fielmente a las informales.

El grado de detalle de la implementación es bastante bueno; hay muy pocos lemas demostrados que no valga la pena ser mencionados.

Los tipos inductivos de Coq permiten un grado de confortabilidad considerable, tanto para definir estructuras de datos como para definir predicados.

Un aspecto a destacar, es que, contrariamente a la idea que teníamos inicialmente, no es absolutamente necesario tener la prueba escrita sobre papel antes de pasarla a Coq. Muchas veces es suficiente una idea más o menos precisa de como hacer la prueba. Evidentemente, tener la prueba escrita nos sirve para guiarnos en el camino hacia la demostración. Esta memoria es testigo de esto, puesto que, aunque hemos intentado seguir, fundamentalmente [4, 31, 6, 35, 15], varios resultados parciales técnicos son inéditos. Otro punto de vista técnico para el cual el sistema está especialmente adaptado es para minimizar hipótesis (en caso de ser posible).

Uno de los problemas técnicos más usuales es que los guiones de prueba son poco modulares, y muy sensibles a los cambios, salvo si se mantienen algunas precauciones.

11.2. Perspectivas

Una vez hecho el desarrollo de la prueba se tiene la sensación de que probar requiere más esfuerzo que programar. Aunque esto no es una sorpresa, indica que la perspectiva de utilizar un sistema *CAS* (Computer Algebra Systems) para certificar completamente no es realista de momento. Aún así, creemos que resultará ventajoso el desarrollo de algoritmos de álgebra computacional en sistemas donde se puede razonar acerca de ellos.

La obtención de implementaciones "certificadas" dentro de la propuesta general de obtención de bibliotecas de programas de cálculo formal certificado, parece abordable por nuestros métodos. Hay proyectos como FOC[1] [62, 22], en esta dirección. A la vista de este trabajo, se pueden seguir varios caminos.

[1]Proyecto de desarrollo de bibliotecas de Cálculo Formal certificadas en Coq, con una metodología basada en los ojetos y módulos de OCaml.

- Extender bibliotecas de programas de cálculo formal certificado para algoritmos algebraicos en CAS. Evidentemente la construcción de estas librerías debería ser hecha de manera gradual.

- Desarrollar tácticas generales de automatización, al estilo de las tácticas *Ring* y *Fourier* en Coq, para simplificaciones de determinados módulos en sistemas de ayuda a la prueba. En particular, la implementación de la mecanización de estructuras cocientes.

- Formalizar en paralelo algoritmos de sistemas CAS en Coq y PVS, al estilo de lo hecho en [21]. Ambos sistemas tienen una arquitectura similar y aportan un entorno interactivo donde el usuario puede desarrollar una teoría y enunciar propiedades expresadas en lenguajes de especificación. A pesar de las similitudes mencionadas, los dos sistemas son fundamentalmente diferentes, como ya hemos comentado en la introducción. Sería interesante un estudio detallado, tanto desde el marco lógico como desde el marco de programación, de las comparaciones de desarrollos similares en ambos sistemas, aprovechando los puntos fuertes de cada uno de ellos. Este estudio podría ayudar a definir una metodología de trabajo, que tanto se echa en falta, en los sistemas de ayuda a la prueba.

Los aplicación de los métodos formales en programación es esencial en la mayor parte del software utilizado en nuestra vida cotidiana, desde los transportes, medios de comunicación, medicina (verificación de marcapasos), ...etc, hasta los servicios bancarios (validación de tarjetas de crédito). La implementación de algoritmos debe respetar propiedades matemáticas. Las hipótesis necesarias para su aplicación, deben estar verificadas. Es decir, la implementación no debe "traicionar" a las matemáticas. Por todo ello, para obtener un software de confianza, hace falta una herramienta que permita al programador realizar pruebas sin alejarse demasiado de las matemáticas.

Bibliografía

[1] The Coq proof assistant. LogiCal proyect. Standard Libray. Documentation. http://pauillac.inria.fr/cdrom/www/coq/.

[2] Formalización de bases de Gröbner: Repositorio de código Coq. Versión 7.3.1. http://www.dc.fi.udc.es/staff/gilberto/.

[3] P. Aczel. *An Introduction to Inductive Definitions*, pages 739–782. Handbook of Mathematical Logic. North-Holland Publishing Company, 1977.

[4] W.W. Adams and P. Loustaunau. *An Introduction to Gröbner Bases.* Graduate Studies in Mathematics. American Mathematical Society, 1994.

[5] F. Baader and Tob. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[6] L. Bachmair and B. Buchberger. A simplified proof of the characterization theorem for Gröbner. *SIGSAM BULL*, (39):19–29, 1976.

[7] H.P. Barendregt. *Lambda calculi with types*, volume II of *Handbook of Logic in Computer Science.* Oxford University Press, 1993.

[8] J.M. Barja. Lógica para Informáticos. Facultad de Informática. Universidad de A Coruña, 1993.

[9] J.M. Barja and G. Pérez. Demostraciones en implementaciones concretas de Anillos de Polinomios. *RSMAE*, 2000.

[10] J.M. Barja and G. Pérez. Extracción de programas a partir de pruebas: Forma Normal de polinomios. *RSMAE*, 2002.

[11] B. Barkee. Gröbner bases: The ancient secret mystic power of the algu Compubraicus: A revelation whose simplicity will make Ladies Swoon and Grown Men Cry. Technical Report 87, Cornell University, 1988.

[12] B. Barras. Pruebas en Coq utilizando listas. Comunicación personal. INRIA (Rocquencourt).

[13] B. Barras. Coq en Coq. Technical Report 3026, INRIA, Octubre 1996.

[14] B. Barras. *Auto-validation d'un système de preuves avec familles inductives.* PhD thesis, Université de Paris 7. Denis Diderot, 1999.

[15] T. Becker and V. Weispfenning. *Gröbner Bases: A Computational Approach to Commutative Algebra.* Springer-Verlag, 1993.

[16] M.J. Beeson. *Foundations of Constructive Mathematics.* A Series of Modern Surveys in Mathematics. Springer-Verlag, 1985.

[17] J. Bertot, Y. Bertot, Y. Coscoy, H. Goguen, and F. Montagnac. *User Guide to the CtCoq Proof Environment.* INRIA, Sophia-Antipolis, 1996.

[18] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[19] Y. Bertot, G. Kahn, and L. Théry. Proof by pointing. *LNCS. Proceedings of STACS*, 1994.

[20] W. Bibel. *Automated Theorem Proving.* Vieweg, second edition, 1987.

[21] J.P. Bodeveix, M. Filali, and C. Muñoz. A formalization of the B-Method in Coq and PVS. *Electronic proceedings of the B-User group meeting at the World Congress on Formal Methods FM99*, 1999.

[22] S. Boulmé. Specifying in Coq inheritance used in Computer Algebra Libraries. Technical report, LIP6, 2000.

[23] S. Boutin. *Réflexion sur les Quotients.* PhD thesis, Université Paris 7, 1997.

[24] R. Boyer and J.S. Moore. A Lemma driven automatic theorem prover for recursive function theory. *5th International Joint Conference on Artificial Intelligence*, pages 511–519, 1977.

[25] R.S. Boyer and J.S. Moore. *The correctness problem in Computer Science.* Academic Press, 1981.

[26] R.S. Boyer and J.S. Moore. *A Computational Logic Handbook.* Academic Press, 1988.

[27] M. Bronstein, J. Grabmeier, and V. Weispfenning, editors. *Symbolic Rewriting Techniques.* Birkhauser, 1998.

[28] N. De Bruijn. *Automath a languaje for mathematics.* Les Presses de l'Université de Montréal, 1973.

[29] B. Buchberger. Ein algorithmus zum auffinden der basiselemente des restklassenringes nach einem nulldimensionalen polynomideal. *Dissertation Math. Inst. Universität Innsbruck*, 1965.

[30] B. Buchberger. Some Properties of Gröbner-Bases for Polynomial Ideals. *SIGSAM BULL*, (10):19–24, 1976.

[31] B. Buchberger. A theoretical basis for the Reduction of Polynomials to Canonical Forms. *ACM-SIGSAM BULL*, (14):29–34, 1980.

[32] B. Buchberger. *Gröbner: An algorithmic method in polynomial ideal theory*, chapter 6. Multidimensional Systems Theory. D. Reidel Publishing Company, 1985.

[33] B. Buchberger. A survey on the method of Gröbner Bases for solving problems in connection with systems or multi-variate polynomials. *Symbolic and algebraic computation by computers*, pages 69–83, 1985.

[34] B. Buchberger. History and Basic Features of the Critical-Pair, Completion Procedure. *J. Symbolic Computation*, (3):3–38, 1987.

[35] B. Buchberger. *Introduction to Gröbner Bases*, pages 35–66. Number 157 in Logic of Computation. Computer and Systems Sciences. Springer-Verlag, 1997.

[36] B. Buchberger and F. Winkler, editors. *Gröbner Bases and Applications*, volume 251 of *London Mathematical Society Lecture Notes Series*. Cambridge University Press,, Febrero 1998. Proc. of the Conference 33 Years of Gröbner Bases.

[37] V. Capretta. Certifying the Fast Fourier Transform with Coq. *LNCS*, (2152):154+, 2001.

[38] A.J. Cohn. High level proof in LCF. *Four Workshop on Automated Deduction*, pages 73–80, 1979.

[39] T. Coquand. *Une Théorie des Constructions*. PhD thesis, Université Paris 7, 1985.

[40] T. Coquand and G. Huet. Constructions: A higher order proof system for mechanizing mathematics. *EUROCAL'85*, 1985.

[41] T. Coquand and C. Paulin-Mohring. Inductively defined types. *Proceedings of Colog'88. LNCS*, (417), 1990.

[42] T. Coquand and H. Persson. Gröbner bases in type theory. *Proceedings of Calculemus and Types'98*, 1998.

[43] C. Cornes. *Conception d'un langage de haut niveau de représentation de preuves*. PhD thesis, Université Paris 7, 1997.

[44] C. Cornes and D. Terrasse. Inverting Inductive Predicates in Coq. *Types for Proofs and Programs: International Workshop TYPES'95. LNCS*, (1185), 1995.

[45] D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties and Algorithms.* Springer-Verlag, 1991.

[46] Ole-Johan Dahl. *Verificable Programming.* Prentice Hall, 1992.

[47] H. Davenport, Y. Siret, and E. Tournier. *Computer Algebra: Systems and algorithms for algebraic computation.* Academic Press, second edition, 1993.

[48] G. Dowek. *Démonstration Automatique dans le Calcul des Constructions.* PhD thesis, Université Paris 7, 1991.

[49] G. Dowek. Théorie des types. Notes de cours du DEA Programmation, 1998.

[50] M.B. Forester. Formalizing constructive real analysis. Technical Report 1382, Cornell University, 1993.

[51] J.L. Freire, J.E. Freire, A. Blanco, and V.M. Gulías. On the Strong Coinduction in Coq. *LNCS*, (2809):279–290, 2003. EUROCAST 2003.

[52] J.L. Freire, J.E. Freire, A. Blanco, and J.J. Sánchez. Fusion in Coq. *LNCS*, (2178):583–596, 2001. EUROCAST 2001.

[53] R. Fröberg. *An Introduction to Gröbner Bases.* Pure and applied mathematics. John Wiley and Sons, 1997.

[54] K.O. Geddes, S.R. Czapor, and G. Labahn. *Algorithms for Computer Algebra.* Kluwer, 1992.

[55] P. Gianni, B. Trager, and G. Zacharias. Gröbner Bases and Primary Decomposition of Polynomial Ideals. *J. Symbolic Computation*, (6):149–167, 1988.

[56] J.I. Girard. *Interpretation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris VI, 1972.

[57] J.Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Cambridge University Press, 1989.

[58] L. González-Vega and T. Recio, editors. *Algorithms in Algebraic Geometry and Applications.* Birkhauser, 1996.

[59] M.J.C. Gordon. *Programming languaje theory and its implementation.* Prentice Hall, 1988.

[60] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A theorem proving, environment for higher orde logic.* Cambridge University Press, 1993.

[61] M.J.C. Gordon, A.J. Milner, and P. Wadsworth. Edinburgh LCF: A mechanised logic of computation. *LNCS*, (78), 1979.

[62] T. Hardin. Produire un logiciel de confiance: quelles hypothèses, quelles limites? *JFLA02*, Enero 2002.

[63] H. Herbelin. Le théorème de Schroeder-Bernstein dans le Calcul des Constructions. Memoire de DEA d'Informatique, 1988.

[64] G. Hermann. Die frage der endlich vielen schritte in der theorie der polynomideale. *Math. Ann*, 95:736–788, 1926.

[65] A. Heyworth. *Applications of rewriting systems and Gröbner Bases to computing Kan extensions and identities among relations.* PhD thesis, University of Wales, 1998.

[66] D. Hirschkoff. *Mise en Oeuvre de Preuves de Bisimulation.* PhD thesis, L'École nationale des Ponts et Chaussées, 1999.

[67] W.A. Howard. The Formulae-as-type Notion of Construction. *To H.B. Curry: Essays on Combinatorial Logic, Lambda Calculus and Formalism, páginas 479-490*, 1980. Editores J.R. Hindley and J.P. Seldin.

[68] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *J.A.C.M.*, 4(27):797–821, Octubre 1980.

[69] G. Huet. Inductive principles formalized in the calculus of constructions. *Programming of Future Generation Computers*, 1988.

[70] G. Huet. The Gallina Specification Language: A case study. *LNCS*, (652):229–240, 1992.

[71] G. Huet. Residual theory in $\lambda-calculus$: A formal development. Technical Report 2009, INRIA, Agosto 1993.

[72] G. Huet, G. Kahn, and C. Paulin-Mohring. *The Coq Proof Assistant, A Tutorial.* Project Coq, INRIA Rocquencourt / CNRS-ENS Lyon, 1998.

[73] INRIA, http://pauillac.inria.fr/cdrom/www/coq/doc. *The Coq Proof Assistant. Reference Manual.*

[74] D. Kapur and J. L. Mundy, editors. *Geometric Reasoning.* MIT Press, 1989.

[75] M. Kaufmann, P. Manolios, and J.S. Moore, editors. *Computed-Aided Reasoning: ACL2 Cases Studies.* Kluwer Academic Publishers, 2000.

[76] H. Kirchner. Orderings in Automated Theorem Proving. *Proceedings of Symposia in Applied Mathematics*, pages 55–95, 1998.

[77] R. Lalement. *Logique. Réduction. Résolution.* Masson, 1990.

[78] M. Lecat. *Erreurs de mathématiciens des origines à nos jours.* Castaigne, 1935.

[79] Chin liang Chang and R. Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, INC, 1973.

[80] Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science.* Oxford Science Publications, 1994.

[81] Z. Luo and R. Pollack. LEGO proof development system: User's manual. Technical Report 211, Department of Computer Science, University of Edinburgh, 1992.

[82] L. Magnusson. *The implementation of ALF- a proof editor based on Martin-Löf's monomorphic type theory with explicit substitution.* PhD thesis, Chalmers University of Göteborg, 1994.

[83] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.

[84] M. Mayero. *Formalisation et automatisation de preuves en analyses réelle et numérique.* PhD thesis, Université Paris 6, 2001.

[85] W. McCune. *OTTER 3.0 Reference Manual and Guide.* ANL, 1994.

[86] I. Medina, J.A. Alonso, and F. Palomo. Automatic verification of polynomial rings fundamental properties in ACL2. *ACL2 Workshop*, 2000.

[87] B. Mishra. Notes on Gröbner Bases. *Information Sciences*, (48):219–252, 1989.

[88] B. Mishra. *Algorithmic Algebra.* Springer-Verlag, 1993.

[89] J.F. Monin. *Understanding Formal Methods.* Springer, 2003.

[90] C.A. Muñoz. Démonstration automatique dans la logique intuitionniste. Master's thesis, DEA d'Informatique Fondamentale, Université Paris 7, Septiembre 1994.

[91] M.H.A. Newman. On theories with a combinatorial definition of equivalence. *Annals of Mathematics*, (43):233–243, 1942.

[92] P. Odifredi, editor. *Logic and Computer Science.* Number 31 in APIC Studies in Data Processing. Academic Press, 1990.

[93] S. Owre, J.M. Rushby, and N. Shankar. PVS: A prototype verification system. *LNCS*, (607), 1992.

[94] C. Parent. Developing certified programmes in the system coq, the program tactic. *Types for Proofs and Programs. LNCS*, (806), 1993.

[95] C. Parent. *Synthèse de preuves de programmes dans le Calcul des Constructions Inductives.* PhD thesis, École Normale Supérieure de Lyon, 1995.

[96] C. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions.* PhD thesis, Université Paris 7, 1989.

[97] C. Paulin-Mohring. Inductive definitions in the System Coq: Rules and Properties. *Proceedings TLCA. LNCS*, (664), 1992.

[98] C. Paulin-Mohring and B. Werner. Synthesis of ml programs in the system Coq. *Journal of Symbolic Computation*, (15):607–640, 1993.

[99] L.C. Paulson. Constructing Recursion Operators in Intuitionistic Type Theory. *J. Symbolic Computation*, II(4):325–355, 1986.

[100] L.C. Paulson. *Logic and Computation. Interactive proof with Cambridge LCF.* Cambridge University Press, 1990.

[101] L.C. Paulson. The Isabelle reference manual. Technical Report 283, University of Cambridge Laboratory, 1993.

[102] H. Perdry. *Aspects constructifs de la théorie des corps valués. (précédée d'un chapitre sur la noetherianité constructive).* PhD thesis, Université de Franche-Comté, 2001.

[103] H. Persson. An Integrated Development of Buchberger's Algorithm in Coq. Technical Report 4271, INRIA, Setiembre 2001.

[104] J. Pfalzgraf and D. Wang, editors. *Automated Practical Reasoning.* Springer-Verlag, 1995.

[105] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. *IEEE, editor, Proceedings, Supercomputing '91*, pages 4–13, 1991.

[106] A. Saïbi. Formalization of a $\lambda - calculus$ with explicit substitutions in Coq. *Types for Proofs and Programs. LNCS*, (996), 1994.

[107] A. Saïbi. Théorie constructive des catégories. Technical Report 1993, INRIA, 1996.

[108] B. Schneier. *Applied Cryptography.* John Wiley Sons, Inc, second edition, 1996.

[109] J. Siekmann and G. Wrightson, editors. *Automation of Reasoning.* Springer-Verlag, 1983. A computer program for Presburger's algorithm.

[110] B. Sturmfels. *Gröbner Bases and Convex Polytopes.* American Mathematical Society, 1995.

[111] D. Terrasse. *Vers un environnement d'aide au développement de preuves en sémantique naturelle.* PhD thesis, L'École nationale des Ponts et Chaussées, 1995.

[112] L. Théry. A certified version of Buchberger's algorithm. *Proceedings of Automated Deduction, CADE-15. LNAI*, (1421):349–364, 1998.

[113] A.S. Troelstra and D. van Dalen. *Constructivism in Mathematics. An introduction I*, volume 121 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1988.

[114] A.S. Troelstra and D. van Dalen. *Constructivism in Mathematics. An introduction II*, volume 123 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1988.

[115] A. Trybulec. The Mizar/*QC*/6000 logic information language. *Bulletin ALCC Association for Literaty and Linguistic Computing*, 2(6):136–140, 1978.

[116] J. Von Zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 1999.

[117] B. Werner. *Une Théorie des Constructions Inductives*. PhD thesis, Université Paris 7, 1994.

[118] B. Werner. Sets in Types, Types in Sets. *Proceedings of TACS'97. LNCS 1281*, pages 530–546, 1999.

Apéndice A

Programas extraídos de los resultados calculatorios

En este apéndice mostramos el código **OCaml** extraído de los resultados calculatorios probados en nuestra formalización. En la primera sección describimos la forma de especificar programas en Coq.

A.1. Especificación de programas

En Coq existen tres tipos para especificar programas:

1. **(sig T [x:T]P)** denotado $\{x : T|P\}$ especifica los elementos x de tipo T que satisfacen la propiedad P. Este tipo está definido de forma existencial, pero dentro de la clase *Set*. Como Coq utiliza una lógica constructiva, los objetos de este tipo se prueban dando un elemento de T y una prueba de que dicho elemento verifica la propiedad P. El programa extraído es un objeto de tipo T que verifica la propiedad P.

   ```
   Inductive sig [A:Set; P:A->Prop]: Set :=
         exist : (x:A)(P x)->(sig A P).
   ```

   El principio de recurrencia asociado a este tipo es el siguiente:

   ```
   sig_ind =
   [A:Set; P:(A->Prop); P0:((sig A P)->Prop)](sig_rect A P P0)
        : (A:Set; P:(A->Prop); P0:((sig A P)->Prop))
           ((x:A; p:(P x))(P0 (exist A P x p)))->(s:((sig A P)))(P0 s)
   ```

   Se diferencia del tipo **ex**, utilizado para la cuantificación existencial, en que **sig** está definido en la clase **Set**, mientras que **ex** lo está en **Prop**.

2. **(sumor [x:T]P Q)** denotado $\{x : T|P\} + \{Q\}$ especifica un objeto de tipo (T *sumor*), que es la suma disjunta de T y un tipo singleton. Se

denota por **inleft** un término x de tipo T que verifica la propiedad P o por **inright** que significa que se verifica Q. Suele utilizarse para funciones de búsqueda que pueden fallar y para la descripción de funciones parciales.

```
Inductive sumor [A:Set; B:Prop]: Set :=
      inleft: A->A+{B}
    | inright: B->A+{B}.
```

Una definición equivalente sería:

```
Inductive option [A:Set; P: A->Prop; B: Prop]: Set :=
      exito : (x:A)(P x)->(option A P B)
    | fallo : B->(option A P B).
```

Un "habitante" de este tipo es un elemento x de A junto con una prueba de que x verifica el predicado P, o una prueba de B.

3. **(sumbool P Q)** denotado $\{P\} + \{Q\}$ especifica **true** si P se verifica, y **false** si Q es verdadera (si P y Q son ambas verdaderas, el resultado no se especifica y puede depender de la implementación). Para esta especificación, es suficiente dar una prueba de P o una prueba de Q. El caso particular de $\{P\} + \{\neg P\}$ especifica la decidibilidad de la propiedad P y su prueba es el algoritmo que decide P, siempre, claro está, que no tengamos axiomas calculatorios.

```
Inductive sumbool [A:Prop; B:Prop]: Set :=
      left: A->{A}+{B}
    | right: B->{A}+{B}.
```

Este tipo inductivo está particularmente adaptado para describir funciones de test, que retornan un valor booleano en la programación convencional. Al igual que el tipo **sig** es el correspondiente en la clase **Set** del tipo **ex**, el tipo **sumbool** es el correspondiente del tipo **or**.

La prueba de un lema con contenido calculatorio tiene idéntica estructura que la del algoritmo que realiza dicho lema, por lo cual nos podemos ayudar de los programas para demostrar estos lemas (ver [95]). Existe una táctica, **Program**[1], que permite asociar un programa a un lema, siendo posible entonces automatizar algunas partes de la prueba siguiendo dichos programas. Una vez que todas las informaciones contenidas en el programa se han utilizado, sólo queda probar las metas no calculatorias, que no tienen ninguna influencia en el algoritmo extraído. Las ventajas de este método son:

- las especificaciones más sencillas se prueban automáticamente mediante dicha táctica, con lo cual podemos concentrarnos en la justificación de la corrección del programa (normalmente los resultados no calculatorios restantes).
- nos aseguramos de que el programa que se extrae sigue el algoritmo dado.

[1] En fase de experimentación.

A.2. Procedimiento de extracción

Las extracciones que mostramos en este apéndice están generadas por la versión V7.3.1, del sistema **Coq**.

Se pueden ver más detalles sobre los principios teóricos de la implementación de la extracción en [98] y sobre ejemplos concretos en [95, 13].

La extracción consiste en la interpretación de una prueba por un programa ejecutable. Esta interpretación se hace en tres etapas:

- Eliminando todos los objetos no calculatorios de la prueba (los de tipo *Prop*), es decir borrando las partes puramente lógicas en la prueba. El sistema de tipos esta construido de tal manera que conserva la validez de una sentencia de tipado para esta operación: no se puede probar un resultado calculatorio por recurrencia sobre una hipótesis lógica.

- Eliminación de los tipos dependientes.

- La interpretación propiamente dicha. Se hace explotando la similitud del sistema Fw^{idt} y los lenguajes de programación funcionales, como ML o *Haskell*.

A continuación exponemos la sintaxis básica del proceso de generación de código *OCaml*. Se puede ver de forma mucho más completa y con otros enfoques en [73].

Finalizada la prueba calculatoria de un teorema, cargamos mediante la táctica **Require**, la librería **Extraction**. Una vez cargada dicha librería podemos extraer el código *OCaml*,[2] asociado a una prueba, de manera rápida de dos formas: utilizando la orden **Extraction** "nombre del teorema" se extrae únicamente al término explicitado; utilizando la orden **Recursive Extraction** "nombre del teorema" se extrae además del término explicitado, todos los elementos del entorno de los cuales depende la prueba. Otra opción que permite el sistema, es la de generar ficheros conteniendo los programas extraídos en el lenguaje elegido; para ello se utiliza la orden **Extraction** "nombre del fichero" "nombre del teorema".

```
Require Extraction.

Extraction for_norm.

Recursive Extraction for_norm.

Extraction "forma_normal" for_norm.
```

[2]Por defecto se obtiene *OCaml*, aunque podria obtenerse código Haskell utilizando la táctica *Extraction Language Haskell*.

A.3. Optimizaciones

El mecanismo de extracción de Coq permite ciertas optimizaciones automáticas para mejorar la eficiencia y legibilidad del código extraído. Estas conciernen principalmente a la manera de interpretar los tipos inductivos: con ellas pueden utilizarse los tipos nativos del lenguaje, en nuestro caso *OCaml*, en vez de los generados a partir de los tipos definidos en Coq.

Por ejemplo podemos implementar el tipo inductivo **sumbool** por el tipo **bool** de *OCaml*, esto nos permitirá utilizar **if/then/else**. También se puede implementar el tipo **list** con los constructores de *OCaml* y, el tipo **prod** por los pares de *OCaml*, como se puede ver a continuación.

```
Extract Inductive sumbool => bool [ true false ].

Extract Inductive list => list [ "[]" "::" ].

Extract Inductive prod => prod [ "" ].
```

Las traducciones que efectua el sistema en el primer ejemplo son las siguientes:

$$inleft \quad \hookrightarrow \quad true$$

$$inright \quad \hookrightarrow \quad false$$

En el segundo:

$$Nil \quad \hookrightarrow \quad []$$

$$Cons \quad \hookrightarrow \quad ::$$

Y en el tercero:

$$Pair\ (x, y) \quad \hookrightarrow \quad (x, y)$$

A.4. Código Extraído

En teoría, el codigo producido por el sistema nos asegura que obedece fielmente a la especificación hecha. Sin embargo, es interesante su lectura y análisis para realizar "a mano" las mejoras que se puedan aportar, en términos de eficacia, legibilidad e interpretación de tipos inductivos, no solamente en nuestro código, sino también en próximas versiones de Coq.

Agrupamos el código extraído, sin optimización, por conceptos y siguiendo el orden de nuestra formalización.

A.4.1. Términos

```
(*************************************************************)
(*  Decidibilidad de la igualdad de términos.                *)
(*************************************************************)

type 'a list =
  | Nil
  | Cons of 'a * 'a list

type sumbool =
  | Left
  | Right

type nat =
  | O
  | S of nat

let rec eq_nat_dec n m =
  match n with
    | O -> (match m with
              | O -> Left
              | S n0 -> Right)
    | S n0 -> (match m with
                 | O -> Right
                 | S n1 -> eq_nat_dec n0 n1)

let rec eq_tm_dec l t2 =
  match l with
    | Nil -> (match t2 with
                | Nil -> Left
                | Cons (a, l0) -> Right)
    | Cons (a, l0) ->
        (match t2 with
           | Nil -> Right
           | Cons (a0, l1) ->
               (match eq_nat_dec a a0 with
                  | Left -> eq_tm_dec l0 l1
                  | Right -> Right))

(*************************************************************)
(*    El orden lexicográfico es un orden total               *)
(*************************************************************)

type 'a sumor =
```

```
  | Inleft of 'a
  | Inright

let pred = function
  | O -> O
  | S u -> u

let rec lt_eq_lt_dec n m =
  match n with
    | O -> (match m with
              | O -> Inleft Right
              | S n0 -> Inleft Left)
    | S n0 ->
        (match m with
           | O -> Inright
           | S n1 -> lt_eq_lt_dec n0 n1)

let rec ttm_total_good l t2 n =
  match l with
    | Nil ->
        (match t2 with
           | Nil -> Inright
           | Cons (a, l0) -> Inleft Left)
    | Cons (a, l0) ->
        (match t2 with
        | Nil -> Inleft Right
        | Cons (a0, l1) ->
            (match lt_eq_lt_dec a a0 with
               | Inleft x ->
                   (match x with
                   | Left -> Inleft Left
                   | Right -> ttm_total_good l0 l1 (pred n))
               | Inright -> Inleft Right))

let ttm_total t1 t2 n =
  ttm_total_good t1 t2 n

(************************************************************)
(*        Decibilidad de la divisibilidad de términos       *)
(************************************************************)

let rec dec_term_div l t2 n =
  match l with
    | Nil -> Left
    | Cons (a, l0) ->
        (match t2 with
```

```
          | Nil -> eq_tm_dec (Cons (a, l0)) Nil
          | Cons (a0, l1) ->
              (match lt_eq_lt_dec a a0 with
                 | Inleft x -> dec_term_div l0 l1 (pred n)
                 | Inright -> Right))

(*************************************************************)
(*   Decibilidad de términos respecto al número de variables *)
(*************************************************************)

let rec length = function
  | Nil -> O
  | Cons (a, m) -> S (length m)

let full_tm_dec t =
  eq_nat_dec n (length t)
```

A.4.2. Cuerpo

En Coq es posible incluir[3] algunos axiomas en el desarrollo de una prueba. En nuestro caso, tenemos que dar una interpretación de los axiomas utilizados para formalizar la noción de cuerpo. Mediante la táctica **Extract** podemos asociar una interpretación a cada axioma, así como, también, a las variables libres que se necesitarán para fijar el número de variables de un término en un polinomio. La sintaxis es la siguiente:

```
Extract Constant multK_neg => string.

Extract Constant n => nat.


(*************************************************************)
(*  Decibilidad  de la igualdad sobre elementos de un cuerpo *)
(*************************************************************)

let multK_neg = string

let multK = string

let invK = string

let plusK = string

let decK = bool
```

[3]Realizing axioms.

```
let oK = string

let decK_t k k' =
  decK (plusK k (multK_neg k'))
```

A.4.3. Monomios

```
(*************************************************************)
(*          Decibilidad respecto al monomio cero             *)
(*************************************************************)

type ('a, 'b) prod =
  | Pair of 'a * 'b

let z_monom_dec = function
  | Pair (x, x0) -> decK x

let mon_coef = function
  | Pair (q, t) -> q

let dec_coef_monom m =
  decK (mon_coef m)
```

A.4.4. Polinomios

```
(*************************************************************)
(*Decibilidad respecto al número de variables de un polinomio*)
(*************************************************************)

type term = nat list

type monom = (Obj.t, term) prod

type pol =
  | Vpol
  | Cpol of monom * pol

let rec full_pol_tm_dec = function
  | Vpol -> Left
  | Cpol (m, p0) ->
      let Pair (x, x0) = m in
      (match full_tm_dec x0 with
```

```
        | Left -> full_pol_tm_dec p0
        | Right -> Right)
```

A.4.5. Coeficientes

```
(************************************************************)
(* Decibilidad de la pertenencia de un término a un polinomio *)
(************************************************************)

let coef_z_in_pol y t p h0 y0 =
  h0

let rec coef p t =
  match p with
    | Vpol -> oK
    | Cpol (m, p1) ->
        let Pair (c, u) = m in
        (match eq_tm_dec t u with
           | Left -> plusK c (coef p1 t)
           | Right -> coef p1 t)

let rec dec_term_in_pol = function
  | Vpol -> (fun t -> Right)
  | Cpol (m, p0) ->
      let h = dec_term_in_pol p0 in
      (fun t ->
      let Pair (x, x0) = m in
      (match decK x with
         | Left -> h t
         | Right ->
             (match eq_tm_dec t x0 with
                | Left ->
                    (match decK_t (coef p0 t) (multK_neg x) with
                       | Left -> Right
                       | Right -> Left)
                | Right -> h t)))
```

A.4.6. Polinomios canónicos

```
(************************************************************)
(*    Decibilidad de polinomios canónicos respecto a vpol    *)
(************************************************************)

let vpol_dec = function
```

```
  | Vpol -> Left
  | Cpol (m, p0) -> Right

(*************************************************************)
(*  Inserción de un monomio no nulo en un polinomio canónico *)
(*************************************************************)

let insert m x =
  let Pair (x0, x1) = m in
  let rec f = function
    | Vpol -> Cpol ((Pair (x0, x1)), Vpol)
    | Cpol (m0, p0) ->
        let Pair (x2, x3) = m0 in
        (match ttm_total x1 x3 n with
           | Inleft x4 ->
               (match x4 with
              | Left -> Cpol ((Pair (x2, x3)), (f p0))
              | Right -> Cpol ((Pair (x0, x1)),
                               (Cpol ((Pair (x2,
                     x3)), p0))))
           | Inright ->
               (match decK (plusK x0 x2) with
              | Left -> p0
              | Right -> Cpol ((Pair ((plusK x0 x2), x1)), p0)))
  in f x

(*************************************************************)
(*      Construcción de un polinomio canónico                *)
(*************************************************************)

let rec can_fun = function
  | Vpol -> Vpol
  | Cpol (m, p0) ->
      let x = can_fun p0 in
      (match z_monom_dec m with
         | Left -> x
         | Right -> insert m x)

let fun_can p =
  match full_pol_tm_dec p with
    | Left -> can_fun p
    | Right -> Vpol

(*************************************************************)
(*   Decibilidad de polinomios respecto a vpol               *)
(*************************************************************)
```

```
let vpol_dec_full_term p =
  match can_fun p with
    | Vpol -> Left
    | Cpol (m, p0) -> Right

(*************************************************************)
(*  Decibilidad de la igualdad de dos polinomios             *)
(*************************************************************)

let rec suma_app p q =
  match p with
    | Vpol -> q
    | Cpol (m1, p1) -> Cpol (m1, (suma_app p1 q))

let rec pol_opp = function
  | Vpol -> Vpol
  | Cpol (m, p0) ->
      let Pair (c, t) = m in
      Cpol ((Pair ((multK_neg c), t)), (pol_opp p0))

let equpol_dec_full_term p q =
  vpol_dec_full_term (suma_app p (pol_opp q))
```

A.4.7. Operaciones con polinomios canónicos

```
(*************************************************************)
(*      Adición de polinomios canónicos                      *)
(*************************************************************)

let rec add_effic = function
  | Vpol -> (fun q -> q)
  | Cpol (m, p0) ->
      let hRp = add_effic p0 in
      (fun q ->
      let Pair (x, x0) = m in
      let rec f = function
        | Vpol -> Cpol ((Pair (x, x0)), p0)
        | Cpol (m0, p2) ->
            let Pair (x1, x2) = m0 in
            (match ttm_total_good x0 x2 n with
               | Inleft x3 ->
                   (match x3 with
                      | Left -> Cpol ((Pair (x1, x2)), (f p2))
```

```
                    | Right -> Cpol ((Pair (x, x0)),
                        (hRp (Cpol ((Pair (x1, x2)), p2)))))
                | Inright ->
                    (match decK_t x (multK_neg x1) with
                      | Left -> hRp p2
                      | Right -> Cpol ((Pair ((plusK x x1), x0)
),
                          (hRp p2))))
      in f q)

let add_effic_fun p q =
  add_effic p q

(************************************************************)
(*    Producto de un polinomio canónico por un monomio      *)
(************************************************************)

let rec plus n0 m =
  match n0 with
    | O -> m
    | S p -> S (plus p m)

let rec term_mult t1 t2 =
 match t1 with
| Nil -> t2
| Cons (n1, t1') ->
    (match t2 with
 | Nil -> t1
 | Cons (n2, t2') -> Cons ((plus n1 n2), (term_mult t1' t2')))

let mult_mon m1 m2 =
  let Pair (k1, t1) = m1 in
  let Pair (k2, t2) = m2 in Pair
                        ((multK k1 k2), (term_mult t1 t2))

let rec mult_m_full p m =
  match p with
    | Vpol -> Vpol
    | Cpol (m0, p0) ->
        (match z_monom_dec m with
          | Left -> Vpol
          | Right ->
              add_effic (Cpol ((mult_mon m0 m), Vpol))
                (mult_m_full p0 m))

let mult_m_effic_fun p m =
```

```
  mult_m_full p m

(*************************************************************)
(*        Producto de polinomios canónicos                   *)
(*************************************************************)

let rec mult_p_can p q =
  match p with
| Vpol -> Vpol
| Cpol (m, p0) -> add_effic (mult_m_full q m) (mult_p_can p0 q)

let mult_p_effic_fun p q =
  mult_p_can p q




(*************************************************************)
(*               Tipo polinomio canónico                     *)
(*************************************************************)

type pol_full = pol sig0

let ex_fpol f =
  f

let ex_fpol_eq f =
  f

(*************************************************************)
(*  Decibilidad respecto de un conjunto de polinomios        *)
(*************************************************************)

let rec full_fam_dec = function
  | Nil -> Left
  | Cons (a, l0) ->
      (match full_fam_dec l0 with
         | Left ->
             (match full_pol_tm_dec a with
                | Left ->
                    (match vpol_dec_full_term a with
                       | Left -> Right
                       | Right -> Left)
                | Right -> Right)
         | Right -> Right)
```

A.4.8. Forma normal

```
(*************************************************************)
(*            Algoritmo de normalización                     *)
(*************************************************************)

let rec mult_m p m =
  match p with
    | Vpol -> Vpol
    | Cpol (m1, p1) -> Cpol ((mult_mon m m1), (mult_m p1 m))

let divK k1 k2 =
  multK k1 (invK k2)

let rec n_term_0 = function
  | 0 -> Nil
  | S n1 -> Cons (0, (n_term_0 n1))

let hmonom p =
  match fun_can p with
    | Vpol -> Pair (oK, (n_term_0 n))
    | Cpol (m, q) -> m

let hcoef p =
  mon_coef (hmonom p)

let rec minus n0 m =
  match n0 with
    | 0 -> 0
    | S k -> (match m with
                | 0 -> S k
                | S l -> minus k l)

let rec div_term t1 t2 =
 match t1 with
| Nil -> t2
| Cons (n1, t1') ->
    (match t2 with
  | Nil -> t1
  | Cons (n2, t2') -> Cons ((minus n1 n2), (div_term t1' t2')))

let mon_term = function
  | Pair (q, t) -> t

let hterm p =
```

```
  mon_term (hmonom p)

let rec prev_calcula_fn2 p f =
  match p with
    | Vpol -> Inright
    | Cpol (m, p0) ->
        let Pair (x, x0) = m in
        (match prev_calcula_fn2 p0 f with
           | Inleft x1 -> Inleft (Cpol ((Pair (x, x0)), x1))
           | Inright ->
               (match dec_term_div (hterm f) x0 n with
                  | Left -> Inleft
                      (suma_app (Cpol ((Pair (x, x0)), p0))
                        (pol_opp
                          (mult_m f (Pair ((divK x (hcoef f)),
                            (div_term x0 (hterm f)))))))
                  | Right -> Inright))

let inc p =
  p

let rec prev_calcula_fn3bis l f =
  match l with
    | Nil -> Inright
    | Cons (a, l0) ->
        (match prev_calcula_fn3bis l0 f with
           | Inleft x -> Inleft x
           | Inright ->
               (match prev_calcula_fn2 (inc f) (inc a) with
                  | Inleft x -> Inleft (can_fun x)
                  | Inright -> Inright))

let rec calcula_fnbis f f0 =
  match prev_calcula_fn3bis f f0 with
    | Inleft x -> calcula_fnbis f x
    | Inright -> f0

let rec inc_list = function
  | Nil -> Nil
  | Cons (p1, p2) -> Cons ((inc p1), (inc_list p2))

let for_norm f g =
  match full_fam_dec (inc_list f) with
    | Left -> calcula_fnbis f g
    | Right -> Vpol
```

```
(*************************************************************)
(*  Decibilidad respecto a la forma normal de un polinomio   *)
(*************************************************************)

let for_norm_dec_vpol f g =
  equpol_dec_full_term
    (inc
      (match full_fam_dec (inc_list f) with
         | Left -> calcula_fnbis f g
         | Right -> Vpol)) Vpol
```

Hemos axiomatizado el cuerpo K, por eso al extraer el código no se genera una definición ni del tipo ni de las funciones definidas sobre él. Como el resto del código depende de dicha axiomatización, para poder compilar en *OCaml*, sería preciso definir una interfaz donde esté declarado el tipo K y sus operaciones.

Si además, se proporciona una implementación se podrá ejecutar el código generado. Tal implementación podría consistir en utilizar directamente el módulo **Num** de *OCaml*, que proporciona una formalización de los números racionales.

Apéndice B

Módulos de utilidad general

B.1. Inducción Completa para listas

El primer módulo contiene un resultado sobre las listas que permite razonar por recurrencia sobre la longitud de una lista. Es equivalente al "Metateorema de la Inducción Completa para listas" (pág 41) utilizado en nuestra formalización. El resultado se obtiene a partir de formulaciones ya existentes en las librerías de Coq; en particular, del uso del módulo *wf_inverse_image* de la librería *Wellfounded* cuyo autor es Bruno Barras.

```
(*************************************************************)
(**                        LFCIA                            **)
(**                                                         **)
(**              Facultad de Informática                    **)
(**                                                         **)
(**              Universidade da Coruña                     **)
(**                                                         **)
(**            Inducción Completa para listas               **)
(**                                                         **)
(** Autores: Gilberto Perez Vega y Jose Luis Freire Nistal **)
(*************************************************************)

Require PolyList.
Require Wf.
Require Wf_nat.
Require Inverse_Image.

(* Un resultado general: *)

Section trans_imp.

Variables H,G,H1:Prop.
```

```
Lemma transimp:(H->H1)->(H1->G)->(H->G).

Tauto.
Qed.

End trans_imp.

Section theorems.

Variables A:Set;P:(list A)->Prop.

Definition ltl:=[l,m:(list A)](lt (length l) (length m)).
Definition long:=[l:(list A)](length l).

(*Probando propiedades de listas utilizando ltl wellfoundedness*)

Theorem list_indls:
(*----------------------------------------*)
( (x:(list A))((v:(list A))(ltl v x)->(P v))
              ->(P x) )
(*----------------------------------------*)
 ->(a:(list A))(P a).

Proof.
Apply
 (well_founded_ind (list A) ltl
   (wf_inverse_image (list A) nat lt long lt_wf)).

Qed.

(* Dos lemas auxiliares *)

Lemma lslg:
( (n:nat)((v:(list A))(lt (length v) n)->(P v))
              ->(x:(list A))(length x)=n->(P x)) ->
( (x:(list A))((v:(list A))(ltl v x)->(P v))
              ->(P x) ).

Proof.
Intro H.
Intro x.
Intro H1.
Apply (H (length x)); Auto.
Qed.

Lemma lgls:
```

```
( (x:(list A))((v:(list A))(ltl v x)->(P v))
              ->(P x) ) ->
( (n:nat)((v:(list A))(lt (length v) n)->(P v))
              ->(x:(list A))(length x)=n->(P x)).

Proof.
Intros H1 n H x H2.
Rewrite <- H2 in H.
Unfold ltl in H1.
Apply (H1 x H).
Qed.


Theorem list_indlg:
(*-----------------------------------------*)
( (n:nat)((v:(list A))(lt (length v) n)->(P v))
              ->(x:(list A))(length x)=n->(P x) )
(*-----------------------------------------*)
->(a:(list A))(P a).

Proof.
Intro H.
Cut (x:(list A))((v:(list A))(ltl v x)->(P v))->(P x);
 [ Intro H'; Apply (list_indls H') | Apply lslg; Auto ].
Qed.

End theorems.
```

B.2. Irreflexividad en una relación bien fundada

En este módulo se prueba que la propiedad irreflexiva es inherente a toda relación bien fundada.

```
(*************************************************************)
(**                        LFCIA                            **)
(**                                                         **)
(**             Facultad de Informática                     **)
(**                                                         **)
(**             Universidad da Coruña                       **)
(**                                                         **)
(**             Propiedad Irreflexiva                       **)
(**                                                         **)
(** Autores: Gilberto Perez Vega y Jose Luis Freire Nistal **)
(*************************************************************)

Section propiedades.
```

```
Variable A:Set.
Variable R:A->A->Prop.
Section dec.

(*toda relacion decidible cumple que, si todos los elementos
"menores" que uno dado no son menores que sí mismos, entonces
él tampoco*)

Hypothesis decR:(a,b:A){(R a b)}+{~(R a b)}.

Lemma key:(x:A)((y:A)(R y x)->~(R y y))->~(R x x).
Proof.
Intros x H.
Auto.
Case (decR x x); Auto.
Qed.

(*Como aplicación, he aquí una prueba de que toda relación bien
fundada y decidible es antirreflexiva*)

Theorem norefleA:(well_founded A R)->(a:A)~(R a a).
Proof.
Intros wf x.,
Apply (Wf.well_founded_ind A R wf [z:A]~(R z z) key).
Qed.
End dec.

(*No obstante, aunque este teorema ilustra una aplicación del
lema key, se puede demostrar que todas las relaciones bien
fundadas son antirreflexivas. He aquí una demostración*)

Lemma noreflacc:(x:A)(Acc A R x)->~(R x x).
Proof.
Red.
Intros x H.
Elim H.
Intros.
Apply (H1 x0); Auto.
Qed.
Theorem norefWf:(well_founded A R)->(a:A)~(R a a).
Proof.
Intros wf a.
Red in wf.
Exact (noreflacc a (wf a)).
Qed.
End propiedades.
```

Índice alfabético