Departamento de Ciencias de la Computación
Inteligencia Artificial

Verificación formal en ACL2 del algoritmo de Buchberger

Memoria presentada por
Inmaculada Medina Bulo
para optar al grado de
Doctor en Informática por
la Universidad de Sevilla.

Inmaculada Medina Bulo

V.ºB.º Directores

Dr. D. José Antonio Alonso Jiménez
Dr. D. José Luis Ruiz Reina

Sevilla, octubre de 2003

Agradecimientos

A mis directores de tesis José Antonio Alonso Jiménez y José Luis Ruiz Reina, por su atención y consejo durante estos años.

A los profesores de los departamentos de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Sevilla, y de Lenguajes y Sistemas Informáticos de la Universidad de Cádiz, en especial a: José Luis por su trabajo sobre reducciones abstractas; Chesco por su formalización del lema Dickson; Paco, por su ayuda inestimable como coautor en muchos de mis trabajos y compañero de asignatura; María José, que junto con el resto de compañeros ya citados, ha aportado mucho a las reuniones de investigación mantenidas durante este tiempo; Gerardo por ayudarme con la asignatura cuando lo he necesitado; Antonia, Esther y Juanjo por su apoyo.

A J Strother Moore y Matt Kaufmann de la Universidad de Texas en Austin por haber creado el sistema ACL2 y permitirme pasar tres meses muy productivos en su grupo de investigación. Además, he de nombrar a Matyas Sustik y agradecerle su colaboración y formalización del lema de Dickson. Gracias J y Jo por vuestra atención, consideración y los buenos momentos pasados en Austin y en Cádiz.

A mi familia, en especial a mis padres y hermanos, por su apoyo y sus palabras de aliento.

A mis amigos de verdad, aquéllos a los que se puede llamar a las cinco de la mañana: siempre están ahí con su cariño para animarme y alegrarme. Gracias de verdad.

Por último, tengo que nombrar otra vez a Paco, porque aparte de haber sido mi mejor compañero de departamento, de asignatura y de investigación, es también mi marido, y como tal me ha dado todo el cariño y comprensión que he necesitado en todo este tiempo. Gracias, Paco.

Inmaculada Medina Bulo
Cádiz, 12 de octubre de 2003

Índice general

Índice de tablas

Índice de figuras

Parte I

Preliminares

Capítulo 1

Introducción

El término *métodos formales* se refiere al empleo de técnicas matemáticas en el desarrollo de sistemas informáticos. Los métodos formales agrupan un conjunto de técnicas muy diversas que son susceptibles de aplicarse en la especificación, análisis, diseño, implementación y mantenimiento de un sistema informático, ya sea físico o lógico. Bien aplicadas, estas técnicas pueden producir productos de gran calidad, bien documentados y fáciles de mantener.

Uno de los problemas principales a los que se han aplicado métodos formales es el *problema de la corrección* [5]. Particularmente, en el caso de un algoritmo o de un programa, se trata de decidir si éste cumple su especificación, un proceso que se conoce como *análisis de la corrección* o más comúnmente como *verificación*. Como argumentaremos a continuación, este problema es tan antiguo como los propios programas.

En 1946, H. H. Goldstine y J. von Neumann introdujeron los diagramas de flujo como un medio de representar los algoritmos que se iban a implementar en las primeras computadoras electrónicas. Ambos autores adquirieron rápidamente la firme convicción de que la programación era algo más que la traducción de unas instrucciones expresadas en un lenguaje matemático al lenguaje de la máquina y que no era, en absoluto, una actividad trivial [99]. De hecho, reclamaron su carácter lógico e independiente:

> Since coding is not a static process of translation, ...it has to be viewed as a logical problem and one that represents a new branch of formal logics.

Fue entonces cuando apareció, de manera embrionaria, el concepto de *aserto*

en el contexto del problema de la corrección. Los diagramas contenían un tipo particular de caja, denominada «caja de aserto», que permitía etiquetar el programa con asertos que documentaban las propiedades que se esperaba cumpliera en un cierto punto.

El primer programa al que se aplicó este método fue una rutina para el cálculo del factorial que fue presentada por A. Turing en un informe de 1949, véase [75]. Esta rutina empleaba sumas en lugar de multiplicaciones y, durante su presentación, Turing expresó abiertamente la preocupación que le había llevado a desarrollar, al parecer de manera independiente, esta idea:

> How can one check a routine in the sense of making sure that it is right? In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.

Nótese cómo Turing separa la tarea de análisis de la de diseño emplazándolas en dos agentes, en principio, separados: el comprobador (responsable de analizar la corrección) y el programador (responsable de diseñar el programa). Acertadamente, Turing incluye como parte de las responsabilidades del programador proporcionar los asertos necesarios para demostrar la corrección del programa. Ambas actividades son consideradas como humanas.

Turing también se da cuenta de que una parte importante del problema general de la corrección es la *terminación*. La siguiente cita conserva hoy toda su vigencia:

> Finally, the checker has to verify that the process comes to an end. Here again he should be assisted by the programmer giving a further definite assertion to be verified. This may take the form of a quantity which is asserted to decrease continually and vanish when the machine stops. To the pure mathematician it is natural to give an ordinal number.

Sin embargo, los trabajos seminales de los tres pioneros fueron en su mayoría desconocidos para las personas que precisamente iban a tener más influencia en el desarrollo de la verificación de programas. Habría que esperar a 1967 para disponer de una formulación rigurosa de las demostraciones de corrección en términos de asertos e inducción, logro que se debe esencialmente a R. W. Floyd quien estableció que el significado de cada operación de un lenguaje de programación puede formularse como una regla lógica que expresa

exactamente qué asertos pueden demostrarse tras la operación a partir de los asertos que eran ciertos antes de ella [24]. Floyd eligió el cálculo de predicados de primer orden como lenguaje para la especificación de los asertos, proclamó la necesidad de proporcionar paralelamente los argumentos de terminación e introdujo explícitamente los conceptos de *invariante de bucle* y *condición de verificación*.

Casi simultáneamente, P. Naur había desarrollado ideas similares, aunque él llamaba a los asertos «instantáneas generales» [76] para indicar que representaban una fotografía del estado del programa en un instante concreto de una de sus ejecuciones, llevada a cabo con datos generales. Sin embargo, el enfoque de Naur para abordar este problema era menos formal y tuvo menos impacto en los trabajos posteriores: las instantáneas generales eran comentarios precisos que acompañaban a los programas, pero no se exigía su expresión en una lógica formal.

Fue C. A. R. Hoare quien en 1969 proporcionó una base axiomática a la programación [34] introduciendo la terminología de *precondición, postcondición e invariante*, las tripletas de Hoare y la lógica de Floyd-Hoare, que hoy día son tan habituales. Corresponde a E. W. Dijkstra y a otros el mérito de extender las ideas previas descubriendo la interesante posibilidad de invertir el sentido de razonamiento para calcular a partir de una postcondición la «precondición más débil» que ha de ser cierta antes de ella. Esta idea hace posible construir programas, cuya corrección está garantizada, comenzando con la especificación de la salida deseada y trabajando hacia atrás [20, 21].

Hemos postpuesto deliberadamente una contribución muy importante y distinta de las realizadas por los anteriores autores. En 1961, John McCarthy presenta su «teoría matemática de la computación» que, a diferencia de los esfuerzos de la época que estaban centrados en la verificación de programas imperativos, toma como fundamento la idea de razonar sobre funciones recursivas. Posteriormente, McCarthy expondrá los vínculos entre las funciones recursivas y los programas iterativos. Sus trabajos fomentarán el estudio conjunto de la inducción y la recursión, y darán lugar a la aparición del lenguaje de programación Lisp.

El amplio desarrollo que han experimentado los métodos formales en diferentes áreas unido a la complejidad creciente de los problemas de interés ha dado lugar a una disciplina denominada *razonamiento automático* [104], que tiene entre sus objetivos el desarrollo de herramientas informáticas para la automatización del razonamiento lógico. Estas herramientas suelen denominarse genéricamente «sistemas de razonamiento automático» e incluso «demostra-

dores automáticos de teoremas», por ser la demostración de teoremas una de las primeras labores en las que tuvieron éxito.

Independientemente del problema de la corrección, y debido principalmente a los éxitos obtenidos tras los trabajos de Robinson [82], se han concentrado grandes esfuerzos en el desarrollo de sistemas de razonamiento basados en resolución para la lógica de primer orden, uno de cuyos máximos exponentes es OTTER [65]. Estos sistemas son automáticos en el sentido de que una vez planteado el problema no existe interacción entre el usuario y el sistema en la búsqueda de la solución.

Sin embargo, el creciente interés en la verificación de sistemas informáticos ha puesto de manifiesto la necesidad de contar con otras lógicas, más potentes o expresivas en algunos aspectos que la de primer orden. Esta mayor potencia ha sido la responsable, en contrapartida, de una pérdida de automatización.

De tal modo esto es así, que el calificativo de «automático» resulta en la actualidad demasiado optimista, cuando no del todo inadecuado, siendo el término «automatizado» más representativo de la verdadera naturaleza de los sistemas disponibles en la actualidad.

En particular, uno de los aspectos clave para la especificación y verificación de la mayor parte de los sistemas informáticos de interés es la inducción. La inducción aparece de uno u otro modo en todos los sistemas de razonamiento que se emplean actualmente en este campo y suele venir acompañada de algún principio de definición inductiva o recursiva para los objetos de la lógica subyacente. Otros aspectos que aparecen en algunos sistemas son el empleo de lógicas de orden superior y de lógicas con tipos, ambos no excluyentes.

En [98] aparece un compendio de sistemas de razonamiento. Entre los sistemas cuya principal característica es el empleo de lógicas de orden superior destacan HOL [30] y PVS [77, 89]. Entre los que utilizan una lógica de tipos constructiva se encuentran NUPRL [14], COQ [22] y LEGO [58]. El grado de automatización es variable, pero estos sistemas generalmente abogan por un estilo interactivo en el que el usuario ha de proporcionar constantemente al sistema indicaciones sobre qué inferencias debe aplicar. Este estilo interactivo es originario de los sistemas inspirados en LCF [29], en los que existe un conjunto de reglas de inferencia que se pueden agrupar en «tácticas» para mayor comodidad de aplicación.

También existen sistemas concebidos para la demostración matemática pura, exentos de todo contenido computacional. Actualmente, uno de sus mejores representantes es MIZAR [83]. Este sistema presenta un nivel muy bajo de

automatización: tanto es así que no es usual referirse a él como un «demostrador» sino como un «comprobador», ya que se limita a comprobar si la prueba presentada por el usuario, en el lenguaje formal de MIZAR, es correcta.

1.1. Objetivos

El objetivo de esta memoria ha sido desarrollar la teoría necesaria para poder implementar, especificar y verificar el *algoritmo de Buchberger* para el cálculo de bases de Gröbner en un sistema de razonamiento automatizado. Concretamente:

1. Desarrollar una teoría computacional sobre los anillos de polinomios de múltiples variables.
2. Proporcionar un orden natural a los polinomios de múltiples variables y demostrar su buena fundamentación.
3. Representar los conceptos asociados a los ideales polinómicos de manera que sea posible razonar sobre ellos de manera automatizada.
4. Desarrollar una teoría computacional sobre las reducciones polinómicas estableciendo su relación con las reducciones abstractas.
5. Representar los conceptos asociados a las bases de Gröbner de manera que sea posible razonar sobre ellos de manera automatizada.
6. Implementar el algoritmo de Buchberger, especificar sus propiedades y demostrar su corrección.
7. Proporcionar un procedimiento de decisión verificado para el problema de la pertenencia al ideal.

Como se observa, nuestro énfasis se sitúa en la automatización del razonamiento. Dentro de los sistemas de razonamiento adecuados para este tipo de tarea hemos elegido a ACL2 [46, 47, 48, 49]. Este sistema merece especial atención, ya que formaliza un subconjunto de COMMON LISP [96], un lenguaje de programación real.[1]

[1] No es de extrañar que las siglas ACL2 signifiquen *A Computational Logic for Applicative Common Lisp*.

De hecho, LISP y sus dialectos se han usado tradicionalmente en la escritura de sistemas de Álgebra Computacional. Por otro lado, ACL2 es especialmente bueno razonando por inducción. En este sentido, estamos ante un demostrador *heurístico* que incorpora no sólo potentes procedimientos de decisión, sino también estrategias sobre cómo abordar diferentes tipos de pruebas, cuándo expandir la definición de una función o generalizar un término, etc.

La mayor parte de las ideas presentes en ACL2 provienen de un sistema anterior, NQTHM, también llamado, el demostrador de teoremas de Boyer y Moore [4, 8, 9]. ACL2 comienza a ser desarrollado en 1989 por Boyer, Moore y Kaufmann como una mejora sustancial de NQTHM [45, 7], estando las últimas versiones desarrolladas fundamentalmente por los dos últimos autores [51].

Hemos demostrado todos los teoremas (del orden de un millar) que aparecen durante la consecución de los objetivos mencionados en este sistema. Desde el punto de vista lógico, ACL2 es una lógica de primer orden con igualdad sin cuantificadores ni tipos y con funciones recursivas totales. La lógica incluye dos principios de extensión importantes, definición y encapsulado, y un principio de inducción que permite razonar sobre las funciones definidas por recursión.

Como se observa, la lógica en sí es muy débil; en particular, la ausencia de cuantificadores supone una restricción importante de la lógica de primer orden. El hecho de que las funciones hayan de ser totales y la ausencia de tipos también tienen su impacto en la expresividad. Éste es el precio que hay que pagar por una mayor automatización: generalmente, nos concentramos en escribir las proposiciones y lemas intermedios que permiten al sistema demostrar un teorema complejo, proporcionándole de vez en cuando indicaciones cuando se desvía de nuestro plan preconcebido.

Discutamos brevemente la importancia del algoritmo de Buchberger como justificación de su elección para nuestro trabajo.

El gran desarrollo sufrido en la última década por el Álgebra Computacional ha dado como resultado la proliferación de numerosos sistemas de propósito general como MATHEMATICA [103]. Éstos son la culminación de los resultados teóricos obtenidos en el último medio siglo, uno de cuyos descubrimientos centrales fue debido a B. Buchberger cuando en 1965 proporcionó un algoritmo para construir bases de Gröbner [11], que se emplean fundamentalmente para resolver el problema de la pertenencia en ideales polinómicos [57].

H. Hironaka ya había demostrado poco antes la existencia de este tipo de bases, a las que llamó *bases estándar* [33], pero su demostración no era cons-

tructiva y no arrojaba ninguna luz sobre el problema de cómo calcularlas. El algoritmo de Buchberger, aunque al principio no tuvo demasiada difusión, causó finalmente un gran impacto en áreas muy diversas. Originalmente, se empleó para resolver problemas en Geometría Algebraica, pero ha sido aplicado con éxito en campos tan diversos como la Teoría de la Codificación, Estadística, Investigación Operativa y Teoría de Invariantes; éstas y otras aplicaciones se describen en [12]. También se ha empleado para automatizar el razonamiento en lógicas modales [13, 55].

Hoy día está implementado en la mayoría de los sistemas como MuPAD [27], Maple [19] y Mathematica [103], por citar sólo algunos. También existen unos pocos sistemas especialmente diseñados para proporcionar implementaciones particularmente eficientes del algoritmo de Buchberger como CoCoA [81], Singular [32] y Macaulay [31].

Es por esto que consideramos que disponer de implementaciones verificadas del algoritmo de Buchberger es importante.

No obstante, esta tarea no resulta sencilla y lo primero que puede sorprendernos es la riqueza de estructuras algebraicas subyacentes a la propia definición del algoritmo de Buchberger y el esfuerzo que hay que dedicar a su formalización. Hay que emplear un cuerpo conmutativo de coeficientes para definir los monomios con los que construir polinomios de múltiples variables, los monomios deben poseer un orden admisible con el que inducir otro sobre los polinomios. Luego, hay que construir funciones de reducción apropiadas entre los polinomios.

Pero asegurar la corrección del algoritmo y proporcionar un procedimiento de decisión verificado para el problema de la pertenencia al ideal aumenta la variedad de estructuras implicadas y el esfuerzo necesario. En primer lugar, hay que definir el concepto de ideal polinómico finitamente generado y su congruencia inducida. Por otro lado, las funciones de reducción conviene estudiarlas en el marco, más general, de las relaciones de reducción abstractas. Sólo entonces, podemos obtener los resultados necesarios sobre las bases de Gröbner. Es en este esfuerzo adicional donde radica la diferencia entre *programar* el algoritmo y *demostrar* que es correcto.

1.2. Trabajos relacionados

Existen hasta la fecha pocos trabajos sobre la construcción verificada de bases de Gröbner. Principalmente, se podría decir que existen tres proyectos que

son representativos de enfoques y sistemas diferentes.

En 1986, Girard [28] introdujo la distinción entre lógicas externas e integradas. Esta distinción ha sido recogida por otros autores, notablemente Dybjer [23]. En un enfoque externo, primero se construye el programa y luego se utiliza una lógica apropiada para demostrar su terminación y su corrección parcial. Por el contrario, un enfoque integrado implica utilizar una lógica en la que sea posible describir una demostración constructiva acerca de la existencia de los objetos cuyo cálculo nos interesa; posteriormente, se extraería el algoritmo de la propia demostración, de manera más o menos automática.

En 1998, Théry [100, 101] presentó una formalización completa del algoritmo de Buchberger siguiendo un desarrollo externo en Coq.

En 1999, Coquand y Persson [16] presentaron un desarrollo integrado del algoritmo de Buchberger en la teoría de tipos de Martin-Löf [60, 61]. Aunque esta línea es muy prometedora, la formalización presentada está incompleta, si bien se han realizado algunas pruebas formales en el sistema Agda [15].

Por último, existe un tercer enfoque que podríamos clasificar también de externo, pero que en cierto modo lo es más que el primero que se ha comentado. En 2001, Rudnicki, Schwarzweller y Trybulec [84] presentaron una propuesta para formalizar una parte importante del Álgebra Conmutativa en Mizar. Aunque Mizar no es un sistema en el que se puedan escribir directamente algoritmos, los autores proponen diseñar un generador de condiciones de verificación para obtener éstas a partir de código escrito en un lenguaje de programación. Las condiciones se expresarían en el lenguaje de Mizar y posteriormente se podría abordar su verificación en el propio sistema.

Como veremos, nuestro trabajo se encuadra en la primera de estas aproximaciones. La diferencia más notable radica en que utilizaremos una lógica muy sencilla para razonar, demostrando así que es posible formalizar el algoritmo y sus propiedades sin apelar a lógicas tan elaboradas. Esta simplicidad será a la vez amiga y enemiga: por un lado, nos permitirá elevar notablemente el grado de automatización de las demostraciones y, por otro, nos obligará en ocasiones a tratar argumentos clásicos de manera poco convencional. Finalmente, nuestros algoritmos son susceptibles de ejecutarse directamente en el sistema en el que son construidos, cosa que no ocurre en ninguno de los otros casos.

1.3. Estructura general

A continuación describiremos cada uno de las partes que componen este trabajo dando una visión general de los aspectos más importantes que lo integran.

El sistema ACL2

En el capítulo 2 se expone una breve descripción de ACL2. Esta descripción no es, en ningún modo, exhaustiva, sino que se centra en presentar las nociones básicas y los diferentes puntos de vista desde los que puede abordarse su estudio.

Con esta introducción a ACL2 únicamente se pretende facilitar la comprensión de esta memoria. Se recomienda especialmente la lectura de este capítulo, así como de sus referencias principales, a aquellas personas sin experiencia previa en la lógica de ACL2.

Anillos polinómicos y su ordenación

Considérese un anillo $A = C[x_1, \ldots, x_k]$ de polinomios de $k \in \mathbb{N}^*$ variables sobre un cuerpo conmutativo arbitrario C. Los elementos de A son polinomios en las indeterminadas $x_1, \ldots, x_k$ con coeficientes en C. Éstos están constituidos por monomios de A, que son productos de potencias de la forma $c \cdot x_1^{a_1} \cdots x_n^{a_k}$, donde $c \in C$ es el coeficiente y $x_1^{a_1} \cdots x_n^{a_k}$ es el término, con $a_1, \ldots, a_k \in \mathbb{N}$.

Esta descripción sirve de punto de partida para la formalización en ACL2 de los anillos polinómicos de múltiples variables, que se presenta en el capítulo 3.

Se aborda pues el problema de la representación. Rápidamente se descarta la posibilidad de emplear una representación densa (donde aparecen explícitamente los monomios nulos), ya que en el caso de múltiples variables es tremendamente ineficiente. A continuación se discute la conveniencia de emplear una representación normalizada frente a una desnormalizada. En una representación dispersa (es decir, no densa) y normalizada, una vez fijado el número de variables, se puede asociar una forma canónica a cada polinomio en la que los monomios están en orden estrictamente decreciente y no existe ninguno que sea nulo. Una de las ventajas de esta representación se encuentra a la hora de decidir la igualdad de polinomios.

No obstante, parece más sencillo comenzar por demostrar las propiedades de la representación desnormalizada aunque posteriormente resulte más conveniente emplear la normalizada para razonar, debido principalmente a la unicidad de dicha representación. Para conjugar estos dos aspectos, se comienza por desarrollar operaciones de anillo desnormalizadas y luego se define una función de normalización con la que se crean sus contrapartidas normalizadas. La función de normalización permite definir una igualdad semántica sobre los polinomios.

Se demuestra en ACL2 que las operaciones de anillo cumplen las propiedades que cabe esperar de ellas, así como la corrección de la función de normalización y la congruencia de la igualdad semántica con las operaciones de anillo.

La formalización es abstracta, en el sentido de que el conjunto de coeficientes y sus propiedades aparecen *encapsulados* en ACL2. Un encapsulado es una construcción en ACL2 que agrupa las nociones clásicas de teoría y modelo, permitiendo extender la lógica de ACL2 conservativamente. Esto permite abstraer las propiedades necesarias de los coeficientes de manera que éstos puedan sustituirse posteriormente por otros que cumplan idénticas propiedades.

A continuación, en el capítulo 4, se desarrolla un orden entre polinomios que viene inducido por el orden de monomios subyacente que aparece junto a la normalización. Se ha escogido aquí un orden lexicográfico para los monomios, por ser uno de los más comúnmente usados. Son imprescindibles aquí las propiedades de buena fundamentación de tales órdenes que, por razones técnicas de la lógica de ACL2, se demuestran mediante inmersión ordinal. ACL2 permite ordinales hasta ϵ_0, lo que en principio supone una limitación teórica de la lógica, pero que en la práctica no afecta a nuestro trabajo.

El anillo de polinomios desarrollado, que es abstracto y descansa sobre unos coeficientes arbitrarios, se particulariza en el capítulo 5 para obtener un anillo de polinomios de múltiples variables sobre el cuerpo de coeficientes de los números racionales. Esto nos proporciona una implementación ejecutable y verificada de los polinomios racionales que serán los que emplearemos posteriormente en el resto de la memoria.

Por último, se introducen nuevas operaciones para incluir conceptos relacionados con la división de términos y también para poder comprobar la pertenencia a polinomios de términos y monomios particulares.

Aquí se observa cómo se aprovecha el potencial de reutilización que presenta

ACL2: demostramos las propiedades más importantes en un marco abstracto y luego las particularizamos y extendemos conforme nos interesa. El mayor trabajo inicial queda compensado por la mayor generalidad de los resultados obtenidos y la posibilidad de obtener distintas instancias verificadas con menor esfuerzo.

Ideales polinómicos: el problema de la pertenencia

El capítulo 6 se dedica a la formalización en ACL2 de la noción de ideal polinómico. Sea un anillo conmutativo A y $B \subseteq A$. Recordemos que se define $\langle B \rangle$, el ideal generado por B, como el menor de los ideales de A que contiene a B. Es posible demostrar que $\langle B \rangle$ coincide con el conjunto de las combinaciones lineales de los elementos de B con coeficientes en A. Al conjunto B se le denomina base de $\langle B \rangle$.

Estamos interesados fundamentalmente en los ideales de anillos polinómicos cuyos coeficientes forman un cuerpo. Como consecuencia del teorema de la base de Hilbert, estos ideales están finitamente generados. Recordemos que un ideal I está finitamente generado si existe F finito tal que $I = \langle F \rangle$.

Como consecuencia de lo anterior, podemos considerar únicamente bases finitas sin pérdida de generalidad y la pertenencia de p al ideal generado por $F = \{f_1, \ldots, f_n\}$ se puede expresar con un predicado de la forma $\exists c_1, \ldots, c_n \; p = \sum_{i=1}^{n} c_i \cdot f_i$. Esto nos permite representar ideales de manera implícita a través de un predicado que recibe p y F, y nos permite expresar la pertenencia del polinomio al ideal generado por la base en cuestión, que es finita.

No obstante, surge inmediatamente un problema relacionado con la falta de potencia expresiva de la lógica que empleamos. En ACL2 no existen cuantificadores existenciales por lo que, en principio, no resulta posible definir un predicado tal. Una alternativa que hemos considerado útil para solventar este problema de expresividad es la de ocultar el cuantificador introduciendo una función de Skolem. Esta función se define junto a unos axiomas que la restringen de manera que esté obligada a devolver una lista de coeficientes que atestigüen la pertenencia del polinomio al ideal. Afortunadamente, ACL2 dispone de un mecanismo que simplifica este tipo de construcciones, aunque el sistema no proporciona prácticamente ningún soporte para la automatización de las demostraciones en las que se emplea este mecanismo.

Una vez definido el predicado de pertenencia al ideal, se demuestran las propiedades fundamentales que debe satisfacer un ideal, obteniéndose prin-

cipalmente que un ideal es cerrado bajo las operaciones de anillo.

A continuación, se define un concepto estrechamente relacionado con el de pertenencia al ideal: el de congruencia inducida. La congruencia inducida por un ideal I, que representaremos por $\equiv_I$, se define de la siguiente forma:

$$p \equiv_I q \iff p - q \in I$$

Se obtiene directamente que el problema de la pertenencia a un ideal es resoluble si, y sólo si, su congruencia inducida es decidible. Posteriormente, en el capítulo 9, se proporciona un procedimiento de decisión verificado para la pertenencia al ideal.

Relaciones de reducción entre polinomios

El algoritmo de Buchberger descansa sobre la noción de *reducción polinómica*. Dado un polinomio $f \neq 0$, la relación de reducción $\to_f$ sobre polinomios inducida por f se define de manera que $p \to_f q$ si p contiene un monomio $m \neq 0$ tal que existe otro monomio c que verifica que $m = -c \cdot mp(f)$ y $q = p + c \cdot f$, siendo $mp(f)$ el monomio principal de f respecto del orden polinómico establecido en el capítulo 4.

A m, c y f los llamamos respectivamente monomio, factor y polinomio de reducción. Decimos también que p se reduce a q mediante f en un paso de reducción.

A continuación, esta definición puede extenderse a conjuntos de manera natural. Si $F = \{f_1, \ldots, f_k\}$ es un conjunto finito de polinomios no nulos, la relación de reducción $\to_F$ inducida por F se define como $\to_F = \bigcup_{i=1}^{k} \to_{f_i}$.

Es interesante, y útil para nuestros fines, emplear conceptos y resultados generales de las reducciones abstractas [1] para razonar sobre estas nociones particulares de reducción. En el capítulo 7 se presenta la formalización de las reducciones sobre polinomios dentro del marco suministrado por [86]. En dicho trabajo, se formalizan las reducciones abstractas en ACL2 como funciones binarias definidas sobre un cierto dominio que a partir de un objeto y de un operador, calculan otro objeto del mismo dominio llevando a cabo un paso de reducción. Un operador no puede ser aplicado a cualquier objeto, por lo que se introduce también un predicado binario que permitirá decidir cuándo dicha aplicación es válida.

Posteriormente, esto nos permitirá traspasar mediante instanciación funcional propiedades bien conocidas de las reducciones abstractas al caso par-

ticular de las reducciones polinómicas, evitando la necesidad de volver a demostrarlas partiendo de cero.

Así, adaptar las reducciones polinómicas al marco abstracto de [86] requiere definir tres funciones.[2] En primer lugar, el predicado unario que especifica el conjunto sobre el cual pretendemos que esté definida la reducción; en nuestro caso particular, el de los polinomios. En segundo lugar, una función binaria que representa la aplicación de un operador a un objeto, donde cada operador se representa por una estructura $\langle m, c, f\rangle$ que consta de tres campos: el monomio de reducción, m, el factor de reducción, c, y el polinomio de reducción, f. Por último, definimos también un predicado binario que comprobará si es válido aplicar un operador a un objeto.

Para que sea válido aplicar un operador $\langle m, c, f\rangle$ a un polinomio p respecto a un conjunto de polinomios F, p debe contener el monomio m, f debe ser un polinomio perteneciente a F y $c = -m/mp(f)$. Como se observa, la última condición implica que se debe comprobar también que $f \neq 0$ y que $mp(f)$ divida a m.

El siguiente paso es definir la relación $\leftrightarrow_F^*$ inducida por $\rightarrow_F$ en términos de las tres funciones anteriormente descritas.[3] Debido a las limitaciones de la lógica de ACL2, y siguiendo a [86], se añade un parámetro extra a la función para evitar la aparición del cuantificador existencial. Este nuevo parámetro tiene como cometido almacenar la secuencia de pasos de reducción que se realizan. Esta secuencia de pasos tiene una interpretación natural: representa la justificación (o *prueba*) de la relación existente entre los polinomios implicados.

Cada paso de prueba se representa mediante una estructura con cuatro campos: un booleano que indica su dirección (directa o inversa), el operador que se aplica y los polinomios que intervienen, es decir, los que quedan conectados en dicho paso.

Una vez que se define la función que comprueba la validez de un paso de prueba, pasamos a definir qué entendemos por equivalencia entre dos polinomios respecto de la reducción polinómica que se ha definido.

A continuación, se demuestra que $\leftrightarrow_F^*$ coincide con la congruencia inducida por el ideal generado por F, con lo que se pone también de manifiesto la

[2]En realidad, abusaremos del lenguaje y llamaremos predicados a las funciones booleanas, ya que ésta es la terminología que emplea la lógica de ACL2. Técnicamente, ACL2 no tiene predicados.

[3]Por $\leftrightarrow^*$ representamos la clausura de equivalencia de una relación $\rightarrow$, es decir, la menor relación que la extiende y es reflexiva, simétrica y transitiva.

conexión existente entre la equivalencia y la pertenencia al ideal.

$$p \leftrightarrow_F^* q \iff p \equiv_{\langle F \rangle} q \iff p - q \in \langle F \rangle$$

Entonces se aborda la importante cuestión de la noetherianidad de $\rightarrow_F$. La inexistencia de cadenas infinitas de elementos relacionados se establece demostrando que con cada paso de reducción se obtiene un polinomio más pequeño en el orden de polinomios definido en el capítulo 4. Recordemos que en él se demuestra que dicho orden está bien fundamentado, con lo que se obtiene inmediatamente la noetherianidad de la relación de reducción.

Por otro lado, se define el cálculo de formas normales, introduciendo una función de normalización fn_F que a partir de un polinomio p y de un conjunto de polinomios F calcula una forma normal de p con respecto de F. Posteriormente, en el capítulo 8 se demuestra que, bajo ciertas condiciones para F, la forma normal es única.

Para terminar el capítulo, se define la función de reducción sobre polinomios red_F^* que se emplea en el algoritmo de Buchberger. Primero se crea la función que reduce un polinomio por otro dado y luego se extiende a conjuntos. Se demuestra la equivalencia de esta función con la que se ha definido previamente para calcular formas normales. Por último, se demuestra la estabilidad del ideal respecto a la función de reducción; esto es muy importante para asegurar posteriormente que el ideal permanece invariante durante la computación de una base de Gröbner.

Bases de Gröbner: el algoritmo de Buchberger

Se dice que G es una base de Gröbner de un ideal I si $I = \langle G \rangle$ y

$$p \in I \iff p \rightarrow_G^* 0$$

En esta caracterización se aprecia inmediatamente la importancia de la noción de base de Gröbner: dichas bases nos permiten resolver fácilmente el problema de la pertenencia al ideal. Si encontramos una base de Gröbner de un ideal, dispondremos de un procedimiento de decisión para su problema de pertenencia.

Buchberger demostró que, para determinar si una base es de Gröbner, basta comprobar que se reducen a 0 unos ciertos polinomios construidos a partir de la base, llamados *s-polinomios*, en lugar de tener que hacerlo con todos

los polinomios del ideal. El número de tales s-polinomios puede ser muy elevado, pero es siempre finito. Esto permite reducir una comprobación que, en principio, era infinita por otra finita.

Para ver si una base de un ideal es de Gröbner, lo que se hace es comprobar si los s-polinomios se reducen a cero. Si no es el caso, existe una solución trivial para que el s-polinomio se reduzca a cero, y es añadirlo a la base. Este nuevo polinomio generará nuevos s-polinomios, cuya reducción a cero debe ser comprobada, y así sucesivamente. Es posible demostrar que este proceso termina, obteniendo como resultado una base de Gröbner del ideal de partida. Este algoritmo es justamente el de Buchberger.

La teoría necesaria sobre bases de Gröbner se desarrolla en el capítulo 8, mientras que la implementación verificada del algoritmo de Buchberger en ACL2 se encuentra en el capítulo 9. La demostración de corrección del algoritmo se divide de manera natural en terminación y corrección parcial.

Para establecer la corrección hay que considerar que la terminación de las funciones en ACL2 ha de estar fundamentada en la existencia de una inmersión ordinal adecuada de sus parámetros. Éste es un requisito ineludible para que ACL2 admita la función bajo su principio de definición y extienda la lógica con el axioma correspondiente. ACL2 dispone de una única estructura bien fundamentada: los ϵ_0-ordinales con su relación de orden habitual. La buena fundamentación de esta estructura es un hecho metateórico que los autores de ACL2 demuestran fuera de la lógica.

A partir de esta estructura, y siempre por inmersión ordinal, introducimos nuevas estructuras bien fundamentadas. En particular, para abordar la terminación del algoritmo de Buchberger empleamos un producto lexicográfico de dos relaciones bien fundamentadas.

La razón para esto es que en el algoritmo existen dos llamadas recursivas. En la primera, el primer parámetro permanece inalterado mientras el segundo decrece estructuralmente. En la segunda, el primer parámetro decrece en un cierto sentido bien fundamentado pese a que se le añade un nuevo polinomio. Este decrecimiento es consecuencia del lema de Dickson.

Siguiendo estas líneas se define la medida ordinal que permite demostrar la terminación del algoritmo de Buchberger a partir de la que se utiliza en [64] para demostrar el lema de Dickson en ACL2.

En cuanto a la corrección parcial, ésta se puede dividir en una serie de etapas que nos conducen al resultado final y que se presentan esquemáticamente a continuación. Estas etapas se expanden a lo largo de tres capítulos. Partimos

de dos resultados importantes sobre las reducciones y las formas normales que se demuestran en el capítulo 7, seguimos con los resultados principales sobre bases de Gröbner del capítulo 8 y terminamos con los resultados que aseguran la corrección parcial del algoritmo de Buchberger descritos en el capítulo 9.

1. Se demuestra que la función fn_F calcula una forma normal, es decir, que $p \to_F^* fn_F(p)$ y $fn_F(p)$ es irreducible (teorema 7.29).

2. Se demuestra que $fn_F(p) = red_F^*(p)$ (teorema 7.40), y por lo tanto que $p \to_F^* red_F^*(p)$ (teorema 7.41), donde red_F^* es la función de reducción que se utiliza en el algoritmo de Buchberger.

3. Consideramos la siguiente propiedad $\Phi(F)$ que expresa el hecho de que los s-polinomios formados a partir de polinomios de una base F se reducen a 0.

$$\Phi(F) \equiv \forall p, q \in F \; s\text{-}polinomio(p, q) \to_F^* 0$$

   Se demuestra que $\Phi(F)$ implica que la relación de reducción es localmente confluente (teorema 8.12).

$$\Phi(F) \implies \forall p, q, r \; (r \to_F p \wedge r \to_F q \implies p \downarrow_F^* q)$$

4. Se demuestra que la clausura de equivalencia inducida por un conjunto de polinomios que verifica la propiedad Φ se puede decidir comprobando la igualdad de formas normales (teorema 8.13).

$$\Phi(F) \implies (p \leftrightarrow_F^* q \iff fn_F(p) = fn_F(q))$$

5. Se demuestra que cualquier conjunto de polinomios que satisfaga la propiedad Φ es una base de Gröbner (teorema 8.14).

$$\Phi(F) \implies (p \in \langle F \rangle \iff p \to_F^* 0)$$

6. Sea $Buchberger(F)$ el resultado de aplicar el algoritmo de Buchberger a la base F. Se demuestra que se cumple $\Phi(Buchberger(F))$ (teorema 9.3).

7. Se demuestra la estabilidad del ideal respecto al algoritmo de Buchberger (teorema 9.23).

$$p \in \langle F \rangle \iff p \in \langle Buchberger(F) \rangle$$

8. Se demuestra que la base devuelta por el algoritmo de Buchberger es una base de Gröbner (teorema 9.24). Si $G = Buchberger(F)$:

$$p \in \langle G \rangle \iff p \to_G^* 0$$

9. Se demuestra, utilizando los resultados anteriores, que la base devuelta por el algoritmo de Buchberger es una base de Gröbner de la base original. Es decir, si $G = Buchberger(F)$, se tiene que G es una base de Gröbner de $\langle F \rangle$ (teorema 9.25):

$$p \in \langle F \rangle \iff p \to_G^* 0$$

Para terminar el capítulo 9, construimos un procedimiento de decisión verificado para el problema de la pertenencia al ideal (teorema 9.27). Para ello, sólo hay que observar que si $G = Buchberger(F)$ se tiene que:

$$p \in \langle F \rangle \iff red_G^*(p) = 0$$

Como se puede comprobar, no nos limitamos a demostrar que el algoritmo de Buchberger devuelve una base de Gröbner sino que proporcionamos un procedimiento de decisión verificado para el problema de la pertenencia al ideal.

1.4. Contenido del CD

Junto a esta memoria se suministra un CD que contiene los ficheros ACL2, (llamados *libros* en la terminología ACL2), que se han desarrollado para llevar a cabo la formalización presentada. Además se incluye un fichero `LÉAME` en el que se describen los pasos necesarios para la instalación del sistema ACL2 y para evaluar en él la formalización.

Pueden encontrarse todos los libros en el directorio `Código`. Estos libros, que poseen todos extensión `.lisp`, ya han sido *certificados*. Esto quiere decir, que el sistema ha admitido todas las definiciones presentadas y logrado demostrar todos los teoremas que se han enunciado. La salida que produce el sistema con el resultado del proceso de certificación se encuentra en los ficheros con extensión `.out` correspondientes. Advertimos que estos ficheros pueden ser muy largos, ya que contienen las demostraciones generadas con todo el detalle que precisa un sistema automatizado como es ACL2.

El directorio `Código` se organiza en los subdirectorios y ficheros que se describen a continuación:

Subdirectorio `polinomios`

Este subdirectorio desarrolla los anillos de polinomios desnormalizados descritos en el capítulo 3.

- `coeficiente`: un anillo abeliano de coeficientes. El conjunto de los coeficientes se representa como un anillo abeliano abstracto mediante un encapsulado. El conjunto de los números de ACL2 con su interpretación habitual sirve como modelo de la teoría generada.
- `termino`: un monoide conmutativo de términos con un orden bien fundamentado cuya representación se abstrae mediante un encapsulado. Las listas propias de números naturales de ACL2 con la suma elemento a elemento y el orden lexicográfico sirven como modelo de la teoría generada. La buena fundamentación del orden se establece por inmersión en los ϵ_0-ordinales.
- `monomio`: pares coeficiente-término. Se define una igualdad semántica, ya que dos monomios con coeficiente nulo han de ser interpretados como el mismo, aunque tengan distinto término. También se implementa el orden de monomios que es heredado de los términos.
- `polinomio`: representación abstracta de los polinomios mediante listas propias de ACL2 formadas por monomios que contienen coeficientes y términos abstractos.
- `forma-normal`: desarrollo de la función de normalización que permite reducir la comprobación de la igualdad semántica de dos polinomios a la de una igualdad sintáctica de sus formas normales. Se define primero una suma externa de un monomio con un polinomio ordenado, teniendo en cuenta una posible cancelación. A partir de aquí se define la forma normal y su relación de equivalencia inducida. Se demuestran algunas propiedades importantes como la idempotencia de la normalización.
- `suma`: desarrollo de la suma de polinomios definida simplemente como la concatenación de las listas de monomios que los integran. Las propiedades de la concatenación de listas permiten establecer la base para realizar demostraciones de propiedades sobre los polinomios más complicadas que incorporan la igualdad semántica. Se demuestra que los polinomios con la operación de suma forman un monoide conmutativo.
- `congruencias-suma`: demostración de las congruencias de la igualdad de polinomios con la suma.

- `opuesto`: desarrollo del opuesto de un polinomio, que se define monomio a monomio. Su corrección se prueba demostrando que la función que lo calcula produce el inverso aditivo. Para que éstas y otras propiedades sean incondicionales (carezcan de hipótesis) se completa cuidadosamente la definición de la función. Se demuestra que los polinomios con las operaciones de suma y opuesto forman un grupo conmutativo.

- `producto`: desarrollo del producto externo de un monomio por un polinomio y del producto de polinomios. Las funciones se completan cuidadosamente, de lo contrario, no es posible establecer las congruencias, ya que éstas no pueden contener hipótesis. Se demuestra que los polinomios con el producto forman un monoide conmutativo y que el producto distribuye respecto de la suma, completándose con esto la demostración de las propiedades del anillo de polinomios.

- `congruencias-producto`: demostración de las congruencias de la igualdad de polinomios con el producto de un monomio y un polinomio, y el producto de polinomios.

Subdirectorio `polinomios-normalizados`

Este subdirectorio desarrolla los anillos de polinomios normalizados descritos en el capítulo 3. Además se introduce el orden bien fundamentado sobre los polinomios descrito en el capítulo 4.

- `polinomio-normalizado`: polinomios normalizados definidos a partir de los polinomios desnormalizados y de la operación de normalización. Ascenso de las propiedades de anillo de la representación desnormalizada a la normalizada.

- `orden`: extensión del orden de monomios a los polinomios y demostración de su buena fundamentación mediante una inmersión de los polinomios en los ϵ_0-ordinales.

En la figura 1.1 aparecen las dependencias entre los libros desarrollados para el anillo de polinomios con coeficientes arbitrarios. Una flecha de un libro a otro indica que el primero se incluye en el segundo.

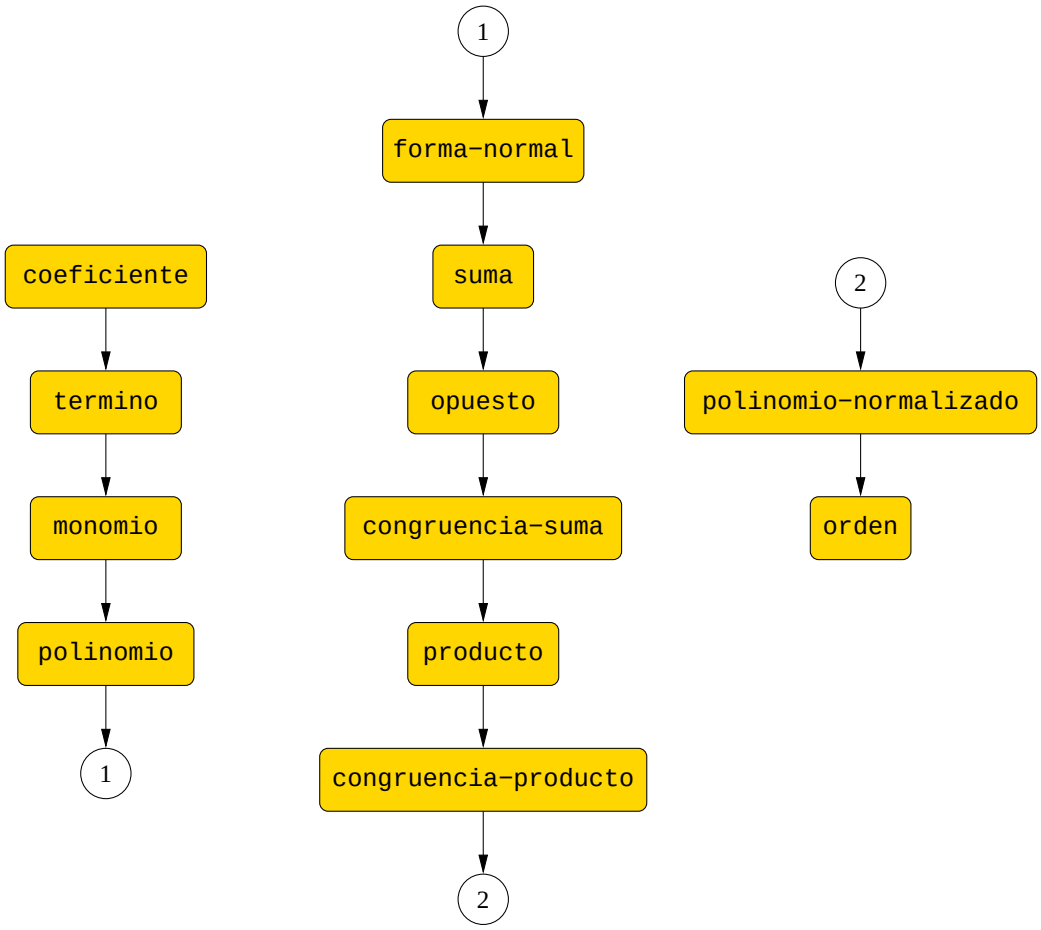

Figura 1.1: Dependencias entre libros: anillo de polinomios normalizados

Subdirectorio `polinomios-racionales`

Este subdirectorio desarrolla un anillo de polinomios concreto: el anillo de los polinomios racionales descrito en el capítulo 5. Gran parte de este desarrollo consiste en instanciar funcionalmente los teoremas que aparecen encapsulados en los libros de los subdirectorios `polinomios` y `polinomios-normalizados`.

Por otra parte, se incluyen nuevas definiciones y propiedades. En primer lugar, se introduce la operación de división en los coeficientes (interesa establecer propiedades de cuerpo sobre ellos). En segundo lugar, aparecen conceptos relacionados con la divisibilidad en los términos (divisibilidad, división y mínimo común múltiplo). En tercer lugar, se introduce la pertenencia de monomios a polinomios y de polinomios a conjuntos de polinomios. Por último, se define el concepto de polinomio uniforme para unificar el conjunto de variables empleado.

- `racional`: instanciación de los coeficientes abstractos con los números racionales de ACL2. Extensión con la operación de división y algunas de sus propiedades más importantes.
- `termino`: instanciación de los términos abstractos con las listas propias de números naturales de ACL2.
- `termino-division`: desarrollo de la divisibilidad, la división y el mínimo común múltiplo sobre términos. Demostración de sus propiedades fundamentales.
- `monomio`: instanciación de los monomios abstractos para conseguir pares racional-término, empleando los términos concretos.
- `polinomio`: instanciación de los polinomios abstractos para conseguir polinomios de coeficientes racionales que se representan como listas propias de ACL2 cuyos componentes son ya monomios concretos.
- `forma-normal`: instanciación de la función de normalización para poder trabajar con polinomios de coeficientes racionales.
- `suma`: instanciación de la suma de polinomios para poder trabajar con polinomios de coeficientes racionales.
- `congruencias-suma`: demostración de las congruencias de la igualdad de polinomios con la suma por instanciación funcional de las demostradas para polinomios abstractos.

- `opuesto`: instanciación del opuesto de polinomios para poder trabajar con polinomios de coeficientes racionales.
- `producto`: instanciación del producto de polinomios (externo e interno) para poder trabajar con polinomios de coeficientes racionales.
- `congruencias-producto`: demostración de las congruencias de la igualdad de polinomios con el producto externo y el producto por instanciación funcional de las demostradas para polinomios abstractos.
- `polinomio-normalizado`: instanciación de los polinomios normalizados abstractos para conseguir polinomios normalizados de coeficientes racionales.
- `orden`: instanciación del orden de polinomios para poder trabajar con polinomios de coeficientes racionales.
- `pertenencia`: desarrollo de las funciones que comprueban la pertenencia de un monomio a un polinomio y de un polinomio a una lista de polinomios. Demostración de sus propiedades fundamentales.
- `defun-k`: macros `defun<k>` y `defun-sk<k>` para definir funciones con un parámetro k implícito.
- `k-polinomio`: concepto de uniformidad, polinomios de k variables y sus propiedades básicas. Se demuestra que las operaciones anteriormente definidas sobre los polinomios preservan la uniformidad.

En la figura 1.2 aparecen las dependencias entre los libros desarrollados para el anillo de polinomios racionales. Omitimos las dependencias con los libros de los que se instancian los resultados genéricos, que son los correspondientes de la figura 1.1.

Subdirectorio `Buchberger`

Este subdirectorio desarrolla los ideales polinómicos finitamente generados, las reducciones polinómicas y el algoritmo de Buchberger para el cálculo de bases de Gröbner. El resultado final es una implementación verificada de dicho algoritmo, esto conduce a la obtención de un procedimiento de decisión verificado para el problema de la pertenencia al ideal.

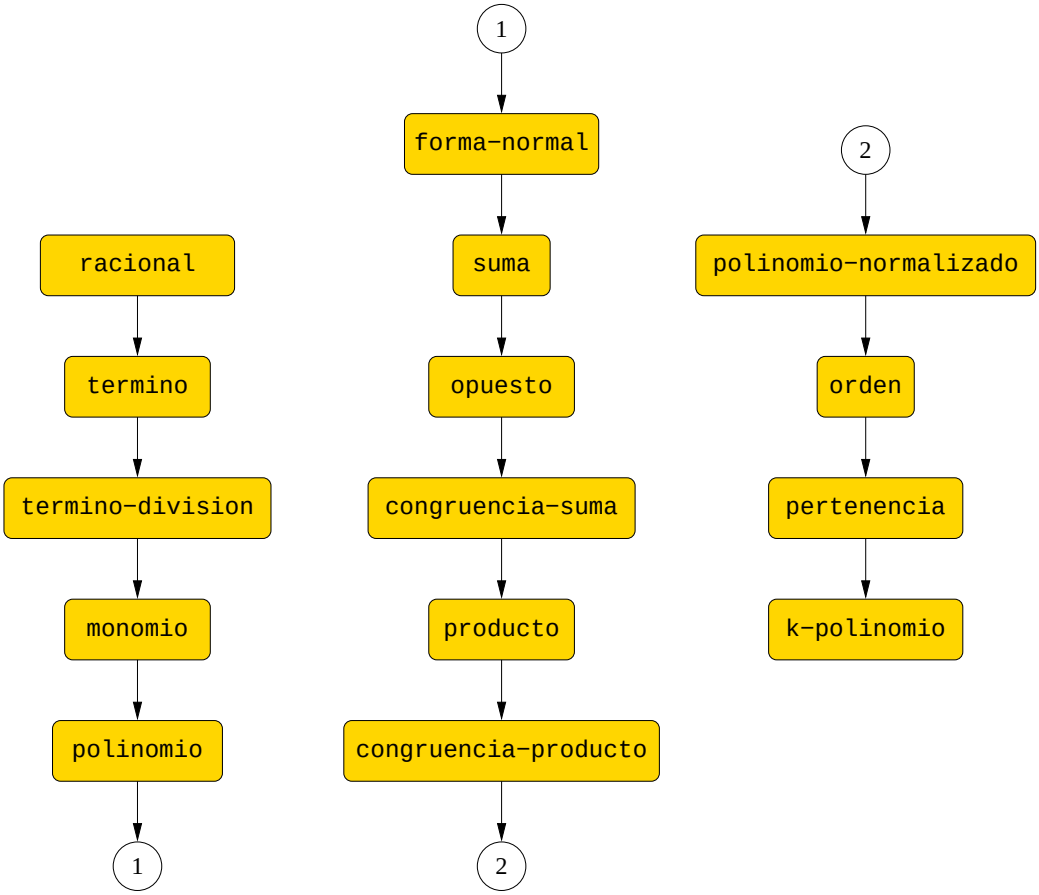

Figura 1.2: Dependencias entre libros: anillo de polinomios racionales

- `ideal`: ideales polinómicos, relación de pertenencia y propiedades fundamentales descritas en el capítulo 6.
- `reduccion-polinomica`: relaciones de reducción sobre polinomios y sus propiedades principales descritas en el capítulo 7.
- `congruencia-ideal`: congruencia inducida por un ideal y demostración de que coincide con la relación de equivalencia definida en el capítulo 7.
- `noetherianidad`: noetherianidad de la relación de reducción sobre polinomios presentada en el capítulo 7.
- `forma-normal`: desarrollo del concepto de forma normal respecto de la relación de reducción a través de operadores, tal y como se describe en el capítulo 7.
- `calculo-forma-normal`: funciones de reducción sobre polinomios utilizadas en la implementación del algoritmo de Buchberger. Se demuestra su equivalencia con las formas normales definidas con operadores. Esto se describe en el capítulo 7.
- `estabilidad-reduccion`: estabilidad del ideal bajo la reducción extendida a conjuntos y bajo su clausura. Esto se describe en el capítulo 7.
- `confluencia`: se demuestra que si todos los s-polinomios de una base se reducen a 0 respecto de ella, entonces la relación de reducción inducida es localmente confluente. Esto se describe en el capítulo 8.
- `buchberger`: implementación del algoritmo de Buchberger y demostración de terminación. Se describe en el capítulo 9.
- `s-polinomio`: definición del concepto de s-polinomio y demostración de sus propiedades fundamentales como se describe en el capítulo 9.
- `estabilidad-s-polinomio`: estabilidad del ideal bajo el cálculo de s-polinomios. Esto se describe en el capítulo 9.
- `estabilidad-sufija`: sufijo de una secuencia de polinomios y sus propiedades básicas. Se demuestra que la pertenencia al ideal se preserva bajo cualquier extensión sufija de la base. Este resultado se utiliza en la demostración de que el algoritmo de Buchberger preserva la estabilidad del ideal. Esto se describe en el capítulo 9.

- `estabilidad-buchberger`: estabilidad del ideal bajo el algoritmo de Buchberger. Durante la demostración se emplean diversas propiedades que demuestran que el ideal permanece invariante bajo las operaciones que componen el algoritmo (reducción, cálculo de s-polinomios, etc.). Se describe en el capítulo 9.

- `bases-grobner`: se demuestra que si todos los s-polinomios de una base se reducen a 0 respecto de ella, entonces ésta es una base de Gröbner en el sentido descrito en el capítulo 8. También se demuestra que todos los s-polinomios de la base calculada por el algoritmo de Buchberger se reducen a 0. A partir de estos dos resultados se concluye que el algoritmo de Buchberger calcula una base de Gröbner. Esto se describe en el capítulo 9.

- `decision`: desarrollo del procedimiento de decisión verificado para el problema de la pertenencia al ideal. Se describe en el capítulo 9.

En la figura 1.3 aparecen las dependencias entre los libros desarrollados para el algoritmo de Buchberger.

Subdirectorio `relaciones`

Este subdirectorio desarrolla la formalización de las reducciones abstractas en la lógica de ACL2 presentada en [86]. Entre otros resultados, se presenta la demostración del lema de Newman, que nos permite deducir la confluencia de toda relación noetheriana y localmente confluente, y la decidibilidad de la relación de equivalencia descrita por una reducción convergente. Este último resultado se utiliza en este trabajo para obtener la decidibilidad de nuestra reducción polinómica concreta, como se describe en el capítulo 9. Por otro lado, la aplicación de esos resultados sólo es posible si nuestras reducciones polinómicas se representan dentro del marco suministrado en [86]. Esto se explica en el capítulo 7.

En concreto, se incluirán los ficheros `abstract-proofs` y `convergent` en los ficheros `bases-groebner` y `reduccion-polinomica`, respectivamente.

Subdirectorio `dickson`

Este subdirectorio incluye la demostración del lema de Dickson en ACL2 desarrollada en [64]. Esta formalización se utiliza en este trabajo para llevar

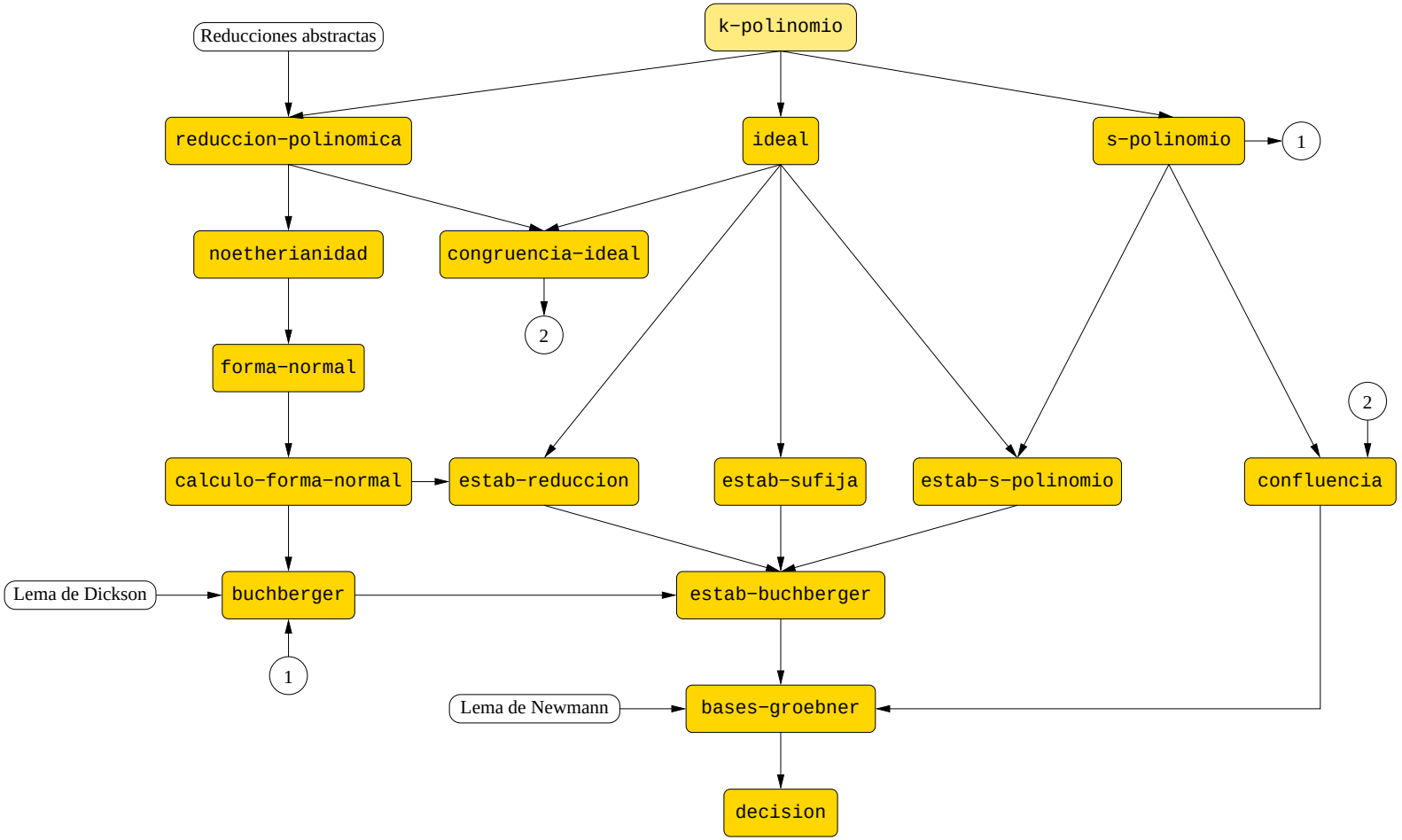

Figura 1.3: Dependencias entre libros: algoritmo de Buchberger

a cabo la construcción de la medida ordinal que se emplea en la demostración de terminación del algoritmo de Buchberger, como se verá en el capítulo 9.

En concreto, será el fichero `dickson` el que se utilizará, incluyéndose en el fichero `buchberger`.

Capítulo 2

La lógica computacional ACL2

2.1. Introducción

Dentro del amplio espectro de sistemas de razonamiento automático, existen varios especialmente adecuados para la especificación y verificación formal de sistemas informáticos, siendo ACL2 uno de ellos.

ACL2 formaliza un subconjunto de un lenguaje de programación real y con él se han obtenido grandes éxitos en la verificación formal automatizada tanto de sistemas de *hardware* como de *software*, además de en la demostración por computador de conocidos teoremas de la Lógica y de las Matemáticas.

Se expondrá a continuación una descripción de ACL2 atendiendo a los diferentes puntos de vista desde los que puede observarse. Esta descripción no es de ningún modo exhaustiva y tiene como principal objetivo facilitar la comprensión del código implementado para formalizar los conceptos y propiedades desarrollados en esta memoria. Una descripción detallada puede encontrarse en [48].

2.2. Las tres visiones de ACL2

ACL2 (*A Computational Logic for Applicative Common Lisp*) [48] es el sucesor de NQTHM, el demostrador de teoremas de Boyer y Moore [4, 9]. Ha sido desarrollado durante la última década en la universidad de Texas en Austin, principalmente por Moore y Kaufmann, y continúa en evolución. Intenta hacer realidad, con bastante éxito, el deseo de McCarthy de disponer

de un sistema capaz de demostrar teoremas sobre programas escritos en LISP puro [73]:

> Instead of debugging a program, one should prove that it meets its specification and this proof should be checked by a computer program.
>
> — John McCarthy, "A Basis for a Mathematical Theory of Computation" (1961)

Tanto NQTHM como ACL2 se han utilizado en áreas muy diversas. Algunos ejemplos son los siguientes:

- La indecidibilidad del problema de parada [6].
- El teorema de Church-Rosser para el λ-cálculo [93].
- La ley de cuadrados recíprocos de Gauss [90].
- El teorema de incompletitud de Gödel [94].
- El teorema de Ramsey [54].
- El teorema fundamental del Cálculo [50].
- La corrección de un algoritmo para el cálculo de la transformada rápida de Fourier [25].
- El microcódigo del algoritmo de división de coma flotante y de cálculo de la raíz cuadrada del procesador AMD K5 [72, 92].
- El código RTL que implementa las operaciones elementales sobre números de coma flotante del procesador AMD Athlon [91].

Puede encontrarse una descripción concisa de ACL2 en [46]. En realidad, para definir qué es ACL2 hay que enfocarlo desde tres puntos de vista distintos:

1. Desde la perspectiva de los lenguajes de programación.
2. Desde un punto de vista lógico.
3. Desde el enfoque de los sistemas de razonamiento automático.

2.2.1. ACL2 es un lenguaje de programación aplicativo

Toda función admitida bajo el principio de definición de ACL2 es una función de COMMON LISP. El recíproco no es cierto, ya que para razonar en la lógica ACL2 sobre una función su ejecución debe sólo depender de sus parámetros reales:[1] a iguales parámetros, igual resultado. Por otro lado, los lenguajes de programación convencionales, y COMMON LISP no es una excepción, permiten definir «funciones» cuya terminación no está garantizada.

Esto hace que sea posible pensar en ACL2 como en un lenguaje de programación *aplicativo* o *funcional puro*. Más explícitamente, puede considerarse que es un subconjunto de COMMON LISP libre de efectos colaterales, esto es, no se permiten variables globales ni modificación destructiva de datos.

Por tanto, se utiliza la sintaxis de COMMON LISP para expresar cualquier concepto en ACL2. Por ejemplo, 1+1 se describe en ACL2 empleando notación prefija, mediante la expresión `(+ 1 1)`.

Al igual que en la mayoría de los dialectos de LISP, ACL2 se presenta al usuario como un bucle de lectura, evaluación e impresión, llamado bucle *read-eval-print*. Es decir, ACL2 actúa como un intérprete que lee la expresión introducida por el usuario, la evalúa, e imprime el resultado. Este proceso prosigue indefinidamente a menos que la evaluación aborte.

Tipos de datos

ACL2 posee cinco tipos de datos básicos: números (naturales, enteros, racionales y racionales complejos), caracteres, cadenas, símbolos y pares ordenados (*conses*). Todos los objetos de estos tipos básicos, a excepción de los pares ordenados, se consideran atómicos en ACL2 y son reconocidos por el predicado `atom`.

Los números se representan de la manera habitual en LISP. Los enteros son reconocidos por el predicado `integerp` y están incluidos en los racionales, que se reconocen mediante `rationalp`. Todos los números son reconocidos por el predicado `acl2-numberp`. No existen números en coma flotante.

Dos símbolos que se emplean con profusión son las constantes `nil` y `t`, reconocidas por el predicado `booleanp`. Estos símbolos lógicos hacen referencia a los valores booleanos «falso» y «verdadero», respectivamente. Sin embargo, en cualquier contexto en el que aparezca una expresión lógica, ACL2 inter-

[1] O su definición no bastaría y sería necesario introducir el concepto de «estado global».

preta `nil` como «falso» y cualquier otro valor como «verdadero», de forma que en tales expresiones cualquier cosa que no sea `nil` actúa como si fuera `t`.

Es importante recordar que todo símbolo pertenece a un paquete. Cada paquete representa un espacio de nombres separado. En cada momento existe un paquete activo o «paquete de trabajo» de manera que al hacer referencia a un símbolo de dicho paquete, el nombre del paquete pueda omitirse. El paquete por defecto es `ACL2`.

Para nombrar explícitamente a un símbolo de un determinado paquete se resuelve el ámbito colocando `::` entre el nombre del paquete y el del símbolo como en `ACL2::if`.

Los pares ordenados o *conses* se construyen con la función binaria `cons` y son reconocidos por el predicado `consp`. Son, sin duda, los objetos más utilizados en ACL2, ya que no son atómicos y permiten construir objetos complejos. Las listas se pueden representar mediante pares ordenados. La primera componente se denomina «primero» y a ella se accede con la función `car`. La segunda componente se denomina «resto» y a ella se accede con la función `cdr`.

Tanto `car` como `cdr` son nombres históricos que poco reflejan su verdadero significado, por lo que pueden ser preferibles sus sinónimos `first` y `rest`, respectivamente.

Es importante saber que al preceder a un objeto con el carácter `'` se le transforma en una constante. Lo mismo ocurre si se le aplica la función especial `quote`. A estas constantes especiales se las denomina *constantes apostrofadas*.

Aunque `nil`, no sea un *cons*, sino un símbolo, es considerado una lista: la *lista vacía*, que es reconocida por el predicado `null`. El conjunto formado por los pares y la lista vacía es reconocido por el predicado `listp`.

Un detalle importante es que, al igual que en COMMON LISP, tanto el `first` como el `rest` de `nil` valen, precisamente, `nil`.

Expresiones

Las expresiones de ACL2 se clasifican en las siguientes categorías:

1. Símbolos de constante: `nil`, `t`, una clave (un símbolo del paquete especial `KEYWORD`), o símbolos definidos con `defconst` que comenzarán y terminarán con el carácter `*`.

2. Expresiones constantes: un número, un carácter, una cadena o una constante apostrofada.

3. Símbolos de variable: todo aquél que no es un símbolo de constante. El valor de una variable respecto de una asignación es el valor que dicho símbolo tiene asignado.

4. Expresiones funcionales: la aplicación de una función definida con `defun`, o de una λ-expresión, a un número de parámetros reales que coincide con su número de parámetros formales. Los parámetros reales son expresiones y los formales, variables.

Funciones y macros

ACL2 incorpora una gran cantidad de funciones y macros primitivas. Aunque la mayoría de estas funciones existen en COMMON LISP, hay que tener en cuenta que pueden existir algunas diferencias sutiles. Por ejemplo, una expresión condicional `if` siempre posee en ACL2 una condición, una rama *entonces* y una rama *si no*. Sin embargo, la rama *si no* puede omitirse en COMMON LISP.

Se puede emplear `cond` en lugar de `if` anidados, aunque esto es una mera cuestión estética. También se puede emplear `case` alternativamente en determinadas ocasiones.

El usuario puede definir nuevas funciones mediante `defun` y nuevas macros mediante `defmacro` (o `defabbrev` que previamente liga los parámetros reales localmente, forzando su evaluación). La aplicación de una macro provoca una mera sustitución textual de los parámetros formales por los reales en su cuerpo previa a la evaluación de éste.

En ocasiones es útil ligar variables localmente (con ámbito léxico). Esto puede lograrse con `let` y `let*`. En realidad, `let` existe sólo por comodidad, ya que es equivalente a la aplicación de una λ-expresión que liga sus parámetros reales a los formales en paralelo.

A veces puede ser útil realizar la ligadura secuencialmente, en lugar de en paralelo, para emplear un valor previamente ligado. En este caso se suele emplear `let*`, que equivale a un `let` anidado.

En ACL2 todas las funciones tienen un número fijo de parámetros, sin embargo, el empleo de macros permite simular funciones con un número variable

de parámetros. Las macros realizan transformaciones meramente sintácticas sobre sus parámetros.

Por ejemplo, `+` es una macro que recibe un número arbitrario de parámetros y que, cuando recibe más de un parámetro, equivale a la aplicación reiterada de la función primitiva `binary-+` que, como su nombre indica, es binaria. Algo similar ocurre con `*`, `or`, `and` y `list`.

Otras macros como `-` y `/` trabajan sólo unaria y binariamente. También las hay que han de recibir al menos un parámetro, como `list*` y `append`.

ACL2 posee dos modos básicos de funcionamiento: modo lógico y modo programa. La diferencia fundamental entre ambos modos es que en modo lógico es obligatorio demostrar la terminación de las funciones introducidas.

Inicialmente ACL2 se encuentra en modo lógico y puede pasar a modo programa mediante la función `program`. El paso contrario se realiza mediante `logic`.

Es absolutamente necesario que introduzcamos nuestras funciones en modo lógico si queremos razonar sobre ellas. Al demostrar su terminación aseguramos que la función está matemáticamente bien definida.

2.2.2. ACL2 es una lógica computacional

Para razonar formalmente sobre las funciones en ACL2 se necesita de algún modo poder describir propiedades sobre ellas. Podría pensarse que es totalmente necesario definir un nuevo formalismo. Sin embargo, no es así.

Por ejemplo, supóngase que se ha programado una función `ordena` que recibe una lista de números enteros y, supuestamente, devuelve dicha lista ordenada ascendentemente. Deseamos demostrar que dicha función es correcta. El problema de la corrección se divide de manera natural en parada y corrección parcial. Olvidemos por un momento el problema de la parada y centrémonos en el de la corrección parcial.

Un aspecto obvio de su corrección parcial es que su resultado es una lista homogénea de números enteros ordenada ascendentemente. Pero esto no es suficiente. Otro aspecto, quizás menos evidente, es que el resultado no debe contener nuevos elementos ni obviar ninguno de los existentes.

Consideremos la primera propiedad; se podría comenzar por comprobar dicha conjetura ejecutando la función `ordena` sobre algunos ejemplares y observan-

do que se cumple. Tras algunas pruebas, podríamos definir un predicado que realizara dichas comprobaciones por nosotros.

Este predicado podría llamarse `ordenadap` y comprobaría si su parámetro es una lista homogénea de números enteros ordenados crecientemente. De esta forma, se podrían evaluar automáticamente expresiones que comprobasen si la salida de `ordena` verifica el predicado `ordenadap`.

Pero aunque repitiéramos estos experimentos durante toda nuestra vida, nunca podríamos estar seguros de que nuestra conjetura es un teorema, ya que hay un número infinito de posibilidades. Por el contrario, un sólo caso en el que el experimento fallase demostraría lo falaz de nuestra conjetura. Aún más, ¿estamos seguros de que la función auxiliar que hemos definido hace bien su trabajo?

Así que, después de realizar un número suficiente de pruebas para convencernos de que la conjetura es plausible y de que no hemos cometido errores crasos, se podría intentar demostrar la propiedad deseada. Muy probablemente quedemos sorprendidos: el teorema bien puede ser falso si no exigimos que `ordena` reciba una lista homogénea de números enteros.

Esto puede parecer trivial a las personas acostumbradas a trabajar con tipos. Pero ACL2 no posee tipos, al menos en el sentido estricto de la palabra, y no se puede expresar lógicamente el hecho de que `ordena` debe recibir un objeto del tipo apropiado durante su definición: debe constar explícitamente como una hipótesis del teorema que pretendemos demostrar.

Otro aspecto importante es que la definición de una función introduce un nuevo axioma y, por consiguiente, ACL2 impone ciertas restricciones para asegurar que no se introducen inconsistencias en la lógica. Como se observa, la lógica de ACL2 es en cierto modo dinámica: depende de la «historia» de la sesión. Al definir una función, las restricciones impuestas por ACL2 aseguran que se obtenga una extensión conservativa de su conjunto de axiomas.

La principal restricción es que las funciones deben ser totales, es decir, toda función debe terminar su ejecución en un tiempo finito cuando se aplica a cualquier objeto de ACL2, supuesto que haya suficientes recursos computacionales para ello. Más aún, esto debe ser demostrado o la función no será aceptada.

Los argumentos de terminación se construyen mediante el concepto de «buena fundamentación». Una estructura bien fundamentada es un conjunto dotado de una relación de orden noetheriana. Es decir, decimos que un conjunto A con una relación $<$ está bien fundamentado si, y sólo si, no existe una se-

cuencia infinita de elementos de A estrictamente decreciente con respecto a $<$. Abusando del lenguaje, obviaremos el conjunto o la relación cuando el contexto lo permita.

El conjunto de los ϵ_0-ordinales (los ordinales menores que ϵ_0) con su relación de orden natural constituyen la única estructura bien fundamentada conocida meta-teóricamente por ACL2. Son por tanto el único modo de demostrar la terminación de funciones en ACL2.

Esto constituye ciertamente una limitación teórica, pero, en la práctica, los ϵ_0-ordinales son suficientes para demostrar la terminación de multitud de funciones.

Lógica de primer orden sin cuantificadores

ACL2 es una lógica de primer orden sin cuantificadores y con igualdad. Su *sintaxis* coincide con la del lenguaje de programación COMMON LISP.

Una *constante* puede ser un símbolo de constante o una expresión constante. Un símbolo de *variable* es cualquier símbolo que no sea un símbolo de constante. Un símbolo de *función* es un símbolo de función primitivo o uno definido por el usuario. Nótese que, en ACL2, un símbolo puede ser a la vez de variable y función.

Un *término* es una constante, una variable o la aplicación de una función (o de una λ-expresión) n-aria a n términos. Más precisamente, si f es un símbolo de función n-aria, $x_1, \ldots, x_n$ son símbolos de variable distintos, $\tau, \tau_1, \ldots, \tau_n$ son términos y $x_1, \ldots, x_n$ son las únicas variables libres de τ, entonces:

$$f(\tau_1, \ldots, \tau_n)$$
$$(\lambda x_1, \ldots, x_n.\tau)(\tau_1, \ldots, \tau_n)$$

son también términos.

Formalmente, en ACL2 no existen símbolos de predicado, aunque a las funciones booleanas se las denomine informalmente «predicados». Siendo estrictos, las únicas fórmulas atómicas existentes son las igualdades entre términos. Una *fórmula* es una fórmula atómica o la aplicación de un operador lógico a otras fórmulas. Los operadores lógicos son la igualdad $=$ y las conectivas proposicionales habituales: $\neg$, $\vee$, $\wedge$, $\rightarrow$ y $\leftrightarrow$.

Aunque, `nil` corresponde a $\bot$ y `t` a $\top$, semánticamente se interpreta `nil` como «falso» y cualquier otro término como «verdadero». Se equiparan así, semánticamente, los términos a las fórmulas.

Debido a esto existe una fuerte relación entre fórmulas y términos. El término τ es semánticamente equivalente a la fórmula $\neg(\tau = \bot)$. Este abuso del lenguaje permite emplear términos con la sintaxis concreta de ACL2 como sustitutos de las fórmulas. Toda fórmula se puede expresar como un término cuyo símbolo de función más externo es `not`, `or`, `and`, `implies`, `iff` o `equal`. Por ejemplo, a la fórmula lógica:

$$\neg(\neg(f(x,y) = \bot) \vee (f(y,x) = \top) = \bot)$$

le corresponden los siguientes términos en la sintaxis de ACL2:

```
(not (equal (or (not (equal (f x y) nil)) (equal (f y x) t)) nil))
(not (equal (or (f x y) (equal (f y x) t)) nil))
(or (f x y) (equal (f y x) t))
```

Obsérvese que la sintaxis de la lógica ACL2 es igual a la sintaxis del lenguaje de programación ACL2. De hecho, los términos también reciben el nombre de «expresiones». Del mismo modo, la semántica de la lógica se define para corresponder también a la del lenguaje de programación.

Axiomas y reglas de inferencia

Los *axiomas* incluyen a los de la lógica proposicional más los de la igualdad y una serie de axiomas sobre sus funciones primitivas que permiten trabajar con números (enteros, racionales y complejos racionales), caracteres, cadenas de caracteres, símbolos y listas.[2]

Las *reglas de inferencia* son las del cálculo proposicional con igualdad junto con instanciación de variables e inducción. Todas permiten deducir nuevos teoremas a partir de otros ya conocidos.

La *regla de instanciación de variables* permite sustituir variables en un teorema por términos para obtener otro teorema. La *regla de inducción* permite reducir la demostración de un teorema a un conjunto finito de casos empleando inducción sobre los ϵ_0-ordinales. La *regla de instanciación funcional* permite sustituir funciones en un teorema por otras para obtener un nuevo teorema.

Los axiomas y reglas de inferencias de la lógica de ACL2 que tratan la parte proposicional pertenecen, en esencia, a Shoenfield [95]. Se establece un

[2]Una extensión de ACL2, denominada ACL2(r), incluye a los números reales.

esquema de axiomas y cuatro reglas de inferencia, considerándose como primitivos los operadores lógicos de negación ($\neg$) y disyunción ($\vee$) y definiendo los restantes ($\wedge$, $\rightarrow$ y $\leftrightarrow$).

Una demostración formal puede representarse mediante un árbol finito de fórmulas, cada una de las cuales es un axioma o se deduce mediante una regla de inferencia de las anteriores. Un teorema es cualquier fórmula en una demostración.

Principios de extensión y de inducción

La lógica incluye también dos principios de extensión: *definición* y *encapsulado*. Ambos permiten añadir nuevos símbolos de función y axiomas a la lógica preservando su consistencia.

Además, se incluye un principio de inducción. Este principio junto con el de definición dependen de una estructura bien fundamentada: los ordinales de ACL2.

Estos ordinales corresponden a los ordinales menores que ϵ_0, ya que se demuestra que existe un isomorfismo entre ambos conjuntos. Algunos valores concretos aparecen en la tabla 2.1. Los ϵ_0-ordinales forman un conjunto bien fundamentado bajo una relación de orden apropiada.

ACL2 axiomatiza el reconocedor `e0-ordinalp` para los ϵ_0-ordinales, así como un orden `e0-ord-<` sobre ellos que corresponde con el orden usual de los ordinales de la teoría de conjuntos.

El principio de definición

El *principio de definición* es esencial, ya que permite introducir nuevos símbolos de función y asociarles (axiomáticamente) una definición.

Un aspecto importante es que la definición de una función introduce un nuevo axioma y, por consiguiente, ACL2 impone ciertas restricciones a las definiciones de las funciones para asegurar que no se introducen inconsistencias en la lógica. Para preservar la consistencia, el sistema únicamente admite una definición de una función bajo este principio si cumple lo siguiente:

1. Su nombre es un nuevo símbolo de función.
2. Sus parámetros formales son símbolos de variable distintos.

0	`0`
1	`1`
ω	`(1 . 0)`
$\omega + 1$	`(1 . 1)`
$\omega + 2$	`(1 . 2)`
$\omega \cdot 2$	`(1 1 . 0)`
$\omega \cdot 3 + 1$	`(1 1 1 . 1)`
ω^2	`(2 . 0)`
ω^3	`(3 . 0)`
ω^ω	`((1 . 0) . 0)`
$\omega^\omega + \omega^5 + \omega \cdot 2 + 3$	`((1 . 0) 5 1 1 . 3)`
ω^{ω^2}	`((2 . 0) . 0)`
ω^{ω^ω}	`(((1 . 0) . 0) . 0)`
ϵ_0	No existe

Tabla 2.1: Ordinales y su representación

3. Su cuerpo es un término cuyas únicas variables libres son los parámetros formales.
4. Una cierta medida ordinal de sus parámetros formales decrece estrictamente en cada llamada recursiva que aparezca en su cuerpo.

De todas las restricciones anteriores la más importante es la última: demostrar la terminación de la función cuando se aplica sobre cualquier objeto de ACL2; ya que, o esto se prueba o la función no será admitida.

Si ACL2 logra demostrar que una función es admisible, añade un axioma de definición que iguala la aplicación de la función a su cuerpo. Nótese que la admisibilidad de las funciones no recursivas es trivial.

En la práctica, si el usuario no suministra explícitamente la función de medida y la relación de orden bien fundamentada, el sistema emplea `acl2-count` y `e0-ord-<`, intentando averiguar un término en el que intervengan los parámetros de la función y cuyo tamaño decrezca estrictamente en cada llamada recursiva.

La función `acl2-count` calcula un número natural que representa, en cierto sentido, el tamaño de un objeto. Por ejemplo, el tamaño de un `cons` es uno más que la suma de los tamaños de su `car` y su `cdr`; o el tamaño de un entero es su valor absoluto.

El principio de encapsulado y la regla de instanciación funcional

El *principio de encapsulado* permite introducir nuevos símbolos de función restringidos por axiomas a tener ciertas propiedades. Para preservar la consistencia, el sistema exige que se proporcionen «testigos» de la existencia de tales funciones, es decir, un modelo de la teoría generada.

La instanciación funcional es una regla de inferencia derivada de este principio. El papel que juega el principio de encapsulado en el desarrollo estructurado de teorías se explica en [52].

El principio de inducción

Este principio está basado en el principio de inducción bien fundamentado y su corrección queda justificada por la buena fundamentación de `e0-ord-<` sobre los objetos que verifican `e0-ordinalp`.

Es habitual establecer el principio de inducción tomando la hipótesis de inducción sobre n y la tesis de inducción sobre $f(n)$, siendo f una función constructora. Esto puede reformularse de manera que la hipótesis de inducción se tome sobre $f^{-1}(n)$, siendo f^{-1} una función destructora, y la tesis de inducción sobre n.[3] Es esta última formulación la que emplea ACL2, ya que facilita la obtención de esquemas de inducción válidos a partir de la descomposición que realizan las funciones recursivas de sus parámetros.

En ACL2 no existe la inducción fuerte, pero es posible emplear un número arbitrario de hipótesis de inducción.

Equivalencias y congruencias

Las relaciones de equivalencia se introducen en ACL2 mediante funciones binarias para las que se han probado las propiedades de reflexividad, simetría y transitividad.

Para ello, normalmente, se utiliza el evento `defequiv` que se encarga de intentar demostrar que la función binaria que recibe como parámetro satisface las tres propiedades, y la incorpora en el sistema como una relación de equivalencia.

`(defequiv` *relación*`)`

[3] Esta formulación suele exigir una nueva hipótesis sobre n.

Por otra parte, el evento `defcong` demuestra que una relación de equivalencia preserva otra en un determinado parámetro de una función dada. Dada una función f y dos relaciones de equivalencia, *equiv1* y *equiv2*, la siguiente orden establece para f que, si se sustituye su i-ésimo parámetro por uno equivalente con respecto a *equiv1*, el resultado es equivalente al original con respecto a *equiv2*.

`(defcong` *equiv1 equiv2* `(`*f x1 ... xn*`)` *i*`)`

Otras características

La ausencia de cuantificación hace que el carácter de la lógica sea fundamentalmente constructivo: más que enunciar el hecho de que un cierto objeto existe, se ha de exponer una función computable que construya un objeto con las propiedades deseadas.

Otro aspecto interesante es la ausencia de tipos[4] y el carácter total de la lógica. Las funciones introducidas bajo el principio de definición han de ser funciones recursivas totales. Aunque esto último constituya una limitación teórica, en la práctica, la mayoría de las funciones parciales que solemos definir pueden ser generalizadas convenientemente para obtener funciones totales equivalentes bajo sus dominios de definición.

Posteriormente se pueden emplear protecciones para declarar el dominio que se pretende que dichas funciones tengan en ejecución. Las protecciones son un mecanismo extra-lógico, es decir, no afectan a las propiedades lógicas de las funciones.

Se puede emplear ACL2 para intentar demostrar dentro de la lógica que si la aplicación de una función no viola su protección, ninguna de las sucesivas llamadas a función de su cuerpo violan las suyas. A esto se le llama «verificar la protección». La verificación de la protecciones de una función garantiza la obtención de código LISP ejecutable eficientemente, y con los mismos resultados, en cualquier plataforma que esté acorde con COMMON LISP.

[4] Sin embargo, un sistema de deducción de tipos está presente internamente en el sistema.

2.2.3. ACL2 es un sistema de razonamiento automático

Cuando se suministra una conjetura a ACL2 para que la demuestre, o se extiende la lógica mediante uno de los principios de extensión, siendo necesario comprobar que se cumplen ciertas condiciones, ACL2 se comporta como un demostrador de teoremas.

Como tal, emplea una serie de *técnicas de prueba* en su búsqueda de una demostración de la conjetura. Cada técnica de prueba puede verse como un proceso que recibe una fórmula como entrada y produce un conjunto de fórmulas como resultado: la fórmula de entrada es un teorema *si* cada una de las resultantes lo es.

Por supuesto, puede que un proceso no sea aplicable a una fórmula. En tal caso, el proceso no falla, sino que produce un conjunto que contiene únicamente a dicha fórmula. Por otro lado, si un proceso es capaz de demostrar que una determinada fórmula es un teorema, devuelve el conjunto vacío.

Cuando el usuario proporciona una conjetura al sistema, su fórmula se convierte en el objetivo de la demostración y pasa secuencialmente por cada uno de los procesos hasta que uno de ellos sea aplicable. Éste produce un conjunto de subobjetivos que reemplazan al objetivo original y todo comienza de nuevo (manteniéndose una «bolsa» de objetivos pendientes).

Esto continúa hasta que no quedan subobjetivos pendientes de demostrar o hasta que se cumplen determinadas condiciones de terminación que indican que el sistema no puede completar la demostración de la conjetura (incluso, a veces, aparece un ciclo); en este último caso la conjetura inicial puede o no ser un teorema. Como se observa, el razonamiento es regresivo. En la siguiente sección se describe, brevemente, cada proceso.

Por otro lado, para llevar a cabo todo esto, el sistema toma información tanto del usuario como de una base de datos, llamada *mundo lógico* o simplemente *mundo*. El mundo incorpora las estrategias de demostración de teoremas desarrolladas por el usuario y codificadas como *reglas* que dirigen ciertos aspectos del comportamiento del sistema. Cuando intenta demostrar un teorema, el sistema aplica la estrategia del usuario, pudiendo utilizar también los consejos que se suministren con el teorema, y muestra su intento de demostración.

Es importante tener en cuenta que el usuario no tiene control sobre el comportamiento del demostrador una vez que empieza la demostración, excepto interrumpiéndola o abortándola. Por otro lado, en el caso que el demostrador

haya tenido éxito, incluye en el mundo nuevas reglas derivadas del teorema que se acaba de demostrar.

Órdenes básicas de demostración

Se puede intentar demostrar una conjetura empleando `defthm`. Si se tiene éxito, el teorema se añadirá en forma de reglas a la base de datos. La forma general de `defthm` es la siguiente:

```
(defthm nombre
  fórmula
  [:rule-classes clases de regla]
  [:hints consejos])
```

La orden anterior indica al sistema que demuestre la fórmula *fórmula*, le ponga al teorema el nombre *nombre* y construya con él reglas de las clases indicadas.

Las clases de regla se especifican mediante una clave o lista de claves apropiadas. Durante la demostración el sistema atenderá a los consejos suministrados.

Lo más importante a la hora de utilizar el demostrador es diseñar una estrategia de demostración «apropiada» y expresarla en función de reglas derivadas de teoremas. Hay diversas clases de regla. Las reglas más comunes son las de reescritura, las de definición y las de prescripción de tipos. Por omisión, las reglas son de reescritura. ACL2 emplea reescritura ordenada, proporcionando así un mecanismo que, bien empleado, permite evitar la aparición de ciclos de reescritura [62].

Cada regla producida tiene su propio estado: activada o desactivada. Las únicas reglas que el demostrador puede emplear implícitamente son las que se encuentran activas. La activación y desactivación de reglas se realiza con la orden `in-theory`, cuyo formato básico es:

```
(in-theory (disable lista de reglas))
(in-theory (enable lista de reglas))
```

También se puede emplear `deftheory` para definir teorías como conjuntos de reglas. Esto facilita su activación o desactivación en bloque.

(`deftheory` *nombre*
lista de reglas)

Técnicas de demostración

El demostrador está organizado como se muestra en la figura 2.1. En el centro está el conjunto de fórmulas a ser demostrado, y alrededor las seis *técnicas de demostración* o *procesos* que incorpora el demostrador en el orden en el que se van aplicando. Además, aunque no aparece en la figura, cada técnica de demostración puede emplear las reglas activas procedentes del mundo lógico.

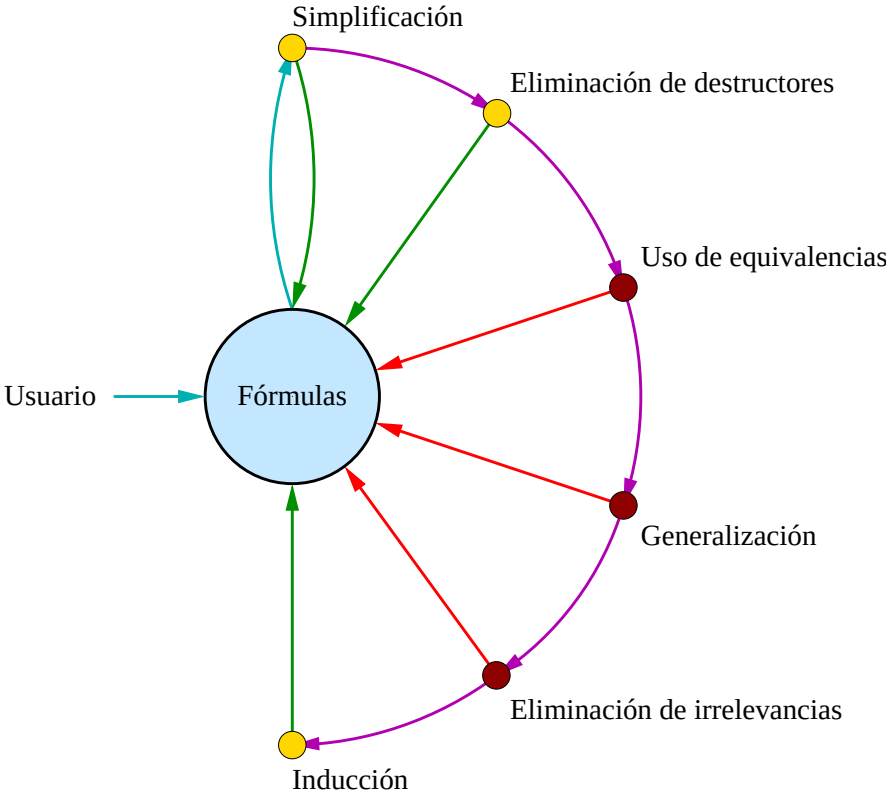

Figura 2.1: Procesos del demostrador de teoremas

Parte II

Polinomios

Capítulo 3

Anillos polinómicos

3.1. Introducción

A lo largo de los últimos cincuenta años se han construido diversos sistemas de cálculo simbólico cuyo objetivo es automatizar las necesidades crecientes de resolución de problemas matemáticos en las Ciencias y en las Ingenierías. A su vez, la popularidad de estos sistemas ha propiciado la aparición de nuevos algoritmos [18, 26].

No obstante, en el núcleo de todos y cada uno de estos sistemas se encuentra una solución, a menudo diferente, al problema de representar los distintos entes matemáticos y, muy en particular, los polinomios de múltiples variables y sus operaciones [102].

Algunos trabajos que tratan la formalización del concepto de polinomio dentro de los sistemas de razonamiento automático se encuentran en [2, 79]. Por otro lado, los intentos de formalización que se han llevado a cabo con más éxito, se han realizado sobre todo en el contexto de herramientas donde el nivel de automatización no es alto y el razonamiento está fundamentalmente dirigido por el usuario [3, 100].

Intentaremos cubrir esta carencia dotando a ACL2 de un libro reutilizable que permita estructurar ulteriores trabajos de verificación automática de algoritmos que manipulen polinomios. No obstante, éste no es el único objetivo puesto que no sólo se trata de encontrar una formalización adecuada para la demostración, sino que también nos permita realizar cálculos y computar con los algoritmos que la empleen.

En aras de la brevedad, no podemos incluir todos los detalles de las demostraciones presentadas. Remitimos al lector interesado al código ACL2 correspondiente en el directorio `polinomios`.

3.2. Representación de los polinomios

La representación de los objetos bajo estudio es una cuestión fundamental para el éxito de cualquier trabajo de certificación. Esto es especialmente importante cuando se trata de polinomios, por la gran variedad de representaciones que admiten y lo distinto de los algoritmos que las operan.

El grado de dificultad asociado a la demostración automática de una determinada propiedad depende en gran medida de la representación escogida. Comentemos esto con algo más de detalle analizando dos esquemas principales de representación explorados durante la realización de este trabajo.

3.2.1. Representación normalizada

Inicialmente se eligió para los polinomios una representación *dispersa* (donde no aparecen los monomios nulos) y *normalizada* (donde los monomios están en orden estrictamente decreciente) [18, 26, 102], en la que a cada polinomio se asocia una representación única donde no se representan los monomios de coeficiente nulo, ni aparecen monomios con idéntico término.

Con este enfoque, por otro lado habitual en la programación de sistemas de cálculo simbólico, se pueden conseguir algoritmos muy eficientes [36, 102] que operan con objetos normalizados produciendo como resultado objetos también normalizados. Para ello es necesaria la definición de un orden total estricto sobre los términos que componen los monomios, existiendo diversas posibilidades al respecto ampliamente comentadas en la literatura.

La principal ventaja de este método desde el punto de vista de la verificación radica en el hecho de que la equivalencia semántica se transforma en igualdad sintáctica, muy simple, y que viene representada en ACL2 mediante el predicado `equal`.

No es difícil formalizar esta representación de manera que sea aceptada por el sistema. Sin embargo, aparecen problemas realmente cuando se intentan demostrar propiedades tan elementales como la asociatividad de la suma.

El análisis de las demostraciones fallidas muestra cómo el inconveniente más importante de esta representación es el de complicar en exceso las definiciones de las operaciones. Cada operación ha de ocuparse del proceso de normalización y, en concreto, de mantener ordenados los monomios del polinomio resultante. Esto produce un gran impacto sobre la complejidad de las pruebas.

Desgraciadamente, todo apunta a que parece existir un compromiso entre eficiencia algorítmica y facilidad de verificación. Esto nos condujo a emplear otra representación algorítmicamente menos eficiente, pero en la que las propiedades sean más sencillas de verificar.

Posteriormente, podría utilizarse refinamiento y demostrar la equivalencia de los algoritmos empleados con versiones más eficientes. En pocas palabras, el problema se reduce a encontrar una función eficiente para cada función ineficiente de nuestra representación y a demostrar que son equivalentes. No obstante, no trataremos este problema en esta tesis.

3.2.2. Representación desnormalizada

Emplear una representación desnormalizada presenta algunos inconvenientes como que la igualdad es semántica, esto es, ha de operar con las clases de equivalencia inducidas por el proceso de normalización, y el demostrador no la maneja directamente.[1]

No obstante, también posee grandes ventajas, ya que evita a las operaciones el tener que trabajar en forma normal y, por tanto, sus definiciones se simplifican considerablemente. En consecuencia, también se hace más sencilla la demostración automática de sus propiedades.

Al separar el cálculo que hace el algoritmo del proceso de normalización, se concentra el problema de normalización en un único lugar: el predicado de igualdad. Por supuesto que la igualdad se ha complicado, pero en menor medida de lo que se complican las operaciones con la representación normalizada.

[1] Este problema puede paliarse en ACL2 mediante el empleo de *congruencias*. Otros sistemas, como NQTHM [4, 9] y COQ [22], no poseen esta posibilidad y es necesario crear *teoremas de compatibilidad* entre las operaciones y las relaciones de equivalencia para obtener algo similar. Véase cómo influye este problema en [100].

3.2.3. Elección de la representación

La alternativa elegida ha sido la de emplear, inicialmente, una representación dispersa y desnormalizada, para posteriormente obtener, encima de esta capa desnormalizada, polinomios normalizados, esto es, polinomios con una representación normalizada.

Las representaciones normalizada y desnormalizada tienen algo en común: ambas utilizan el concepto de *orden de monomios* (un orden estricto total sobre términos extendido a monomios). Este orden se emplea en diferentes lugares dependiendo de la representación. En una representación normalizada este orden se tiene en cuenta en cada una de las operaciones. Sin embargo, en una representación desnormalizada, el orden se utilizará en el predicado de equivalencia semántica.

Esta sencilla observación nos permite construir polinomios normalizados sobre una capa de polinomios desnormalizados.

Nótese que una representación densa (donde se aparecen también los monomios nulos) no sería en ningún caso apropiada, ya que no soluciona ninguno de los problemas planteados y es tremendamente ineficiente en espacio (sobre todo cuando el número de variables es elevado).

Para resolver el problema en ACL2, un polinomio se representará como una lista de monomios y se definirá un predicado de igualdad semántica que demostrará ser una relación de equivalencia. A continuación, se definirán las operaciones polinómicas principales y se demostrará que existe una congruencia entre cada una de estas operaciones (por cada uno de sus parámetros) y dicha relación de equivalencia. Finalmente, se demostrará que los polinomios con estas operaciones tienen estructura de anillo.

Cada monomio vendrá dado por un par (su coeficiente y el término que lo acompaña) y se definirá un predicado de igualdad semántica que permita decidir si dos monomios son o no iguales.[2] Como veremos, los términos se representarán mediante listas de exponentes, definiéndose sobre ellos la operación de producto y una relación de orden total.

[2]Nótese que esto no es posible hacerlo sintácticamente: dos monomios con coeficiente nulo se consideran iguales, independientemente de cuáles sean sus términos.

3.3. Anillos de polinomios

Recordaremos en primer lugar las definiciones de algunas estructuras algebraicas básicas que necesitaremos antes de definir qué es un anillo de polinomios.

Definición 3.1. Sea C un conjunto y $+$ una operación binaria en C. Se dice que $\langle C, + \rangle$ es un semigrupo si $+$ es asociativa respecto de C.

Definición 3.2. Sea C un conjunto, $+$ una operación binaria en C y $0 \in C$. Se dice que $\langle C, +, 0 \rangle$ es un monoide si $\langle C, + \rangle$ es un semigrupo y 0 es elemento neutro respecto de $+$. El monoide se dice conmutativo si $+$ lo es.

Definición 3.3. Sea C un conjunto, $+$ una operación binaria en C, $-$ una operación unaria en C (a la que se llama opuesto o inverso) y $0 \in C$. Decimos que $\langle C, +, -, 0 \rangle$ es un grupo si $\langle C, +, 0 \rangle$ es un monoide y para todo $c \in C$, $c + (-c) = (-c) + c = 0$. El grupo se dice conmutativo si $+$ lo es.

Definición 3.4. Sea C un conjunto, $+$ y $\cdot$ operaciones binarias en C, $-$ una operación unaria en C y $0 \in C$. Decimos que $\langle C, +, -, \cdot, 0 \rangle$ es un anillo si:

- $\langle C, +, -, 0 \rangle$ es un grupo conmutativo.
- $\langle C, \cdot \rangle$ es un semigrupo.
- $\cdot$ es distributiva respecto de $+$.

El anillo se dice conmutativo si $\cdot$ lo es.

Definición 3.5. Sea C un conjunto, $+$ y $\cdot$ dos operaciones binarias en C, $-$ una operación unaria en C y $0, 1 \in C$. Decimos que $\langle C, +, -, \cdot, 0, 1 \rangle$ es un anillo con identidad si:

- $\langle C, +, -, \cdot, 0 \rangle$ es un anillo.
- $\langle C, \cdot, 1 \rangle$ es un monoide.

Si $\cdot$ es conmutativa, se dice que $\langle C, +, -, \cdot, 0, 1 \rangle$ es un anillo conmutativo con identidad.

Por último, terminemos definiendo el concepto de anillo de polinomios.

Definición 3.6. Sea $\langle C, +, -, \cdot, 0, 1\rangle$ un anillo conmutativo con identidad. El anillo de polinomios en k indeterminadas $x_1, \ldots, x_k$ con coeficientes en C es el conjunto de expresiones formales que tienen la siguiente forma:

$$c_1 \cdot x_1^{e_{11}} \cdots x_k^{e_{1k}} + \cdots + c_m \cdot x_1^{e_{m1}} \cdots x_k^{e_{mk}} = \sum_{i=1}^{m} c_i \cdot x_1^{e_{i1}} \cdots x_k^{e_{ik}}$$

donde $c_i \in C$. Al conjunto C se le llama anillo de coeficientes.

Como se observa existe toda una serie de estructuras algebraicas que es necesario formalizar para llegar al concepto de polinomio. En este capítulo se ha desarrollado una teoría computacional en ACL2 de los polinomios de múltiples variables sobre un anillo de coeficientes. Esta formalización incluye sus operaciones y propiedades fundamentales que establecen, principalmente, la existencia de la estructura de anillo. El objetivo aquí ha sido desarrollar una biblioteca reutilizable sobre polinomios con la que realizar ulteriores trabajos.

Pese a lo que pueda pensarse, ni siquiera la demostración de las propiedades básicas resulta trivial. No parece que esto se deba a la simplicidad de la lógica ACL2. En [84] se reconoce que desarrollar la asociatividad del producto en MIZAR supuso un reto. Se sorprenden sus autores de que en tratados de Álgebra se deje la demostración como ejercicio o se desarrolle sólo en el caso univariable y de manera incompleta, recurriendo a un razonamiento por analogía. De igual forma, el autor de [101] vio incrementado su esfuerzo por problemas surgidos durante la formalización de los polinomios en COQ. Otra formalización del anillo de polinomios se encuentra en [3].

Idealmente, se dispondría de una serie de propiedades para trabajar con los polinomios olvidándonos de su representación. En nuestra experiencia, el conjunto de propiedades necesarias para verificar algoritmos de cierta entidad sobre polinomios supera con mucho a las propiedades básicas de anillo, lo que lleva a su continua extensión.

3.3.1. Coeficientes

Para llevar a cabo la abstracción en ACL2 (que se presenta en el fichero `coeficiente.lisp` y dentro del paquete `COE`) del anillo de coeficientes vamos a utilizar el principio de encapsulado cuya sintaxis viene precedida por la cabecera en donde se indican el prototipo de las funciones, es decir el tipo y número de parámetros que van a recibir y el tipo del valor de retorno; le

siguen las funciones testigo de la existencia de tales funciones; y finalmente los axiomas.

El *principio de encapsulado* permite introducir nuevos símbolos de función restringidos por axiomas a tener ciertas propiedades; para preservar la consistencia, el sistema exige que se proporcionen «testigos» de la existencia de tales funciones.

Así, la siguiente definición bajo el principio de encapsulado introduce una restricción llamada `restriccion` sobre un nuevo símbolo de función `f`.

```
(encapsulate (((f x1 ... xn) => *))
  (local (defun f (x1 ... xn) cuerpo))
  (defthm restriccion propiedad))
```

El anterior encapsulado es admisible si su definición de `f` con `defun` es admisible y la fórmula *propiedad* es un teorema del mundo lógico extendido con la definición de `f`.

Si es admisible, el encapsulado extiende la teoría añadiendo el axioma de restricción para `f`, *propiedad*.

Nótese que en la extensión resultante después de introducir el encapsulado anterior, `f` no está definida pero se sabe que satisface *propiedad*.

El problema que nos encontramos al hacer este tipo de abstracción es que se pierde el procedimiento de decisión para la aritmética lineal que incorpora el sistema. Por tanto, debemos proporcionar los teoremas suficientes para poder razonar sin él. De esta forma, además, de las propiedades del anillo hemos proporcionado un conjunto mínimo de teoremas que juntos van a cubrir el vacío dejado por el procedimiento de decisión.

En primer lugar, veamos la cabecera del encapsulado. Esta cabecera suministra la información sobre el prototipo de las funciones que se van a emplear: un reconocedor, `coeficientep`; las operaciones del anillo, suma (`+`), producto (`*`) y opuesto (`-`); y los elementos neutro de la suma y el producto, `nulo` y `identidad`, respectivamente.

```
(encapsulate
  ((coeficientep (a) boolean)
   (+ (a b) coeficiente)
   (- (a) coeficiente)
   (* (a b) coeficiente)
```

```
(nulo () coeficiente)
(identidad () coeficiente))
```

Asumiremos que $\langle coeficientep, +, -, *, nulo, identidad\rangle$ (que de aquí en adelante denotaremos por $\langle C, +, -, \cdot, 0, 1\rangle$) forma un anillo conmutativo con identidad.

A continuación es necesario proporcionar unos testigos de la existencia de tales funciones. En este caso se han tomado las operaciones análogas de los números racionales de ACL2. Nótese la utilización del paquete `LISP` para distinguir las operaciones matemáticas del sistema de las que estamos definiendo.

```
(local
  (defun coeficientep (a)
    (acl2-numberp a)))

(local
  (defun + (a b)
    (LISP::+ a b)))

(local
  (defun * (a b)
    (LISP::* a b)))

(local
  (defun - (a)
    (LISP::- a)))

(local
  (defun nulo ()
    0))

(local
  (defun identidad ()
    1))
```

Para la igualdad se utilizará la sintáctica.

```
(defmacro = (a b)
  `(equal ,a ,b))
```

Estructura de anillo conmutativo

En cuanto a los axiomas que establecen que los coeficientes tienen estructura de anillo conmutativo, deben estar:

- Los que establecen que $\langle C, +, -, 0\rangle$ es un grupo conmutativo (axiomas 3.12, 3.13, 3.14 y 3.15 que vienen a continuación).
- Los que establecen que $\langle C, \cdot, 1\rangle$ es un monoide conmutativo (axiomas 3.16, 3.17 y 3.18 que vienen a continuación).
- La distributividad del producto sobre la suma que completaría las propiedades del anillo (axiomas 3.19, que vienen a continuación).

Además, están los siguientes axiomas de tipo:

- **Axioma 3.7.** *Sea* $a,\ b \in C.\ a + b \in C.$ *En* ACL2,

```
(defthm coeficientep-+
  (implies (and (coeficientep a) (coeficientep b))
           (coeficientep (+ a b))))
```

- **Axioma 3.8.** *Sea* $a,\ b \in C.\ a \cdot b \in C.$ *En* ACL2,

```
(defthm coeficientep-*
  (implies (and (coeficientep a) (coeficientep b))
           (coeficientep (* a b))))
```

- **Axioma 3.9.** *Sea* $a \in C.\ -a \in C.$ *En* ACL2,

```
(defthm coeficientep--
  (implies (coeficientep a)
           (coeficientep (- a))))
```

- **Axioma 3.10.** $0 \in C.$ *En* ACL2,

```
(defthm coeficientep-nulo
  (coeficientep (nulo)))
```

- **Axioma 3.11.** $1 \in C.$ *En* ACL2,

```
(defthm coeficientep-identidad
  (coeficientep (identidad)))
```

Continuemos con los axiomas que establecen que $\langle C, +, -, 0\rangle$ es un grupo conmutativo.

- **Axioma 3.12.** *Sea* $a \in C$. $0 + a = a$. *En* ACL2,

```
(defthm |0 + a = a|
  (implies (coeficientep a)
           (= (+ (nulo) a) a)))
```

- **Axioma 3.13.** *Sean* $a, b, c \in C$. $(a + b) + c = a + (b + c)$. *En* ACL2,

```
(defthm |(a + b) + c = a + (b + c)|
  (implies (and (coeficientep a)
                (coeficientep b)
                (coeficientep c))
           (= (+ (+ a b) c) (+ a (+ b c)))))
```

- **Axioma 3.14.** *Sean* $a, b \in C$. $a + b = b + a$. *En* ACL2,

```
(defthm |a + b = b + a|
  (implies (and (coeficientep a) (coeficientep b))
           (= (+ a b) (+ b a))))
```

- **Axioma 3.15.** *Sea* $a \in C$. $a + (-a) = 0$. *En* ACL2,

```
(defthm |a + (- a) = 0|
  (implies (coeficientep a)
           (= (+ a (- a)) (nulo))))
```

Seguiremos con los axiomas que establecen que $\langle C, \cdot, 1\rangle$ es un monoide conmutativo.

- **Axioma 3.16.** *Sea* $a \in C$. $1 \cdot a = a$. *En* ACL2,

```
(defthm |1 * a = a|
  (implies (coeficientep a)
           (= (* (identidad) a) a)))
```

- **Axioma 3.17.** *Sean* $a, b, c \in C$. $(a \cdot b) \cdot c = a \cdot (b \cdot c)$. *En* ACL2,

```
(defthm |(a * b) * c = a * (b * c)|
  (implies (and (coeficientep a)
                (coeficientep b)
                (coeficientep c))
           (= (* (* a b) c) (* a (* b c)))))
```

- **Axioma 3.18.** *Sean* $a, b \in C$. $a \cdot b = b \cdot a$. *En* ACL2,

```
(defthm |a * b = b * a|
  (implies (and (coeficientep a) (coeficientep b))
           (= (* a b) (* b a))))
```

Por último, se tiene la distributividad del producto sobre la suma que completaría las propiedades del anillo.

- **Axioma 3.19.** *Sean* $a, b, c \in C$. $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$. *En* ACL2,

```
(defthm |a * (b + c) = (a * b) + (a * c)|
  (implies (and (coeficientep a)
                (coeficientep b)
                (coeficientep c))
           (= (* a (+ b c)) (+ (* a b) (* a c)))))
```

Con esto terminamos el encapsulado de coeficientes. Nótese que estos axiomas se introducen en ACL2 como teoremas ya que se demuestran directamente con la implementación de los números racionales que ya incorpora el sistema.

Además también son necesarios entre otros los siguientes teoremas.

Teorema 3.20. *Sean* $a, b, c \in C$. $a + c = b + c \iff a = b$. *En* ACL2,

```
(defthm |a + c = b + c <=> a = b|
  (implies (and (coeficientep a)
                (coeficientep b)
                (coeficientep c))
           (iff (= (+ a c) (+ b c)) (= a b))))
```

Demostración. Por los axiomas 3.12, 3.13, 3.14 y 3.15. □

Teorema 3.21. *Sean* $a, b \in C$. $a + b = b \iff a = 0$. *En* ACL2,

```
(defthm |a + b = b <=> a = 0|
  (implies (and (coeficientep a) (coeficientep b))
           (iff (= (+ a b) b) (= a (nulo)))))
```

Demostración. Por el axioma 3.12 y el teorema 3.20. □

Teorema 3.22. *Sea* $a, b \in C$. $a + b = 0 \iff b = -a$. *En* ACL2,

```
(defthm |a + b = 0 <=> b = - a|
  (implies (and (coeficientep a) (coeficientep b))
           (iff (= (+ a b) (nulo)) (= b (- a)))))
```

Demostración.

$$\begin{aligned} & a + b = 0 \\ \iff & a + b = a + (-a) && \text{(axioma 3.15)} \\ \iff & b + a = -a + a && \text{(axioma 3.14)} \\ \iff & b = -a && \text{(axioma 3.20)} \end{aligned}$$

□

Teorema 3.23. *Sean* $a, b, c \in C$. $a + (b + c) = b + (a + c)$. *En* ACL2,

```
(defthm |a + (b + c) = b + (a + c)|
  (implies (and (coeficientep a)
                (coeficientep b)
                (coeficientep c))
           (= (+ a (+ b c)) (+ b (+ a c)))))
```

Demostración. Por los axiomas 3.14 y 3.13. □

Teorema 3.24. *Sea* $a, b \in C$. $-(a + b) = (-a) + (-b)$. *En* ACL2,

```
(defthm |- (a + b) = (- a) + (- b)|
  (implies (and (coeficientep a) (coeficientep b))
           (= (- (+ a b)) (+ (- a) (- b)))))
```

Demostración. Por $a + b = 0 \iff b = -a$ (teorema 3.22) basta demostrar que $((-a) + (-b)) + (a + b) = 0$, que se demuestra como sigue:

$$
\begin{aligned}
&(-a + -b) + (a + b) \\
&= a + ((-a + -b) + b) && \text{(teorema 3.23)} \\
&= a + (-a + (-b + b)) && \text{(axioma 3.13)} \\
&= a + (-a + 0) && \text{(axioma 3.14 y 3.15)} \\
&= a + -a && \text{(axioma 3.12 y 3.14)} \\
&= 0 && \text{(axioma 3.15)}
\end{aligned}
$$

□

Teorema 3.25. *Sea* $a \in C$. $0 \cdot a = 0$. *En* ACL2,

```
(defthm |0 * a = 0|
  (implies (coeficientep a)
           (= (* (nulo) a) (nulo))))
```

Demostración. Por $a + b = b \iff a = 0$ (teorema 3.21) basta probar que $(0 \cdot a) + (a \cdot 0) = a \cdot 0$, que se demuestra como sigue

$$
\begin{aligned}
&(0 \cdot a) + (a \cdot 0) \\
&= (a \cdot 0) + (a \cdot 0) && \text{(axioma 3.18)} \\
&= a \cdot (0 + 0) && \text{(axioma 3.19)} \\
&= a \cdot 0 && \text{(axioma 3.12)}
\end{aligned}
$$

□

3.3.2. Términos

En esta sección se presenta la formalización de términos realizada en el fichero `termino.lisp` dentro del paquete `TER`.

Sea un conjunto finito y no vacío de variables $X = \{x_1, \ldots, x_k\}$, con una relación de orden $<_X = \{(x_i, x_j) : 1 \leq i < j \leq k\}$ entre sus elementos.

En lo que sigue, consideraremos fijado este conjunto X de variables.

Definición 3.26. Un término sobre X es un producto finito de potencias de la forma:

$$x_1^{e_1} \dots x_k^{e_k} = \prod_{i=1}^{k} x_i^{e_i} \qquad \forall i\ e_i \in \mathbb{N},$$

al que, abreviadamente, denotaremos por $X^{[e_1,\dots,e_k]}$.

El resultado principal que se obtiene en esta sección es que los términos forman un monoide conmutativo con la operación producto.

Siguiendo el mismo esquema que con los coeficientes y así obtener el monoide conmutativo de términos abstractos, realizamos la implementación mediante un encapsulado. Veamos la cabecera del encapsulado. Esta cabecera suministra la información sobre el prototipo de las funciones que se van a emplear: un reconocedor, `terminop`; la operación del monoide, el producto $(\cdot)$, y el elemento neutro del producto, `uno`.

```
(encapsulate
  ((terminop (a) boolean)
   (* (a b) termino)
   (uno () termino))
```

Demostraremos que $\langle terminop, *, uno\rangle$ (que de aquí en adelante denotaremos por $\langle T[X], \cdot, 1\rangle$) forma un monoide conmutativo.

A continuación es necesario proporcionar los testigos de la existencia de tales funciones. Un término sobre X se puede representar de manera sencilla mediante una lista de números naturales. Su reconocedor sería el siguiente predicado:

```
  (local
    (defun terminop (a)
      (if (atom a)
          (equal a nil)
        (and (naturalp (first a)) (terminop (rest a))))))
```

La función `naturalp` (definida fuera del encapsulado) comprueba si su parámetro es un número natural.

```
(defmacro naturalp (x)
  `(and (integerp ,x) (LISP::<= 0 ,x)))
```

Representamos el término uno mediante `nil`.

```
(local
  (defun uno ()
    nil))
```

Con las definiciones realizadas queda claro que únicamente es necesario definir una igualdad meramente sintáctica sobre los términos. No obstante, por comodidad notacional, y para facilitar cambios futuros, definimos el símbolo de igualdad como sinónimo de `equal`.

```
(defmacro = (a b)
  `(equal ,a ,b))
```

Una vez fijado el conjunto de variables, para calcular el producto de dos términos basta sumar sus exponentes variable a variable.

Definición 3.27. Sean $X^{[a_1,\ldots,a_k]}$, $X^{[b_1,\ldots,b_k]} \in T[X]$.

$$X^{[a_1,\ldots,a_k]} \cdot X^{[b_1,\ldots,b_k]} = X^{[a_1+b_1,\ldots,a_k+b_k]}$$

La siguiente función realiza esta tarea en ACL2 y se tomará como testigo del producto.

```
(local
  (defun * (a b)
    (cond ((and (not (terminop a)) (not (terminop b)))
           (uno))
          ((not (terminop a)) b)
          ((not (terminop b)) a)
          ((endp a) b)
          ((endp b) a)
          (t
           (cons (LISP::+ (first a) (first b))
                 (* (rest a) (rest b)))))))
```

Como se observa, los elementos que no son términos se comportan como si se correspondieran con el término uno. En el caso de términos de distinto número de variables se completa el que posee menor número de variables; esto equivale a suponer que la menor de las listas se rellena de ceros por la derecha.

Estructura de monoide conmutativo

La demostración de que los términos tienen estructura de monoide conmutativo respecto a la operación producto se obtiene suministrando al sistema los siguientes axiomas.

- **Axioma 3.28.** *Sea* a, $b \in T[X]$. $a \cdot b \in T[X]$. *En* ACL2,

```
(defthm terminop-*
  (implies (and (terminop a) (terminop b))
           (terminop (* a b))))
```

- **Axioma 3.29.** $1 \in T[X]$. *En* ACL2,

```
(defthm terminop-uno
  (terminop (uno)))
```

- **Axioma 3.30.** *Sea* $a \in T[X]$. $1 \cdot a = a$. *En* ACL2,

```
(defthm |1 * a = a|
  (implies (terminop a)
           (= (* (uno a) a)))
```

- **Axioma 3.31.** *Sean* $a, b \in T[X]$. $a \cdot b = b \cdot a$. *En* ACL2,

```
(defthm |a * b = b * a|
  (implies (and (terminop a) (terminop b))
           (= (* a b) (* b a))))
```

- **Axioma 3.32.** *Sean* $a, b, c \in C$. $(a \cdot b) \cdot c = a \cdot (b \cdot c)$. *En* ACL2,

```
(defthm |(a * b) * c = a * (b * c)|
  (implies (and (terminop a) (terminop b) (terminop c))
           (= (* (* a b) c) (* a (* b c)))))
```

3.3.3. Monomios

En esta sección se presenta la formalización de monomios realizada en el fichero `monomio.lisp` dentro del paquete `MON`.

Definición 3.33. Un monomio sobre X es un producto de la forma $c \cdot X^{[e_1,\ldots,e_k]}$, donde a c se le denomina coeficiente y a los e_i, exponentes. La operación $\cdot$ se define entre el conjunto de coeficientes y el de los valores que pueden tomar los elementos de X.

Notaremos por $M_C[X]$ al conjunto de monomios.

Claramente, para la representación de un monomio basta emplear una lista cuyo primer elemento sea su coeficiente y su resto el término acompañante. El constructor y las operaciones accesoras se definen como:

```
(defun monomio (c e)
  (cons c e))

(defun coeficiente (a)
  (if (not (monomiop a))
      (COE::nulo)
    (first a)))

(defun termino (a)
  (if (or (not (consp a))
          (not (terminop (rest a))))
      (TER::uno)
    (rest a)))
```

Es preciso también definir un reconocedor que permita distinguir qué objetos de ACL2 son monomios y cuáles no:

```
(defun monomiop (a)
  (and (consp a)
       (coeficientep (first a))
       (terminop (rest a))))
```

El monomio unidad y el monomio nulo de término uno se definen mediante las siguientes funciones.

```
(defun identidad ()
  (monomio (COE::identidad) (TER::uno)))

(defun nulo ()
  (monomio (COE::nulo) (TER::uno)))
```

Definición 3.34. Un monomio es nulo si su coeficiente lo es. En ACL2,

```
(defun nulop (a)
  (COE::= (coeficiente a) (COE::nulo)))
```

La igualdad sería en este caso semántica.

Definición 3.35. Dos monomios son semánticamente iguales si ambos son nulos, o si tienen iguales coeficientes y términos. En ACL2,

```
(defun = (a b)
  (or (and (not (monomiop a)) (not (monomiop b)))
      (and (monomiop a) (monomiop b)
           (nulop a) (nulop b))
      (and (monomiop a) (monomiop b)
           (COE::= (coeficiente a) (coeficiente b))
           (TER::= (termino a) (termino b)))))
```

Teorema 3.36. *La igualdad de monomios es una relación de equivalencia. En* ACL2,

```
(defequiv =)
```

Demostración. Inmediata por su definición y porque las igualdades de coeficientes y términos son relaciones de equivalencias. □

Por último se definen las funciones suma y opuesto. Nótese que la suma de monomios sólo tiene sentido cuando sus términos son iguales.

Definición 3.37. Sean $c_1 \cdot X^{[e_1,\ldots,e_k]}$, $c_2 \cdot X^{[e_1,\ldots,e_k]} \in M_C[X]$.

$$c_1 \cdot X^{[e_1,\ldots,e_k]} + c_2 \cdot X^{[e_1,\ldots,e_k]} = (c_1 + c_2) \cdot X^{[e_1,\ldots,e_k]}$$

En ACL2,

```
(defun + (a b)
  (monomio (COE::+ (coeficiente a) (coeficiente b))
           (termino a)))
```

Definición 3.38. Sea $c \cdot X^{[e_1,\ldots,e_k]} \in M_C[X]$.

$$-(c_1 \cdot X^{[e_1,\ldots,e_k]}) = (-c_1) \cdot X^{[e_1,\ldots,e_k]}$$

En ACL2,

```
(defun - (a)
  (monomio (COE::- (coeficiente a)) (termino a)))
```

Teorema 3.39. *Sea* $a \in M_C[X]$. $a + (-a) = 0$. *En* ACL2,

```
(defthm |a + (- a) = 0|
  (implies (monomiop a)
           (= (+ a (- a)) (nulo))))
```

Demostración. Inmediata por las definiciones de la suma y el opuesto, y por los teoremas 3.12 y 3.15. □

Teorema 3.40. *Sea* $a, b \in M_C[X]$. $-(a + b) = (-a) + (-b)$. *En* ACL2,

```
(defthm |- (a + b) = (- a) + (- b)|
  (implies (and (monomiop a) (monomiop b))
           (= (- (+ a b)) (+ (- a) (- b)))))
```

Demostración. Por definición del opuesto y el teorema 3.24. □

Estructura de monoide conmutativo

Para calcular el producto de dos monomios basta multiplicar sus coeficientes y sus términos.

Definición 3.41. Sean $c_1 \cdot X^{[a_1,\ldots,a_k]}$, $c_2 \cdot X^{[b_1,\ldots,b_k]} \in M_C[X]$.

$$c_1 \cdot X^{[a_1,\ldots,a_k]} \cdot c_2 \cdot X^{[b_1,\ldots,b_k]} = (c_1 \cdot c_2) \cdot X^{[a_1+b_1,\ldots,a_k+b_k]}$$

En ACL2,

```
(defun * (a b)
  (monomio (COE::* (coeficiente a) (coeficiente b))
           (TER::* (termino a) (termino b))))
```

Demostraremos a continuación que $\langle monomiop, *, identidad\rangle$ (que lo notaremos por $\langle M_C[X], \cdot, 1\rangle$) forma un monoide conmutativo.

Los monomios heredan directamente la estructura de monoide conmutativo de los términos y de las propiedades de la operación producto del anillo de coeficientes. Demostraremos a continuación cada uno de los teoremas.

Teorema 3.42. *Sea $a \in M_C[X]$. $1 \cdot a = a$. En* ACL2*,*

```
(defthm |1 * a = a|
  (implies (monomiop a)
           (= (* (identidad) a) a)))
```

Demostración. Por las definiciones del producto y la igualdad de monomios, y los axiomas 3.16 y 3.30. □

Teorema 3.43. *Sean $a, b \in M_C[X]$. $a \cdot b = b \cdot a$. En* ACL2*,*

```
(defthm |a * b = b * a|
  (implies (and (monomiop a) (monomiop b))
           (= (* a b) (* b a))))
```

Demostración. Por las definiciones del producto y la igualdad de monomios, y los axiomas 3.18 y 3.31. □

Teorema 3.44. *Sean $a, b, c \in M_C[X]$. $(a \cdot b) \cdot c = a \cdot (b \cdot c)$. En* ACL2*,*

```
(defthm |(a * b) * c = a * (b * c)|
  (implies (and (monomiop a) (monomiop b) (monomiop c))
           (= (* (* a b) c) (* a (* b c)))))
```

Demostración. Por las definiciones del producto y la igualdad de monomios, y los axiomas 3.17 y 3.32. □

Congruencias

Por último, se demuestra también que la igualdad semántica es congruente con la operación de producto en ambos parámetros.

Teorema 3.45. *La igualdad es congruente respecto del producto.*

Sean a, a', b, $b' \in M_C[X]$.

- $a = a' \implies a \cdot b = a' \cdot b$
- $b = b' \implies a \cdot b = a \cdot b'$

Demostración. Por las definiciones del producto y la igualdad de monomios, los axiomas 3.18 y 3.31, y el teorema 3.25. □

ACL2 también soporta reescritura basada en congruencias. Esto permite sustituciones de equivalentes no sólo de iguales. Para esto habrá que definir, primero, relaciones de equivalencias y, segundo, demostrar las reglas de congruencias que permitan sustituir equivalentes, utilizando `defcong`.

Los teoremas correspondientes en ACL2 serían los siguientes.

```
(defcong = = (* a b) 1)

(defcong = = (* a b) 2)
```

3.3.4. Polinomios

En esta sección se presenta la formalización de polinomios realizada en el fichero `polinomio.lisp` dentro del paquete `POL`.

Definición 3.46. Un polinomio sobre k indeterminadas $X = \{x_1, \ldots, x_k\}$ con coeficientes en C es una suma finita de monomios de la siguiente forma:

$$c_1 \cdot X^{[e_{11},\ldots,e_{1k}]} + \cdots + c_m \cdot X^{[e_{m1},\ldots,e_{mk}]} = \sum_{i=1}^{m} c_i \cdot X^{[e_{i1},\ldots,e_{ik}]}$$

donde $c_i \in C$, y si $m = 0$ decimos que se trata del polinomio nulo y lo notamos por 0. Al anillo de polinomios lo llamaremos $C[X]$.

Un polinomio se representa en ACL2 simplemente mediante una lista de monomios.

$$[(c_1, [e_{11}, \ldots, e_{1k}]), \ldots, (c_m, [e_{m1}, \ldots, e_{mk}])]$$

```
(defun polinomiop (p)
  (if (atom p)
      (equal p nil)
    (and (monomiop (primero p))
         (polinomiop (resto p)))))
```

El polinomio nulo sin monomios se define como una macro y es reconocido por otra apropiada para su empleo en los casos base de la recursividad.

```
(defmacro nulo () nil)

(defmacro nulop (p)
  `(endp ,p))
```

Las accesoras al primer monomio del polinomio y al resto del polinomio se definen como sigue. Nótese que a partir de aquí, dado un polinomio, p, notaremos por $mp(p)$ y $resto(p)$ al primero y al resto de p, respectivamente. También, notaremos por $cp(p)$ y $tp(p)$ al coeficiente y al término de $mp(p)$, respectivamente. Además, dado un monomio m, notaremos por $cp(m)$ y $tp(m)$ al coeficiente y al término, respectivamente, asociados al monomio.

```
(defmacro primero (p)
  `(first ,p))

(defmacro resto (p)
  `(rest ,p))
```

Obsérvese que las operaciones que definiremos sobre los polinomios estarán generalizadas para manipular correctamente cualquier objeto de ACL2, aunque no sea un polinomio. Un objeto no polinómico será tratado como el polinomio nulo. Gracias a esto es posible enunciar la mayoría de los teoremas de congruencia con las operaciones, ya que `defcong` no permite el establecimiento de ninguna hipótesis restrictiva sobre el carácter de los objetos implicados.

No conviene olvidar que esto no impide en ningún modo que se especifiquen las protecciones adecuadas al carácter de cada función, puesto que éstas carecen de significado lógico. Las versiones ejecutables de las funciones podrán ser así más eficientes, ya que se les permite suponer que reciben objetos polinómicos[3].

Igualdad semántica

Para decidir si dos polinomios son iguales semánticamente, debemos comprobar si ambos pertenecen a la misma clase de equivalencia. Esto se consigue calculando sus formas normales (es decir, representantes canónicos de sus respectivas clases de equivalencia) y comprobando si ambas son iguales sintácticamente. En esta sección se presenta la formalización de la igualdad semántica realizada en el fichero `forma-normal.lisp` dentro del paquete `POL`.

Definición 3.47. Se dice que un polinomio está ordenado o normalizado si cumple las siguientes condiciones:

1. Sus monomios aparecen ordenados estrictamente en orden decreciente de término.
2. Entre sus monomios no aparece ninguno nulo.

Nótese que la primera condición implica la no existencia de monomios con idénticos términos dentro de un polinomio ordenado.

```
(defun ordenadop (p)
  (and (polinomiop p)
       (or (nulop p)
           (and (not (MON::nulop (primero p)))
                (termino-mayor-termino-principal (primero p)
                                                 (resto p))
                (ordenadop (resto p))))))
```

donde `termino-mayor-termino-principal` permite averiguar si un monomio debe preceder al monomio principal de un polinomio ordenado.

[3]En general, si una operación se ejecuta fuera de su dominio en un sistema LISP sin comprobación de protecciones en tiempo de ejecución, su comportamiento es dependiente del sistema.

```
(defmacro termino-mayor-termino-principal (m p)
  `(or (nulop ,p)
       (TER::< (termino (primero ,p)) (termino ,m))))
```

La función `TER::<` representa un orden total estricto sobre términos, $<_t$, y será descrito en detalle en el capítulo siguiente cuando se traten los órdenes polinómicos y su buena fundamentación. No obstante, adelantaremos su definición matemática.

Definición 3.48. Sean $[a_1, \ldots, a_k]$ y $[b_1, \ldots, b_k] \in T[X]$, se define el orden, $<_T$, sobre términos de la siguiente forma:

$$[a_1, \ldots, a_k] <_T [b_1, \ldots, b_k] \equiv \exists i \ (a_i < b_i \wedge \forall j < i \ a_j = b_j)$$

A continuación presentamos cómo se puede obtener la forma normal de un polinomio dado. Para ello se definen previamente dos funciones: una que suma un monomio a un polinomio y otra que lo suma de forma que si el polinomio que recibe como parámetro está en forma normal el resultado también lo estará.

Definición 3.49. Sea $m \in M_C[X]$ y $p \in C[X]$ donde

$$m = a \cdot X^{[e_{11},\ldots,e_{1k}]}$$
$$p = a_1 \cdot X^{[r_{11},\ldots,r_{1k}]} + \cdots + a_m \cdot X^{[r_{m1},\ldots,r_{mk}]}$$

entonces

$$m +_M p = a \cdot X^{[e_{11},\ldots,e_{1k}]} + a_1 \cdot X^{[r_{11},\ldots,r_{1k}]} + \cdots + a_m \cdot X^{[r_{m1},\ldots,r_{mk}]}$$

La función correspondiente en ACL2 es `+M`. Esta función se implementa de la forma más sencilla: consiste, simplemente, en añadir un monomio a un polinomio utilizando la función `cons`, aunque se define de manera que trate los casos anómalos de forma razonable. Esto es esencial para posteriormente poder definir congruencias.

```
(defun +M (m p)
  (cond ((and (not (monomiop m)) (not (polinomiop p)))
         (nulo))
        ((not (polinomiop p))
         (list m))
        ((not (monomiop m))
         p)
        (t
         (cons m p))))
```

Definición 3.50. Sea $m \in M_C[X]$ y $p \in C[X]$ donde

$$m = a \cdot X^{[e_{11},\ldots,e_{1k}]}$$
$$p = a_1 \cdot X^{[r_{11},\ldots,r_{1k}]} + \cdots + a_m \cdot X^{[r_{m1},\ldots,r_{mk}]}$$

entonces

$$m +^{<}_{M} p = \begin{cases} p, & m = 0 \\ m, & p = 0 \\ m +_M p, & tp(p) <_T tp(m) \\ resto(p), & tp(p) = tp(m) \wedge cp(m) = -cp(p) \\ (m + mp(p)) +_M resto(p), & tp(p) = tp(m) \wedge cp(m) \neq -cp(p) \\ mp(p) +_M (m +^{<}_{M} resto(p)), & \text{e.o.c.} \end{cases}$$

Nótese que $+^{<}_{M}$ suma un monomio a un polinomio de forma que si el polinomio que recibe como parámetro está en forma normal el resultado también lo estará.

Este concepto se implementará en ACL2 mediante la función `+-monomio`.

```
(defun +-monomio (m p)
  (cond ((and (not (monomiop m)) (not (polinomiop p)))
         (nulo))
        ((not (monomiop m))
         p)
        ((and (not (polinomiop p)) (MON::nulop m))
         (nulo))
        ((not (polinomiop p))
         (+M m (nulo)))
        ((MON::nulop m)
         p)
        ((nulop p)
         (+M m (nulo)))
        ((TER::= (termino m) (termino (primero p)))
         (let ((c (COE::+ (coeficiente m)
                          (coeficiente (primero p)))))
           (if (COE::= c (COE::nulo))
               (resto p)
             (+M (MON::+ (primero p) m) (resto p)))))
        ((TER::< (termino (primero p)) (termino m))
```

```
 (+M m p))
(t
 (+M (primero p) (+-monomio m (resto p))))))
```

Definición 3.51. Dado un polinomio p, definimos su forma normal, $fn(p)$, de la siguiente forma:

$$fn(p) = \begin{cases} 0, & p = 0 \\ mp(p) +^{<}_{M} fn(resto(p)) & \text{e.o.c.} \end{cases}$$

En ACL2,

```
(defun fn (p)
  (cond ((or (not (polinomiop p)) (nulop p))
         (nulo))
        (t
         (+-monomio (primero p) (fn (resto p))))))
```

Es decir, si el polinomio es nulo, ya está en forma normal; basta pues normalizar el resto del polinomio si éste no es nulo y añadir al resultado su primer monomio mediante la función ACL2 `+-monomio`.

El siguiente teorema demuestra que realmente la función `fn` devuelve un polinomio ordenado o normalizado.

```
(defthm ordenadop-fn
  (ordenadop (fn p)))
```

Definición 3.52. Se dice que dos polinomios, p y q son semánticamente iguales si sus formas normales son sintácticamente iguales. En ACL2,

```
(defun = (p q)
  (equal (fn p) (fn q)))
```

Una vez hecho esto, se demuestra que la relación de igualdad definida entre los polinomios es una equivalencia.

Teorema 3.53. *La igualdad de polinomios es una relación de equivalencia. En* ACL2,

```
(defequiv =)
```

Demostración. Inmediata por su definición. □

Nótese que en lo que sigue, consideraremos que el símbolo = denota igualdad semántica.

También se demuestran otras propiedades importantes, como que el orden de las operaciones no altera la suma de monomios a un polinomio ordenado.

Esta última propiedad tiene una prueba ACL2 bastante extensa debido a la cantidad de casos que se generan y juega un papel fundamental en la demostración de la propiedad conmutativa de la suma de polinomios. Nótese que, de hecho, la igualdad en este caso no es sólo semántica sino también sintáctica ($=_e$), como puede verse a continuación. A partir de aquí se distingue la igualdad semántica ($=$) de la sintáctica ($=_e$).

Lema 3.54. *Sean m_1, $m_2 \in M_C[X]$, $p \in C[x]$ ordenado.*

$$m_1 +_M^< (m_2 +_M^< p) =_e m_2 +_M^< (m_1 +_M^< p)$$

En ACL2,

```
(defthm |m1 +Mo (m2 +Mo p) =e m2 +Mo (m1 +Mo p)|
  (implies (ordenadop p)
           (equal (+-monomio m1 (+-monomio m2 p))
                  (+-monomio m2 (+-monomio m1 p)))))
```

Demostración. Por inducción sobre la estructura de la función $+_M^<$.

Caso 1: $m_1 = 0 \vee m_2 = 0$

Inmediato por definición de $+_M^<$.

Caso 2: $p = 0$

$$m_1 +_M^< (m_2 +_M^< p) =_e m_1 +_M^< m_2 =_e m_2 +_M^< m_1 =_e m_2 +_M^< (m_1 +_M^< p)$$

por definición de $+_M^<$.

Caso 3: $tp(m_1) = tp(mp(p))$

Caso 3.1: $tp(m_1) <_T tp(m_2)$

$$m_1 +_M^< (m_2 +_M^< p) =_e m_1 +_M^< (m_2 +_M p) =_e$$
$$m_2 +_M (m_1 +_M^< p) =_e m_2 +_M^< (m_1 +_M^< p)$$

por definición de $+_M^<$ y y $+_M$.

Caso 3.2: $tp(m_1) = tp(m_2)$

$$m_1 +^<_M (m_2 +^<_M p) =_e m_1 +^<_M ((m_2 + mp(p)) +_M resto(p)) =_e$$
$$(m_1 + (m_2 + mp(p))) +_M resto(p) =_e$$
$$(m_2 + (m_1 + mp(p))) +_M resto(p) =_e$$
$$m_2 +^<_M ((m_1 + mp(p)) +_M resto(p)) =_e m_2 +^<_M (m_1 +^<_M p)$$

por definición de $+^<_M$ y $+_M$, y los teoremas 3.43 y 3.44.

Caso 3.3: $tp(m_2) <_T tp(m_1)$

$$m_1 +^<_M (m_2 +^<_M p) =_e m_1 +^<_M (mp(p) +_M (m_2 +^<_M resto(p))) =_e$$
$$(m_1 + mp(p)) +_M (m_2 +^<_M resto(p))) =_e$$
$$m_2 +^<_M ((m_1 + mp(p)) +_M resto(p)) =_e m_2 +^<_M (m_1 +^<_M p)$$

por definición de $+^<_M$ y $+_M$.

Caso 4: $tp(mp(p)) <_T tp(m_1)$

Caso 4.1: $tp(m_1) <_T tp(m_2)$

$$m_1 +^<_M (m_2 +^<_M p) =_e m_1 +^<_M (m_2 +_M p) =_e$$
$$m_2 +_M (m_1 +^<_M p) =_e m_2 +^<_M (m_1 +^<_M p)$$

por definición de $+^<_M$ y $+_M$.

Caso 4.2: $tp(m_1) = tp(m_2)$

$$m_1 +^<_M (m_2 +^<_M p) =_e m_1 +^<_M (m_2 +_M p) =_e$$
$$(m_1 + m_2) +_M p =_e (m_2 + m_1) +_M p =_e m_2 +^<_M (m_1 +^<_M p)$$

por definición de $+^<_M$ y $+_M$, y por el teorema 3.43.

Caso 4.3: $tp(m_2) <_T tp(m_1)$

$$m_1 +^<_M (m_2 +^<_M p) =_e m_1 +_M (m_2 +^<_M p) =_e$$
$$m_2 +^<_M (m_1 +_M p) =_e m_2 +^<_M (m_1 +^<_M p)$$

por definición de $+^<_M$ y $+_M$.

Paso de inducción: $tp(m_1) <_T tp(mp(p))$

Por hipótesis de inducción:

$$m_1 +^<_M (m_2 +^<_M resto(p)) =_e m_2 +^<_M (m_1 +^<_M resto(p))$$

entonces

Caso 1: $tp(m_2) <_T tp(mp(p))$

$$\begin{array}{ll} m_1 +_M^< (m_2 +_M^< p) =_e & \\ mp(p) +_M (m_1 +_M^< (m_2 +_M^< resto(p))) =_e & \text{(def. de } +_M^< \text{ y } +_M) \\ mp(p) +_M (m_2 +_M^< (m_1 +_M^< resto(p))) =_e & \text{(hipótesis de inducción)} \\ m_2 +_M^< (m_1 +_M^< p) & \text{(def. de } +_M^< \text{ y } +_M) \end{array}$$

Caso 2: $tp(m_2) = tp(mp(p))$

$$\begin{array}{l} m_1 +_M^< (m_2 +_M^< p) =_e \\ m_1 +_M^< ((m_2 + mp(p)) +_M resto(p)) =_e \\ (m_2 + mp(p)) +_M (m_1 +_M^< resto(p)) =_e \\ m_2 +_M^< (mp(p) +_M (m_1 +_M^< resto(p))) =_e \\ m_2 +_M^< (m_1 +_M^< p) \end{array}$$

por definición de $+_M^<$ y $+_M$.

Caso 3: $tp(mp(p)) <_T tp(m_2)$

$$m_1 +_M^< (m_2 +_M^< p) =_e m_2 +_M (m_1 +_M^< p) =_e m_2 +_M^< (m_1 +_M^< p)$$

por definición de $+_M^<$ y $+_M$.

□

Lema 3.55. *Sean m_1, $m_2 \in M_C[X]$ y $p \in C[x]$ ordenado.*

$$m_1 +_M^< (m_2 +_M^< p) = m_2 +_M^< (m_1 +_M^< p)$$

En ACL2,

```
(defthm |m1 +Mo (m2 +Mo p) = m2 +Mo (m1 +Mo p)|
  (implies (ordenadop p)
           (= (+-monomio m1 (+-monomio m2 p))
              (+-monomio m2 (+-monomio m1 p)))))
```

Demostración. Inmediata por la definición de $=$ y por el teorema 3.54. □

Lema 3.56. *Sean m_1, $m_2 \in M_C[X]$ y $p \in C[x]$ y p está ordenado.*

$$m_1 +_M (m_2 +_M p) = m_2 +_M (m_1 +_M p)$$

En ACL2,

```
(defthm |m1 +M (m2 +M p) = m2 +M (m1 +M p)|
  (implies (ordenadop p)
           (= (+M m1 (+M m2 p)) (+M m2 (+M m1 p)))))
```

Demostración. Inmediata por la definición de $=$, $+_M$ y por el teorema 3.54. □

Congruencias de la suma de monomio y polinomio

Uno de los aspectos más interesantes de la formalización elegida es que permite definir congruencias entre la relación de equivalencia dada por la igualdad de polinomios bajo forma normal y sus operaciones. Esta característica aumenta notablemente las posibilidades de reutilización del libro como herramienta de demostración de propiedades de más alto nivel.

La primera operación con la que podemos establecer congruencias es la suma de monomio y polinomio. En este caso intervienen dos relaciones de equivalencia: las definidas sobre los monomios y los polinomios.

Teorema 3.57. *La igualdad es congruente respecto de* $+_M$.

Sean m, $m' \in M_C[X]$, *y* p, $p' \in C[X]$.

$$m = m' \implies m +_M p = m' +_M p$$
$$p = p' \implies m +_M p = m +_M p'$$

En ACL2,

```
(defcong MON::= = (+M m p) 1)
(defcong = = (+M m p) 2)
```

Demostración. Inmediata por la definición de $+_M$, la igualdad de monomios y la igualdad de polinomios. □

Estructura de anillo conmutativo

A continuación se definirán las operaciones que nos permitirán sumar y multiplicar polinomios y calcular el opuesto de un polinomio. Para asegurarnos

de que estas operaciones cumplen, con la representación escogida para los polinomios, las propiedades fundamentales que se espera de ellas, se debe demostrar que dotan a $C[X]$ de una estructura de anillo conmutativo.

Para esto es necesario comprobar que los polinomios con la suma y el opuesto forman un grupo abeliano, mientras que con el producto forman un monoide conmutativo; además el producto debe distribuir sobre la suma.

Grupo conmutativo con la suma y el opuesto

En esta sección se presenta las formalizaciones de la suma y el opuesto realizadas en los ficheros `suma.lisp` y `opuesto.lisp` dentro del paquete `POL`.

Definición 3.58. Sean $p, q \in C[X]$ donde

$$p = a_1 \cdot X^{[r_{11},\ldots,r_{1k}]} + \cdots + a_m \cdot X^{[r_{m1},\ldots,r_{mk}]}$$
$$q = b_1 \cdot X^{[s_{11},\ldots,s_{1k}]} + \cdots + b_m \cdot X^{[s_{m1},\ldots,s_{mk}]}$$

entonces

$$p + q = a_1 \cdot X^{[r_{11},\ldots,r_{1k}]} + \cdots + a_m \cdot X^{[r_{m1},\ldots,r_{mk}]} + b_1 \cdot X^{[s_{11},\ldots,s_{1k}]} + \cdots + b_m \cdot X^{[s_{m1},\ldots,s_{mk}]}$$

Para sumar dos polinomios basta concatenar sus listas de monomios. En realidad, ésta es la forma más sencilla de definir esta operación y presenta la ventaja de simplificar mucho la demostración de la propiedad asociativa. Si lo que se desea es obtener el resultado reducido, basta con calcular su forma normal.

```
(defun + (p q)
  (cond ((and (not (polinomiop p)) (not (polinomiop q)))
         (nulo))
        ((not (polinomiop p))
         q)
        ((not (polinomiop q))
         p)
        (t
         (append p q))))
```

Definición 3.59. Sea $p = c_1 \cdot X^{[e_{11},\ldots,e_{1k}]} + \cdots + c_m \cdot X^{[e_{m1},\ldots,e_{mk}]} \in C[X]$ entonces

$$-p = (-c_1) \cdot X^{[e_{11},\ldots,e_{1k}]} + \cdots + (-c_m) \cdot X^{[e_{m1},\ldots,e_{mk}]}$$

Para calcular el opuesto de un polinomio únicamente es necesario sustituir en cada monomio el coeficiente con su opuesto.

```
(defun - (p)
  (cond ((or (not (polinomiop p)) (nulop p))
         (nulo))
        (t
         (+M (MON::- (primero p)) (- (resto p))))))
```

Los siguientes teoremas (3.60, 3.64, 3.61, 3.66 y 3.67) demuestran que los polinomios con las operaciones antedichas tienen estructura de grupo conmutativo.

Teorema 3.60. *Sea* $p \in C[X]$. $0 + p = p$. *En* ACL2,

```
(defthm |0 + p = p|
  (= (+ (nulo) p) p))
```

Demostración. Inmediata por la definición de la suma y la definición de la igualdad. □

Lema 3.61. *Sean* $p \in C[X]$. $p + 0 = p$. *En* ACL2,

```
(defthm |p + 0 = p|
  (= (+ p (nulo)) p))
```

Demostración. Por inducción y por las definiciones de la igualdad y la suma. □

Lema 3.62. *Sean* $m \in M_C[X]$, p, $q \in C[X]$. $(m +_M p) + q =_e m +_M (p + q)$. *En* ACL2,

```
(defthm |(m +M p) + q =e m +M (p + q)|
  (equal (+ (+M m p) q) (+M m (+ p q))))
```

Demostración. Inmediata por la definiciones de $+$ y $+_M$. □

Lema 3.63. *Sean* p, $q \in C[X]$. $p + q =_e mp(p) +_M (resto(p) + q)$. *En* ACL2,

```
(defthm |p + q =e mp(p) +M (resto(p) + q)|
  (implies (and (polinomiop p) (not (nulop p)))
           (equal (+ p q) (+M (primero p) (+ (resto p) q)))))
```

Demostración. Inmediata por la definiciones de + y $+_M$. □

Teorema 3.64. *Sean p, q, r* $\in C[X]$. $(p+q)+r = p+(q+r)$. *En* ACL2,

```
(defthm |(p + q) + r = p + (q + r)|
  (= (+ (+ p q) r) (+ p (+ q r))))
```

Demostración. Por inducción según el siguiente esquema. Sea $P(p,q,r) \equiv (p+q)+r = p+(q+r)$:

1. $p = 0 \rightarrow P(p,q,r)$
2. $(p \neq 0 \wedge P(resto(p),q,r)) \rightarrow P(p,q,r)$

Caso base: $p = 0$

Inmediata por la definición de + y =, y por el teorema 3.60.

Paso de inducción:

Por hipótesis de inducción:

$$(resto(p)+q)+r = resto(p)+(q+r)$$

entonces

$$\begin{array}{ll}
(p+q)+r = & \\
(mp(p) +_M (resto(p)+q))+r = & \text{(lema 3.63)} \\
mp(p) +_M ((resto(p)+q)+r) = & \text{(lema 3.62)} \\
mp(p) +_M (resto(p)+(q+r)) = & \text{(hip. ind. y teo. 3.57)} \\
p+(q+r) & \text{(lema 3.63)}
\end{array}$$

□

La conmutatividad del grupo es más complicada de demostrar que las restantes propiedades. Para ello, demostramos también el lema 3.65.

Lema 3.65. *Sean q,* $p \in C[X]$ *y* $p \neq 0$. *Entonces,* $q + p = mp(p) +_M (q + resto(p))$. *En* ACL2,

```
(defthm |q + p = mp(p) +M (q + resto(p))|
  (implies (and (polinomiop q)
                (polinomiop p)
                (not (nulop p)))
           (= (+ q p)
              (+M (primero p) (+ q (resto p))))))
```

Demostración. Por inducción según el siguiente esquema. Sea $P(q,p) \equiv q + p = mp(p) +_M (q + resto(p))$:

1. $q = 0 \rightarrow P(q,p)$
2. $(q \neq 0 \wedge P(resto(q),p)) \rightarrow P(q,p)$

Caso base: $q = 0$

Inmediata por la definición de $=$, $+_M$, y por el teorema 3.60.

Paso de inducción:

Por hipótesis de inducción:

$$resto(q) + p = mp(p) +_M (resto(q) + resto(p))$$

entonces

$$\begin{array}{ll} q + p = & \\ mp(q) +_M (resto(q) + p) = & \text{(lema 3.63)} \\ mp(q) +_M (mp(p) +_M (resto(q) + resto(p))) = & \text{(hip. ind. y teo 3.57)} \\ mp(p) +_M (mp(q) +_M (resto(q) + resto(p))) = & \text{(lema 3.56)} \\ mp(p) +_M (q + resto(p)) & \text{(lema 3.63)} \end{array}$$

□

Teorema 3.66. *Sean* p, $q \in C[X]$. $p + q = q + p$. *En* ACL2,

```
(defthm |p + q = q + p|
  (= (+ p q) (+ q p)))
```

Demostración. Por inducción según el siguiente esquema. Sea $P(p,q) \equiv p + q = q + p$:

1. $(p = 0 \vee q = 0) \rightarrow P(p, q)$
2. $(p \neq 0 \wedge q \neq 0 \wedge P(resto(p), q)) \rightarrow P(p, q)$

Caso base: $p = 0 \vee q = 0$

Inmediata por la definición de $=$, y por los teoremas 3.60 y 3.61.

Paso de inducción:

Por hipótesis de inducción:

$$resto(p) + q = q + resto(p)$$

entonces

$$\begin{array}{ll} p + q = & \\ mp(p) +_M (resto(p) + q) = & \text{(lema 3.63)} \\ mp(p) +_M (q + resto(p)) = & \text{(hip. ind. y teo. 3.57)} \\ q + p & \text{(lema 3.65)} \end{array}$$

□

Teorema 3.67. *Sea $p \in C[X]$. Entonces, $p + (-p) = 0$. En* ACL2,

```
(defthm |p + (- p) = 0|
  (= (+ p (- p)) (nulo)))
```

Demostración. Por inducción según el siguiente esquema. Sea $P(p) \equiv p + (-p) = 0$:

1. $p = 0 \rightarrow P(p)$
2. $(p \neq 0 \wedge P(resto(p))) \rightarrow P(p)$

Caso base: $p = 0$

Inmediata por la definición de $=$, $-$ y $+$, y por el teorema 3.60.

Paso de inducción:

Por hipótesis de inducción:

$$resto(p) + (-resto(p)) = 0$$

entonces

$$
\begin{array}{ll}
p + (-p) = & \\
mp(p) +_M (resto(p) + (-p)) = & \text{(lema 3.63)} \\
mp(p) +_M (resto(p) + ((-mp(p)) +_M (-resto(p)))) = & \text{(def. de } -) \\
mp(p) +_M (((-mp(p)) +_M (-resto(p))) + resto(p)) = & \text{(teo. 3.66 y 3.57)} \\
mp(p) +_M ((-mp(p)) +_M (resto(p) + (-resto(p)))) = & \text{(lema 3.62)} \\
mp(p) +_M ((-mp(p)) + 0) = & \text{(h.i. y teo. 3.57)} \\
mp(p) + (-mp(p)) = & \text{(df.}+_M \text{ y t. 3.61)} \\
0 & \text{(teo. 3.39)}
\end{array}
$$

□

Congruencias de la suma

La congruencia de la igualdad respecto de la suma se ha formalizado en el fichero `congruencias-suma.lisp` dentro del paquete `POL`.

Para demostrar que la igualdad es congruente respecto de la suma se demuestra primero el siguiente lema.

Lema 3.68. *Sean* p, $q \in C[X]$. *Entonces,* $p + q = p + fn(q)$. *En* ACL2,

```
(defthm |p + q = p + fn(q)|
  (implies (syntaxp (not (and (consp q) (eq (primero q) 'fn))))
           (= (+ p q) (+ p (fn q)))))
```

Este teorema produce en ACL2 una reescritura infinita:

`(+ p q)` $\longrightarrow$ `(+ p (fn q))` $\longrightarrow$ `(+ p (fn (fn q)))` $\longrightarrow \cdots$

Para evitar esto, se introducen restricciones sintácticas a la aplicabilidad de la regla mediante `syntaxp`. De esta forma, un término `(+ p q)` sólo será reescrito por estas reglas si el término `q` no es de la forma `(fn` x`)`.

Demostración. Por inducción según el siguiente esquema. Sea $P(p, q) \equiv p + q = p + fn(q)$:

1. $p = 0 \to P(p, q)$
2. $(p \neq 0 \wedge P(resto(p), q)) \to P(p, q)$

Caso base: $p = 0$

Inmediata por la definición de $=$ y $+$.

Paso de inducción:

Inmediato por la hipótesis de inducción y las definiciones de $+$ y $=$.

□

Teorema 3.69. *La igualdad es congruente respecto de la suma.*

Sean p, p', q, $q' \in C[X]$.

- $p = p' \implies p + q = p' + q$
- $q = q' \implies p + q = p + q'$

En ACL2,

```
(defcong = = (+ p q) 1)
(defcong = = (+ p q) 2)
```

Demostración. Inmediata por la definición de $=$ y por los teoremas 3.66 y 3.68. □

Congruencia del opuesto de la suma

Para demostrar que la igualdad de polinomios es congruente con la operación opuesto es necesario primero demostrar el lema 3.71.

Lema 3.70. *Sean* $m \in M_C[X]$ *y* $p \in C[X]$.

$$-(m +^{<}_{M} p) =_e (-m) +^{<}_{M} (-p)$$

En ACL2 queda de la siguiente forma. Como ya se ha comentado la igualdad sintáctica en ACL2 se representa por `equal`.

```
(defthm |- (m +Mo p) = (- m) +Mo (- p)|
  (implies (and (monomiop m) (polinomiop p))
           (equal (- (+-monomio m p))
                  (+-monomio (MON::- m) (- p)))))
```

Demostración. Por inducción según el siguiente esquema. Sea $P(m,p) \equiv -(m +_M^< p) =_e (-m) +_M^< (-p)$:

1. $p = 0 \rightarrow P(m,p)$
2. $(p \neq 0 \wedge P(m, resto(p))) \rightarrow P(m,p)$

Caso base: $p = 0$

Inmediata por la definición de $+_M^<$, la del opuesto de monomios y la del opuesto de polinomios.

Paso de inducción:

Por hipótesis de inducción:

$$-(m +_M^< resto(p)) =_e (-m) +_M^< (-resto(p))$$

entonces

Caso 1: $m = 0$

Inmediata por la definición de $+_M^<$, la del opuesto de monomios y la del opuesto de polinomios.

Caso 2: $tp(mp(p)) <_T tp(m)$

$$\begin{array}{ll} -(m +_M^< p) =_e & \\ -(m +_M p) =_e & (\text{def. de } +_M^<) \\ (-m) +_M (-p) =_e & (\text{def. de } - \text{ y } +_M) \\ (-m) +_M^< (-p) & (\text{def. de } +_M^<) \end{array}$$

Caso 3: $tp(m) <_T tp(mp(p))$

$$\begin{array}{ll} -(m +_M^< p) =_e & \\ -(mp(p) +_M (m +_M^< resto(p))) =_e & (\text{def. de } +_M^<) \\ (-mp(p)) +_M (-(m +_M^< resto(p))) =_e & (\text{def. de } - \text{ y } +_M) \\ (-mp(p)) +_M ((-m) +_M^< (-resto(p))) =_e & (\text{hip. de inducción}) \\ (-m) +_M^< (-p) & (\text{def. de } +_M^<) \end{array}$$

Caso 4: $tp(m) = tp(mp(p))$

$$
\begin{array}{ll}
-(m +^{<}_{M} p) =_e & \\
-((m + mp(p)) +_M resto(p)) =_e & (\text{def. de } +^{<}_{M}) \\
-(m + mp(p)) +_M (-resto(p)) =_e & (\text{def. de } - \text{ y } +_M) \\
((-m) + (-mp(p))) +_M (-resto(p)) =_e & (\text{teo. } 3.40) \\
(-m) +^{<}_{M} ((-mp(p)) +_M (-resto(p))) =_e & (\text{def. de } +^{<}_{M}) \\
(-m) +^{<}_{M} (-p) & (\text{def. de } +_M)
\end{array}
$$

□

Lema 3.71. *Sea $p \in C[X]$.*

$$fn(-p) =_e -fn(p)$$

En ACL2,

```
(defthm |fn(- p) = - fn(p)|
  (equal (fn (- p)) (- (fn p))))
```

Demostración. Por inducción, con el siguiente esquema. Sea $P(p) \equiv fn(-p) = -fn(p)$:

1. $p = 0 \to P(p)$
2. $(p \neq 0 \wedge P(resto(p))) \to P(p)$

Caso base: $p = 0$

Inmediata por la definición de $-$ y fn.

Paso de inducción:

Por hipótesis de inducción:

$$fn(-resto(p)) =_e -fn(resto(p))$$

entonces

$$\begin{aligned}
&fn(-p) =_e \\
&(-mp(p) +^<_M fn(-resto(p)) =_e && \text{(def. de } - \text{ y } fn) \\
&(-mp(p) +^<_M -fn(resto(p)) =_e && \text{(hip. ind.)} \\
&-(mp(p) +^<_M fn(resto(p))) =_e && \text{(lema 3.70)} \\
&-fn(p) && \text{(def. de } fn)
\end{aligned}$$

□

Teorema 3.72. *La igualdad es congruente respecto del opuesto.*

Sean p, $p' \in C[X]$. *Entonces,* $p = p' \implies -p = -p'$. *En* ACL2,

```
(defcong = = (- p) 1)
```

Demostración. Inmediata por la definición de la igualdad de polinomios y por el lema 3.71. □

Propiedades adicionales de la suma y el opuesto

Finalmente, se demuestra que el opuesto de una suma es la suma de los opuestos.

Lema 3.73. *Sean* p, q *y* $r \in C[X]$.

$$p + r = q + r \iff p = q$$

En ACL2,

```
(defthm |p + r = q + r <=> p = q|
  (implies (and (polinomiop p) (polinomiop q))
           (iff (= (+ p r) (+ q r)) (= p q))))
```

Demostración.

$$\begin{aligned}
&p + r = q + r \\
&\iff (p + r) + (-r) = (q + r) + (-r) && \text{(teo. 3.66 y 3.69)} \\
&\iff (p + (r + (-r)) = q + (r + (-r)) && \text{(teo. 3.64)} \\
&\iff p + 0 = q + 0 && \text{(teo. 3.67)} \\
&\iff p = q && \text{(teo. 3.61)}
\end{aligned}$$

□

Lema 3.74. *Sean p, $q \in C[X]$.*

$$p + q = 0 \iff q = -p$$

En ACL2,

```
(defthm |p + q = 0 <=> q = - p|
  (implies (and (polinomiop p) (polinomiop q))
           (and (iff (= (+ p q) (nulo)) (= q (- p))))))
```

Demostración.

$$\begin{aligned}
& p + q = 0 \\
& \iff p + q = p + (-p) && \text{(teorema 3.67)} \\
& \iff q + p = (-p) + p && \text{(teorema 3.66)} \\
& \iff q = -p && \text{(teorema 3.73)}
\end{aligned}$$

□

Lema 3.75. *Sean p, q y $r \in C[X]$.*

$$p + (q + r) = q + (p + r)$$

En ACL2,

```
(defthm |p + (q + r) = q + (p + r)|
  (= (+ p (+ q r)) (+ q (+ p r))))
```

Demostración.

$$\begin{aligned}
& p + (q + r) \\
& = (p + q) + r && \text{(teorema 3.64)} \\
& = (q + p) + r && \text{(teorema 3.66)} \\
& = q + (p + r) && \text{(teorema 3.64)}
\end{aligned}$$

□

Teorema 3.76. *Sean p, $q \in C[X]$.*

$$-(p + q) = (-p) + (-q)$$

En ACL2,

```
(defthm |- (p + q) = (- p) + (- q)|
  (= (- (+ p q)) (+ (- p) (- q))))
```

Demostración. Por el teorema 3.74 ($p+q=0 \iff q=-p$) basta demostrar que $((-p)+(-q))+(p+q)=0$, que se demuestra como sigue

$$\begin{aligned}
&((-p)+(-q))+(p+q) \\
&= p+(((-p)+(-q))+q) && \text{(teorema 3.75)} \\
&= p+((-p)+((-q)+q)) && \text{(teorema 3.64)} \\
&= (p+((-p)+0) && \text{(teorema 3.67)} \\
&= p+(-p) && \text{(teorema 3.61)} \\
&= 0 && \text{(teorema 3.67)}
\end{aligned}$$

□

Monoide conmutativo con el producto

En esta sección se presenta la formalización del producto de polinomios realizada en el fichero `producto.lisp` dentro del paquete `POL`.

El polinomio unitario en forma normal en ACL2 se define a continuación. Los elementos de su clase de equivalencia pueden ser reconocidos por una sencilla macro.

```
(defmacro identidad ()
  `(+M (MON::identidad) (nulo)))

(defmacro identidadp (p)
  `(= ,p (identidad)))
```

Antes de definir la operación de producto entre polinomios es factible definir una función auxiliar que represente el producto entre monomios y polinomios.

Definición 3.77. Sean $m \in M_C[X]$ y $p = m_1 + \cdots + m_n \in C[K]$ entonces

$$m \cdot_M p = m \cdot m_1 +_M (\cdots +_M m \cdot m_n)$$

En ACL2,

```
(defun *-monomio (m p)
  (cond ((or (nulop p) (not (monomiop m)) (not (polinomiop p)))
         (nulo))
        (t
         (+M (MON::* m (primero p))
             (*-monomio m (resto p))))))
```

Se demuestra que esta operación tiene elemento unidad (`|1 *M p = p|`), que `|m = 0 => m *M p = 0|` y que es distributiva la suma de polinomios (`|m *M (p + q) = (m *M p) + (m *M q)|`).

A continuación, para calcular el producto de dos polinomios procedemos de la siguiente forma.

Definición 3.78. Sean $p = m_{11} + \cdots + m_{1n}$ y $q = m_{21} + \cdots + m_{2r} \in C[K]$ entonces

$$\begin{aligned} p \cdot q = & m_{11} \cdot m_{21} + \cdots + m_{11} \cdot m_{2r} + \\ & \cdots \\ & m_{1n} \cdot m_{21} + \cdots + m_{1n} \cdot m_{2r} \end{aligned}$$

La definición en ACL2 se haría recursivamente utilizando las funciones que calculan el producto de dos monomios y el producto de un monomio por un polinomio.

```
(defun * (p q)
  (cond ((or (nulop p) (not (polinomiop p)))
         (nulo))
        (t
          (+ (*-monomio (primero p) q)
             (* (resto p) q)))))
```

Que los polinomios con esta operación de producto tienen estructura de monoide conmutativo se deduce de los teoremas 3.80, 3.87 y 3.88.

El primero que vamos a ver es el teorema 3.80 que establece la existencia de elemento neutro para el producto de polinomios. Ya que el producto de polinomios se define a partir del producto de un monomio por un polinomio, tendremos que demostrar el siguiente lema.

Lema 3.79. *Sea $p \in C[X]$. Entonces, $1 \cdot_M p = p$. En* ACL2,

```
(defthm |1 *M p = p|
  (implies (polinomiop p)
           (= (*-monomio (MON::identidad) p) p)))
```

Demostración. Por inducción sobre la estructura de la función $\cdot_M$ y por el teorema 3.42. □

Teorema 3.80. *Sea $p \in C[X]$. Entonces, $1 \cdot p = p$. En* ACL2,

```
(defthm |1 * p = p|
  (= (* (identidad) p) p))
```

Demostración.

$$1 \cdot p = 1 \cdot_M p = p$$

por definición de $\cdot$ y por el lema 3.79. □

Las propiedades de existencia de un elemento cancelador no presentan dificultad para el demostrador.

Teorema 3.81. *Sea $p \in C[X]$. Entonces, $0 \cdot p = 0$. En* ACL2

```
(defthm |0 * p = 0|
  (= (* (nulo) p) (nulo)))
```

Demostración. Inmediata por la definición del producto. □

A continuación se demuestra que el producto de polinomios es distributivo respecto a la suma que hace uso del teorema 3.83.

Lema 3.82. *Sean $m1$, $m_2 \in M_C[X]$ y, $p \in C[X]$. Entonces,*

$$m_1 \cdot_M (m_2 +_M p) =_e (m_1 \cdot m_2) +_M m_1 \cdot_M p.$$

```
(defthm |m1 *M (m2 +M p) =e (m1 * m2) +M (m1 *M p)|
  (implies (and (monomiop m1) (monomiop m2))
           (equal (*-monomio m1 (+M m2 p))
                  (+M (MON::* m1 m2) (*-monomio m1 p)))))
```

Demostración. Inmediata por las definiciones de $\cdot_M$ y $+_M$. □

Teorema 3.83. *Sean $m \in M_C[X]$ y, p, $q \in C[X]$. Entonces,*

$$m \cdot_M (p + q) = m \cdot_M p + m \cdot_M q.$$

En ACL2,

```
(defthm |m *M (p + q) = (m *M p) + (m *M q)|
  (= (*-monomio m (+ p q))
     (+ (*-monomio m p) (*-monomio m q))))
```

Demostración. Por inducción según el siguiente esquema. Sea $P(p, q, r) \equiv m \cdot_M (p + q) = m \cdot_M p + m \cdot_M q$:

1. $p = 0 \to P(m, p, q)$
2. $(p \neq 0 \wedge P(m, resto(p), q)) \to P(m, p, q)$

Caso base: $p = 0$

Inmediata por la definición de $\cdot_M$ y el teorema 3.60.

Paso de inducción:

Por hipótesis de inducción:

$$m \cdot_M (resto(p) + q) = m \cdot_M resto(p) + m \cdot_M q$$

entonces

$$\begin{array}{ll}
m \cdot_M (p + q) = & \\
m \cdot_M (mp(p) +_M (resto(p) + q)) = & \text{(teo. 3.63)} \\
m \cdot mp(p) +_M (m \cdot_M (resto(p) + q)) = & \text{(teo. 3.82)} \\
m \cdot mp(p) +_M (m \cdot_M resto(p) + m \cdot_M q) = & \text{(hip. ind. y teo 3.57)} \\
(m \cdot mp(p) +_M m \cdot_M resto(p)) + m \cdot_M q = & \text{(teo. 3.62)} \\
m \cdot_M p + m \cdot_M q & \text{(def. de } \cdot_M\text{)}
\end{array}$$

□

Teorema 3.84. *Sea p, q, $r \in C[X]$.*

$$(p + q) \cdot r = p \cdot r + q \cdot r$$

En ACL2,

```
(defthm |(p + q) * r = (p * r) + (q * r)|
  (= (* (+ p q) r) (+ (* p r) (* q r))))
```

Demostración. Por inducción según el siguiente esquema. Sea $P(p,q,r) \equiv (p+q)\cdot r = p\cdot r + q\cdot r$:

1. $p = 0 \rightarrow P(p,q,r)$
2. $(p \neq 0 \wedge P(resto(p),q,r)) \rightarrow P(p,q,r)$

Caso base: $p = 0$

Inmediata por la definición de $\cdot$ y por los teoremas 3.60 y 3.81.

Paso de inducción:

Por hipótesis de inducción:

$$(resto(p)+q)\cdot r = resto(p)\cdot r + q\cdot r$$

$(p+q)\cdot r =$	
$(mp(p) +_M (resto(p)+q))\cdot r =$	(lema 3.63)
$mp(p) \cdot_M r + (resto(p)+q)\cdot r =$	(def. de $+_M$ y $\cdot$)
$mp(p) \cdot_M r + (resto(p)\cdot r + q\cdot r) =$	(hip. ind. y teo 3.69)
$(mp(p) \cdot_M r + resto(p)\cdot r) + q\cdot r =$	(teo. 3.64)
$p\cdot r + q\cdot r$	(def. de $\cdot_M$)

□

A partir del teorema anterior y del lema 3.86 se demuestra la asociatividad del producto (teorema 3.87). Para el lema 3.86 hace falta el siguiente.

Lema 3.85. *Sean m_1, $m_2 \in M_C[X]$ y, p, $q \in C[X]$.*

$$m_1 \cdot_M (m_2 \cdot_M p) = (m_1 \cdot m_2) \cdot_M p$$

En ACL2,

```
(defthm |m1 *M (m2 *M p) = (m1 * m2) *M p|
  (implies (and (monomiop m1) (monomiop m2))
           (= (*-monomio m1 (*-monomio m2 p))
              (*-monomio (MON::* m1 m2) p))))
```

Demostración. Por inducción según el siguiente esquema. Sea $P(m_1, m_2, p) \equiv m_1 \cdot_M (m_2 \cdot_M p) = (m_1 \cdot m_2) \cdot_M p$:

1. $p = 0 \rightarrow P(m_1, m_2, p)$
2. $(p \neq 0 \wedge P(m_1, m_2, resto(p))) \rightarrow P(m_1, m_2, p)$

Caso base: $p = 0$

Inmediata por la definición de $\cdot_M$.

Paso de inducción:

Por hipótesis de inducción:

$$m_1 \cdot_M (m_2 \cdot_M resto(p)) = (m_1 \cdot m_2) \cdot_M resto(p)$$

$m_1 \cdot_M (m_2 \cdot_M p) =$	
$m_1 \cdot_M (m_2 \cdot mp(p) +_M m_2 \cdot_M resto(p)) =$	(def. de $\cdot_M$)
$m_1 \cdot (m_2 \cdot mp(p)) +_M m_1 \cdot_M (m_2 \cdot_M resto(p)) =$	(lema 3.82)
$m_1 \cdot (m_2 \cdot mp(p)) +_M (m_1 \cdot m_2) \cdot_M resto(p) =$	(hip. ind. y teo 3.57)
$(m_1 \cdot m_2) \cdot mp(p) +_M (m_1 \cdot m_2) \cdot_M resto(p) =$	(teo. 3.44 y teo 3.57)
$(m_1 \cdot m_2) \cdot_M p$	(def. de $\cdot_M$)

□

Lema 3.86. *Sean $m \in M_C[X]$ y, p, $q \in C[X]$.*

$$m \cdot_M (p \cdot q) = (m \cdot_M p) \cdot q$$

En ACL2,

```
(defthm |m *M (p * q) = (m *M p) * q|
  (implies (monomiop m)
           (= (*-monomio m (* p q))
              (* (*-monomio m p) q))))
```

Demostración. Por inducción según el siguiente esquema. Sea $P(m,p,q) \equiv m \cdot_M (p \cdot q) = (m \cdot_M p) \cdot q$:

1. $p = 0 \to P(m,p,q)$
2. $(p \neq 0 \wedge P(m, resto(p), q)) \to P(m,p,q)$

Caso base: $p = 0$

Inmediata por la definición de $\cdot_M$ y el teorema 3.81.

Paso de inducción:

Por hipótesis de inducción:

$$m \cdot_M (resto(p) \cdot q) = (m \cdot_M resto(p)) \cdot q$$

entonces

$$\begin{array}{ll}
m \cdot_M (p \cdot q) = & \\
m \cdot_M (mp(p) \cdot_M q + resto(p) \cdot q) = & \text{(def. de } \cdot) \\
m \cdot_M (mp(p) \cdot_M q) + m \cdot_M (resto(p) \cdot q) = & \text{(teo. 3.83)} \\
m \cdot_M (mp(p) \cdot_M q) + (m \cdot_M resto(p)) \cdot q = & \text{(h. ind. y teo 3.69)} \\
(m \cdot mp(p)) \cdot_M q + (m \cdot_M resto(p)) \cdot q = & \text{(lema 3.85 y 3.69)} \\
(m \cdot mp(p) +_M m \cdot_M resto(p)) \cdot q & \text{(def. de } +_M \text{ y } \cdot) \\
(m \cdot_M p) \cdot q & \text{(def. de } \cdot_M)
\end{array}$$

□

Teorema 3.87. *Sean p, q, $r \in C[X]$. Entonces, $(p \cdot q) \cdot r = p \cdot (q \cdot r)$. En* ACL2*,*

```
(defthm |(p * q) * r = p * (q * r)|
  (= (* (* p q) r) (* p (* q r))))
```

Demostración. Por inducción según el siguiente esquema. Sea $P(p,q,r) \equiv (p \cdot q) \cdot r = p \cdot (q \cdot r)$.

1. $p = 0 \to P(p,q,r)$
2. $(p \neq 0 \wedge P(resto(p), q, r)) \to P(p,q,r)$

Caso base: $p = 0$

Inmediata por la definición de $\cdot$, porque $=$ es una relación de equivalencia y por el teorema 3.81.

Paso de inducción:

Por hipótesis de inducción:

$$(resto(p) \cdot q) \cdot r = resto(p) \cdot (q \cdot r)$$

entonces

$$\begin{array}{ll}
(p \cdot q) \cdot r = & \\
(mp(p) \cdot_M q + resto(p) \cdot q) \cdot r = & \text{(def. de } \cdot) \\
(mp(p) \cdot_M q) \cdot r + (resto(p) \cdot q) \cdot r = & \text{(teo. 3.84)} \\
(mp(p) \cdot_M q) \cdot r + resto(p) \cdot (q \cdot r) = & \text{(h. ind. y teo. 3.69)} \\
mp(p) \cdot_M (q \cdot r) + resto(p) \cdot (q \cdot r) = & \text{(lema 3.86 y teo. 3.69)} \\
p \cdot (q \cdot r) & \text{(def. de } \cdot)
\end{array}$$

□

Por último, la conmutatividad del monoide es más complicada de demostrar. Su demostración es un ejemplo de la utilidad de las congruencias definidas entre la igualdad y la operación de suma. Requiere además definir un esquema de inducción apropiado.

Teorema 3.88. *Sean p, $q \in C[X]$. Entonces, $p \cdot q = q \cdot p$. En* ACL2,

```
(defthm |p * q = q * p|
  (= (* p q) (* q p)))
```

Demostración. Por inducción según el siguiente esquema. Sea $P(p, q) \equiv p \cdot q = q \cdot p$.

1. $(p = 0 \vee q = 0) \rightarrow P(p, q)$
2. $(p \neq 0 \wedge q \neq 0 \wedge P(resto(p), q) \wedge P(resto(p), resto(q)) \wedge P(p, resto(q))) \rightarrow P(p, q)$

Caso base: $p = 0 \vee q = 0$

Inmediata por la definición de $\cdot$ y por el teorema 3.81.

Paso de inducción:

Por hipótesis de inducción:

$$\begin{aligned}
&resto(p)\cdot q = q\cdot resto(p) && (1)\\
&resto(p)\cdot resto(q) = resto(q)\cdot resto(p) && (2)\\
&p\cdot resto(q) = resto(q)\cdot p && (3)
\end{aligned}$$

$$\begin{aligned}
&p\cdot q = \\
&mp(p)\cdot_M q + resto(p)\cdot q = && \text{(def. de } \cdot)\\
&(mp(p)\cdot mp(q) +_M mp(p)\cdot_M resto(q)) + resto(p)\cdot q = && \text{(def. de } \cdot_M)\\
&(mp(p)\cdot mp(q) +_M mp(p)\cdot_M resto(q)) + q\cdot resto(p) = && \text{(h.i.(1) y teo 3.69)}\\
&\\
&(mp(p)\cdot mp(q) +_M mp(p)\cdot_M resto(q)) + \\
&(mp(q)\cdot_M resto(p) + resto(q)\cdot resto(p)) = && \text{(def. de } \cdot)\\
&\\
&mp(q)\cdot mp(p) +_M (mp(p)\cdot_M resto(q) + \\
&(mp(q)\cdot_M resto(p) + resto(q)\cdot resto(p)) = && \text{(lemas 3.43 y 3.62,}\\
&\\
&mp(q)\cdot mp(p) +_M (mp(q)\cdot_M resto(p) + && \text{ y teo 3.57)}\\
&(mp(p)\cdot_M resto(q) + resto(q)\cdot resto(p)) = && \text{(teo. 3.64, 3.66}\\
&\\
&mp(q)\cdot mp(p) +_M (mp(q)\cdot_M resto(p) + && \text{ y 3.57)}\\
&(mp(p)\cdot_M resto(q) + resto(p)\cdot resto(q)) = && \text{(h.i.(2) y teo 3.69)}\\
&\\
&mp(q)\cdot mp(p) +_M (mp(q)\cdot_M resto(p) + p\cdot resto(q)) = && \text{(def. de } \cdot)\\
&mp(q)\cdot mp(p) +_M (mp(q)\cdot_M resto(p) + resto(q)\cdot p) = && \text{(h.i.(3) y teo 3.69)}\\
&(mp(q)\cdot mp(p) +_M mp(q)\cdot_M resto(p)) + resto(q)\cdot p = && \text{(lema 3.62)}\\
&mp(q)\cdot_M p + resto(q)\cdot p = && \text{(def. de } \cdot_M)\\
&q\cdot p && \text{(def. de } \cdot)
\end{aligned}$$

□

Teorema 3.89. *Sea p, q, $r \in C[X]$.*

$$p\cdot(q+r) = p\cdot q + p\cdot r$$

En ACL2,

```
(defthm |p * (q + r) = (p * q) + (p * r)|
  (= (* p (+ q r)) (+ (* p q) (* p r))))
```

Demostración. Inmediata por los teoremas 3.69, 3.84 y 3.88. □

Congruencias del producto

En esta sección demostramos que la igualdad es congruente respecto de la operación de multiplicación de un monomio y un polinomio (teorema 3.92, y respecto a la multiplicación de polinomios (teorema 3.93). La formalización se encuentra en `congruencias-producto.lisp` dentro del paquete `POL`.

Lema 3.90. *Sean* m_1, $m_2 \in M_C[X]$ *y* $p \in C[X]$.

$$m_1 \cdot_M (m_2 +^<_M p) = (m_1 \cdot m_2) +^<_M (m_1 \cdot_M p)$$

En ACL2,

```
(defthm |m *M (n +Mo p) = (m * n) +Mo (m *M p)|
  (implies (and (MON::monomiop m) (MON::monomiop n) (polinomiop p))
           (= (*-monomio m (+-monomio n p))
              (+-monomio (MON::* m n) (*-monomio m p)))))
```

Demostración. Por inducción según el siguiente esquema. Sea $P(m_1, m_2, p) \equiv m_1 \cdot_M (m_2 +^<_M p) = (m_1 \cdot m_2) +^<_M (m_1 \cdot_M p)$.

1. $(p = 0 \vee tp(p) \leq tp(m_2)) \rightarrow P(m_1, m_2, p)$
2. $(p \neq 0 \wedge tp(m_2) < tp(p) \wedge P(m_1, m_2, resto(p))) \rightarrow P(m_1, m_2, p)$

Caso base: $p = 0 \vee tp(p) \leq tp(m_2)$

Se tiene distinguiendo casos, utilizando las definiciones de $+^<_M$, $\cdot_M$ y $=$ y propiedades del producto y suma de monomios, y del orden de términos.

Paso de inducción:

Por hipótesis de inducción:

$$m_1 \cdot_M (m_2 +^<_M resto(p)) = (m_1 \cdot m_2) +^<_M (m_1 \cdot_M resto(p))$$

entonces se tiene el resultado por los lemas 3.55 y 3.82.

□

Lema 3.91. *Sean $m \in M_C[X]$ y $p \in C[X]$. Entonces, $m \cdot_M p = m \cdot_M fn(p)$. En* ACL2*,*

```
(defthm |m *M p = m *M fn(p)|
  (implies (syntaxp (not (and (consp p) (eq (primero p) 'fn))))
           (= (*-monomio m p) (*-monomio m (fn p))))))
```

Demostración. Por inducción según el siguiente esquema. Sea $P(m,p) \equiv m \cdot_M p = m \cdot_M fn(p)$:

1. $p = 0 \rightarrow P(m,p)$
2. $(p \neq 0 \wedge P(m, resto(p))) \rightarrow P(m,p)$

Caso base: $p = 0$

Inmediata por la definición de $\cdot_M$ y fn.

Paso de inducción:

Por hipótesis de inducción:

$$m \cdot_M resto(p) = m \cdot_M fn(resto(p))$$

entonces se tiene el resultado por las definiciones de $\cdot_M$, $=$ y fn, el lema 3.90 y el teorema 3.57.

□

Teorema 3.92. *La igualdad es congruente respecto de la multiplicación de un monomio y un polinomio.*

Sean m, $m' \in C[X]$ y p, $p' \in C[X]$.

- $m = m' \implies m \cdot_M p = m' \cdot_M p$

- $p = p' \implies m \cdot_M p = m \cdot_M p'$

En ACL2,

```
(defcong MON::= = (*-monomio m p) 1)
(defcong = = (*-monomio m p) 2)
```

Demostración.

- $m = m' \implies m \cdot_M p = m' \cdot_M p$

  Inmediata por las definiciones de $\cdot_M$ y de la igualdad de monomios.

- $p = p' \implies m \cdot_M p = m \cdot_M p'$

  Inmediata por el lema 3.91.

□

Por último, se extienden las congruencias a la operación de producto de polinomios, puesto que esta última se define a partir de la operación de producto un monomio por un polinomio.

Teorema 3.93. *La igualdad es congruente respecto de la multiplicación de polinomios.*

Sean p, p', q, $q' \in C[X]$.

- $q = q' \implies p \cdot q = p \cdot q'$
- $p = p' \implies p \cdot q = p' \cdot q$

En ACL2,

```
(defcong = = (* p q) 2)
(defcong = = (* p q) 1)
```

Demostración.

- $q = q' \implies p \cdot q = p \cdot q'$

  Por inducción según el siguiente esquema. Sea $P(p, q, q') \equiv p \cdot q = p \cdot q'$:

1. $p = 0 \rightarrow P(p, q, q')$
2. $(p \neq 0 \wedge P(resto(p), q, q')) \rightarrow P(p, q, q')$

Caso base : $p = 0$

Inmediata por la definición de $\cdot$.

Paso de inducción:

Por hipótesis de inducción:

$$resto(p) \cdot q = resto(p) \cdot q'$$

$$\begin{array}{ll} p \cdot q = & \\ mp(p) \cdot_M q + resto(p) \cdot q = & \text{(def. de } \cdot) \\ mp(p) \cdot_M q + resto(p) \cdot q' = & \text{(hip. ind. y teo. 3.69)} \\ mp(p) \cdot_M q' + resto(p) \cdot q' = & \text{(teo. 3.92)} \\ p \cdot q' & \text{(def. de } \cdot) \end{array}$$

- $p = p' \implies p \cdot q = p' \cdot q$

  Inmediata utilizando la congruencia del segundo parámetro y conmutatividad.

□

3.3.5. Polinomios normalizados

Ahora, a partir del predicado `ordenadop` que comprueba si un objeto es una lista de monomios no nulos en un orden estrictamente decreciente, la función de normalización `fn` y las operaciones del anillo de polinomios, se definen el reconocedor y las operaciones normalizadas correspondientes, además del polinomio nulo y el identidad. Nótese como se cambia de paquete (del paquete `POL` al `NPOL`) para poder seguir utilizando los mismos nombres en las operaciones. La formalización se encuentra en el fichero `polinomio.lisp` del directorio `polinomios-normalizados` dentro del paquete `NPOL`.

```
(in-package "NPOL")

(defmacro polinomiop (p)
  `(POL::ordenadop ,p))
```

```
(defun + (p q)
  (POL::fn (POL::+ p q)))

(defun - (p)
  (POL::fn (POL::- p)))

(defun * (p q)
  (POL::fn (POL::* p q)))

(defun nulo ()
  (POL::nulo))

(defun identidad ()
  (POL::identidad))
```

Nótese que las funciones `POL::nulo` y `POL::identidad` de los polinomios desnormalizados devuelven polinomios que ya están en forma normal. El núcleo de propiedades de anillos para estas operaciones normalizadas son las siguientes:

```
(defthm polinomiop-nulo
  (polinomiop (nulo))

(defthm polinomiop-identidad
  (polinomiop (identidad)))

(defthm polinomiop-+
  (polinomiop (+ p q)))

(defthm polinomiop--
  (polinomiop (- p)))

(defthm polinomiop-*
  (polinomiop (* p q)))

(defthm |p + q = q + p|
  (equal (+ p q) (+ q p)))

(defthm |(p + q) + r = p + (q + r)|
  (equal (+ (+ p q) r) (+ p (+ q r))))
```

```
(defthm |p * q = q * p|
  (equal (* p q) (* q p)))

(defthm |(p * q) * r = p * (q * r)|
  (equal (* (* p q) r) (* p (* q r))))

(defthm |p * (q + r) = (p * q) + (p * r)|
  (equal (* p (+ q r)) (+ (* p q) (* p r))))

(defthm |p + (- p) = 0|
  (equal (+ p (- p)) (nulo)))

(defthm |0 + p = p|
  (implies (polinomiop p)
           (equal (+ (nulo) p) p)))

(defthm |1 * p = p|
  (implies (polinomiop p)
           (equal (* (identidad) p) p)))
```

La mayoría de estos teoremas sobre las operaciones normalizadas se demuestran deshabilitando las operaciones desnormalizadas correspondientes y usando la contrapartida del teorema correspondiente a la representación desnormalizada.

A partir de ahora, los polinomios están normalizados y por *polinomiop* entendemos que están en forma normal.

3.4. Resumen

En este capítulo:

- Hemos presentado una formalización de los anillos de polinomios de múltiples variables en ACL2.
- Hemos abstraído el problema encapsulando un anillo de coeficientes que ha servido de base al desarrollo de los polinomios y a la verificación de sus propiedades fundamentales.
- Hemos estudiado los problemas de representación.

- Hemos construido y verificado una función de normalización que permite definir una igualdad semántica sobre los polinomios.
- Hemos demostrado la congruencia de dicha igualdad con las operaciones de anillo.

En un capítulo posterior emplearemos esta formalización abstracta para obtener un anillo de polinomios concreto, con coeficientes en el cuerpo de los racionales, verificado y ejecutable.

En realidad, lo expuesto proviene de una formalización anterior en la que empleábamos NQTHM, el demostrador de teoremas de Boyer-Moore. Es interesante hacer notar algunas de las desventajas que presentaba este demostrador respecto a ACL2 y que influyeron notablemente en nuestra decisión de traspasar el problema a ACL2.

La principal desventaja que se presentaba a la hora de formalizar los polinomios radicaba en la elección del cuerpo de coeficientes. En NQTHM sólo están los números naturales, por lo que había que formalizar e implementar un cuerpo de coeficientes partiendo de cero. En contra de lo que se pueda pensar, esto no es tarea fácil, sobre todo debido a un aumento en los problemas para decidir cuestiones aparentemente sencillas mediante aritmética lineal.

En cambio, ACL2 ya incorpora una formalización apropiada de $\mathbb{Q}$ y prácticamente todos estos problemas desaparecieron durante el tránsito a ACL2. Los problemas derivados de la ausencia de *shells* (un mecanismo primitivo para definir tipos abstractos de datos) se resolvieron con una formalización más cuidadosa.

Por otro lado, la ausencia de congruencias en NQTHM suponía una continua necesidad de demostración de lemas triviales, a simple vista con poca o ninguna relación con los teoremas que realmente se querían demostrar. En el desarrollo del trabajo en ACL2, las congruencias han jugado su papel, bien reduciendo la longitud y el tiempo de varias demostraciones, bien eliminando consejos innecesarios que disminuían su grado de automatización. Un ejemplo patente de esto lo constituye la demostración de la conmutatividad del producto de polinomios.

Capítulo 4

Órdenes polinómicos

4.1. Introducción

Este capítulo explica el desarrollo de un orden polinómico y la verificación de sus propiedades en ACL2. El resultado principal que se obtiene es la buena fundamentación del orden de polinomios definido, que se lleva a cabo mediante una inmersión ordinal apropiada.

La motivación para esto es la de servir de base para la demostración de la terminación de la reducción polinómica descrita en el capítulo 7 en ACL2.

La noción abstracta de reducción se puede modelar utilizando una función unaria *red*, que intenta simplificar su parámetro con respecto a un orden parcial estricto dado, $<$. Si el elemento no se puede simplificar se dice que es *irreducible* o que está *en forma normal.* y *red* lo devuelve sin modificar. En otro caso, el elemento es *reducible*. La propiedad característica de tales tipos de funciones es:

$$red(p) \neq p \implies red(p) < p$$

En este contexto, *red* se dice que es una *función de reducción*. La clausura de la función de reducción *red* se define recursivamente de la siguiente forma:

$$red^*(p) = \begin{cases} p, & \text{si } red(p) = p \\ red^*(red(p)), & \text{en otro caso (e.o.c.)} \end{cases}$$

Desafortunadamente, no está garantizado que esta función termine. Sin embargo, se puede garantizar su terminación siempre que $<$ esté bien fundamentado, que es justamente lo que capacita a ACL2 para admitir red^* bajo su principio de definición.

Por tanto, estamos fundamentalmente interesados en funciones de reducción relativas a órdenes bien fundados. Fijar un orden sobre los términos es sólo el primer paso para obtener un orden sobre polinomios.

A continuación expondremos la forma de realizar esto. En particular, se parte de un orden definido sobre términos que se extiende a monomios. Posteriormente, éste a su vez se extiende para obtener el orden polinómico inducido.

4.2. Orden de términos

A continuación se muestra cómo definir un orden total estricto entre los términos. Además, demostraremos que este orden está bien fundamentado. La formalización se encuentra en `termino.lisp` dentro del directorio `polinomios-racionales`.

Para ordenar los términos, una vez determinado el conjunto de variables X, únicamente es necesario tener en cuenta las listas de exponentes. La elección obvia consiste en fijar un *orden lexicográfico* entre dichas secuencias de números naturales.

4.2.1. El orden lexicográfico

En el caso de términos definidos sobre el mismo conjunto de variables, la definición del orden lexicográfico es directa, puesto que las secuencias de números naturales implicadas tienen la misma longitud.

Definición 4.1. Sean $[a_1, \ldots, a_k]$ y $[b_1, \ldots, b_k] \in T[X]$, se define el orden, $<_T$, sobre términos de la siguiente forma:

$$[a_1, \ldots, a_k] <_T [b_1, \ldots, b_k] \equiv \exists i \ (a_i < b_i \wedge \forall j < i \ a_j = b_j)$$

La siguiente función booleana ACL2 define la relación de orden lexicográfico estricto sobre los términos de esta forma, pero, análogamente a lo que ocurría con otras operaciones sobre términos, será algo más general. Así, si dos términos no son compatibles, es decir, no están definidos sobre el mismo

conjunto de variables, el de menor número de variables será considerado el menor si, y sólo si, es prefijo del otro.

```
(defun < (a b)
  (cond ((or (atom a) (atom b))
         (not (atom b)))
        ((equal (first a) (first b))
         (< (rest a) (rest b)))
        (t
         (LISP::< (first a) (first b)))))
```

No es difícil demostrar, que esta definición cumple las propiedades fundamentales inherentes a este tipo de relaciones (irreflexividad, antisimetría y transitividad).

Teorema 4.2 (irreflexividad). *Sea $a \in T[X]$. Entonces, $\neg(a <_T a)$. En* ACL2,

```
(defthm |~(a < a)|
  (not (< a a)))
```

Demostración. Por inducción sobre la estructura de la función $<_T$. □

Teorema 4.3. *Sean $a, b \in T[X]$. Si $a <_T b$, entonces $\neg(b <_T a)$. En* ACL2,

```
(defthm |a < b => ~(b < a)|
  (implies (< a b) (not (< b a))))
```

Demostración. Por inducción sobre la estructura de la función $<_T$. □

Teorema 4.4 (transitividad). *Sean $a, b, c \in T[X]$.*

$$a <_T b \wedge b <_T c \implies a <_T c$$

En ACL2,

```
(defthm |a < b & b < c => a < c|
    (implies (and (< a b) (< b c)) (< a c)))
```

Demostración. Por inducción sobre la estructura de la función $<_T$ y por el teorema 4.3. □

También es posible demostrar la tricotomía.

Teorema 4.5. *Sean* $a,\ b \in T[X]$.

$$a <_T b \vee b <_T a \vee a = b$$

En ACL2,

```
(defthm |a < b or b < a or a = b|
  (implies (and (terminop a) (terminop b))
           (or (< a b) (< b a) (= a b))))
```

Demostración. Por inducción sobre la estructura de la función $<_T$. □

4.2.2. Inmersión de los términos en los ϵ_0-ordinales

Para sumergir los términos en los ϵ_0-ordinales adoptaremos el siguiente criterio.

Definición 4.6. Sea $a = [a_1, \ldots, a_k] \in T[X]$. El ϵ_0-ordinal asociado con a, $T_{\epsilon_0}(a)$ será el siguiente.

$$T_{\epsilon_0}(a) = \begin{cases} 0, & \text{si } k = 0 \\ \omega^{\omega^k + a_1} + \cdots + \omega^{\omega + a_k}, & \text{e.o.c.} \end{cases}$$

Esta inmersión presenta la ventaja de proporcionar una traducción directa a partir de la lista de exponentes del término, como se puede observar en los ejemplos que se muestran a continuación. Por otro lado, el tipo ordinal que se obtiene no es muy alto, lo que facilita la manipulación en esta representación.

$$\underbrace{x}_{\texttt{(1)}} \longmapsto \underbrace{\omega^{\omega+1}}_{\texttt{((1 . 1) . 0)}}$$

$$\underbrace{x^8 \cdot y^0}_{\texttt{(8 0)}} \longmapsto \underbrace{\omega^{\omega^2+8} + \omega^{\omega}}_{\texttt{((2 . 8) (1 . 0) . 0)}}$$

$$\underbrace{x^4 \cdot y^3 \cdot z^5}_{\texttt{(4 3 5)}} \longmapsto \underbrace{\omega^{\omega^3+4} + \omega^{\omega^2+3} + \omega^{\omega+5}}_{\texttt{((3 . 4) (2 . 3) (1 . 5) . 0)}}$$

En ACL2 se procede a la inmersión de los términos en los ϵ_0-ordinales mediante la siguiente función recursiva. La función `len` devuelve la longitud del término que recibe como parámetro, o lo que es lo mismo, el número de variables del término.

```
(defun termino->e0-ordinal (a)
  (if (atom a)
      0
    (cons (cons (len a) (first a))
          (termino->e0-ordinal (rest a)))))
```

Como veremos a continuación, se demuestra que verdaderamente la función `termino->e0-ordinal` produce un ϵ_0-ordinal a partir de un término.

4.2.3. Buena fundamentación

Teorema 4.7. *El orden sobre términos, $<_T$, está bien fundamentado.*

Sean $a, b \in T[X]$, $a <_T b$ y $<_{\epsilon_0}$ el orden sobre los ϵ_0-ordinales entonces $T_{\epsilon_0}(a)$ es un ordinal y $T_{\epsilon_0}(a) <_{\epsilon_0} T_{\epsilon_0}(b)$. En ACL2*,*

```
(defthm buena-fundamentacion-<
  (and (implies (terminop a)
                (e0-ordinalp (termino->e0-ordinal a)))
       (implies (and (terminop a) (terminop b)
                     (< a b))
                (e0-ord-< (termino->e0-ordinal a)
                          (termino->e0-ordinal b))))
  :rule-classes (:rewrite :well-founded-relation))
```

Demostración. Por inducción según el siguiente esquema. Sea $P(a, b) \equiv$ $T_{\epsilon_0}(a) <_{\epsilon_0} T_{\epsilon_0}(b)$:

1. $b = [] \rightarrow P(a, b)$
2. $(b \neq [] \wedge P(resto(a), resto(b))) \rightarrow P(a, b)$

Caso base: $b = []$

Inmediata por la definición de $<_T$, $<_{\epsilon_0}$ y y T_{ϵ_0}.

Paso de inducción:

Por hipótesis de inducción:

$$T_{\epsilon_0}(resto(a)) <_{\epsilon_0} T_{\epsilon_0}(resto(b))$$

entonces

$$\begin{array}{ll}
T_{\epsilon_0}(a) = & \\
\omega^{\omega^{len(a)}+primero(a)} + T_{\epsilon_0}(resto(a)) <_{\epsilon_0} & (\text{def. } T_{\epsilon_0}) \\
\omega^{\omega^{len(a)}+primero(a)} + T_{\epsilon_0}(resto(b)) <_{\epsilon_0} & (\text{hip. ind. y def. } <_{\epsilon_0}) \\
\omega^{\omega^{len(b)}+primero(b)} + T_{\epsilon_0}(resto(b)) = & (a < b) \\
T_{\epsilon_0}(b) & (\text{def. } T_{\epsilon_0})
\end{array}$$

□

Para establecer que una relación está bien fundamentada en ACL2 es necesario primero disponer de una función que realice la inmersión de los objetos de la relación en los ϵ_0-ordinales. Sin embargo, es muy importante demostrar la corrección de la función inmersora, lo que no siempre es sencillo cuando su tipo ordinal es elevado. En este caso no es difícil, previa demostración de un lema:

```
(local
  (defthm extension-correccion
    (implies (and (terminop a)
                  (e0-ordinalp (termino->e0-ordinal (rest a))))
             (e0-ordinalp (termino->e0-ordinal a)))))

(local
  (defthm e0-ordinalp-termino->e0-ordinal
    (implies (terminop a)
             (e0-ordinalp (termino->e0-ordinal a)))))
```

Una vez demostrada la corrección de la función inmersora, basta comprobar que preserva el orden, es decir, que los ϵ_0-ordinales correspondientes a cada par de elementos relacionados siguen estando relacionados.

Este proceso nos permite en ACL2 añadir el teorema de buena fundamentación a la clase de reglas `:well-founded-relation`, con lo que la relación de

orden definida (que es *noetheriana*) puede ser utilizada, cuando sea necesario, para demostrar la terminación de funciones definidas sobre términos.

Desgraciadamente, con la función < presentada, este teorema no puede ser demostrado, ya que es falso. En efecto, basta pensar en términos con distinto número de variables para comprender el problema; claramente hay dos casos simétricos, según el primero tenga menos variables que el segundo o viceversa:

$$\begin{array}{ccc} x^4y^2z & <_T & x^6y^4 \\ \downarrow & & \downarrow \\ \omega^{\omega^3+4}+\omega^{\omega^2+2}+\omega^{\omega+1} & \not<_{\epsilon_0} & \omega^{\omega^2+6}+\omega^{\omega+4} \end{array} \qquad \begin{array}{ccc} x^8 & \not<_T & x^3y^2 \\ \downarrow & & \downarrow \\ \omega^{\omega+8} & <_{\epsilon_0} & \omega^{\omega^2+3}+\omega^{\omega+2} \end{array}$$

Cuando los términos son compatibles, el problema desaparece. Podría pensarse que completando de algún modo adecuado el término con menor número de variables puede evitarse el problema. Sin embargo, la solución no es tan simple, ya que al sumergir un término nada se sabe acerca de con cuáles puede ser comparado. Una solución factible consiste en tratar de manera especial ambos casos:

```
(defun < (a b)
  (cond ((or (atom a) (atom b))
         (not (atom b)))
        ((LISP::< (len a) (len b))
         t)
        ((LISP::> (len a) (len b))
         nil)
        ((equal (first a) (first b))
         (< (rest a) (rest b)))
        (t
         (LISP::< (first a) (first b)))))
```

Con esta nueva formalización sigue siendo cierto el teorema de buena fundamentación.

4.3. Orden de monomios

En la sección 4.2 se ha mostrado que el orden de términos definido está bien fundamentado. El *orden de monomios* es justo la traducción del orden de

términos a monomios donde los monomios se comparan de acuerdo con sus términos. Por tanto, el orden de monomios hereda todas la propiedades del orden de términos. Su formalización se encuentra en `monomios.lisp`.

Definición 4.8. Sean $a = c_a \cdot X^{[a_1,\ldots,a_k]}$, $b = c_b \cdot X^{[b_1,\ldots,b_k]} \in M_C[X]$, se define el orden, $<_M$, sobre monomios de la siguiente forma:

$$a <_M b \equiv [a_1, \ldots, a_k] <_T [b_1, \ldots, b_k]$$

En ACL2,

```
(defmacro < (a b)
  `(TER::< (termino ,a) (termino ,b)))
```

Definición 4.9. Sea $a \in M_C[X]$. El ϵ_0-ordinal asociado con a, $M_{\epsilon_0}(a)$ será el siguiente.

$$M_{\epsilon_0}(a) = T_{\epsilon_0}(tp(a))$$

En ACL2,

```
(defmacro monomio->e0-ordinal (a)
  `(TER::termino->e0-ordinal (termino ,a)))
```

Teorema 4.10. *El orden sobre monomios, $<_M$, está bien fundamentado.*

Sean a, $b \in M_C[X]$, $a <_M b$ y $<_{\epsilon_0}$ el orden sobre los ϵ_0-ordinales entonces

$$M_{\epsilon_0}(a) <_{\epsilon_0} M_{\epsilon_0}(b)$$

En ACL2,

```
(defthm buena-fundamentacion-<-M
  (and (implies (monomiop a)
                (e0-ordinalp (monomio->e0-ordinal a)))
       (implies (and (monomiop a) (monomiop b)
                     (< a b))
                (e0-ord-< (monomio->e0-ordinal a)
                          (monomio->e0-ordinal b)))))
```

Demostración. Inmediata por el teorema 4.7. □

4.4. Orden polinómico inducido

Como ya se ha expuesto en el capítulo anterior, los polinomios se construyen a partir de monomios. Por tanto, el orden de monomios se puede extender a polinomios normalizados de una forma directa. Su formalización se encuentra en `orden.lisp`.

Definición 4.11. Sean p, q dos polinomios normalizados. Se dice que, $p < q$, si se cumple alguna de las tres siguientes condiciones:

- $p = 0 \wedge q \neq 0$
- $p \neq 0 \wedge q \neq 0 \wedge tp(p) = tp(q) \wedge resto(p) < resto(q)$
- $p \neq 0 \wedge q \neq 0 \wedge mp(p) <_M mp(q)$

El siguiente predicado ACL2 implementa el orden anterior, comparando dos polinomios como listas de monomios utilizando `MON::=T` (la igualdad entre los términos subyacentes a los dos monomios) y `MON::<` (el orden de monomios).

```
(defun < (p q)
  (cond ((or (nulop p) (nulop q))
         (not (nulop q)))
        ((MON::=T (primero p) (primero q))
         (< (resto p) (resto q)))
        (t
         (MON::< (primero p) (primero q)))))
```

Se demuestra que esta relación satisface las propiedades de un orden parcial (irreflexividad y transitividad).

Teorema 4.12 (irreflexividad). *Sea $p \in C[X]$. Entonces, $\neg(p < p)$. En* ACL2,

```
(defthm |~(p < p)|
  (not (< p p)))
```

Demostración. Por inducción sobre la estructura de la función $<$. □

Teorema 4.13. *Sean p, $q \in T[X]$. Si $p < q$, entonces, $\neg(q < p)$. En* ACL2,

```
(defthm |p < q => ~(q < p)|
  (implies (< p q)
           (not (< q p))))
```

Demostración. Por inducción sobre la estructura de la función $<$ y por el teorema 4.3. □

Teorema 4.14 (transitividad). *Sean p, q, $r \in T[X]$.*

$$p < q \wedge q < r \implies p < r$$

En ACL2,

```
(defthm |p < q & q < r => p < r|
  (implies (and (< p q) (< q r))
           (< p r)))
```

Demostración. Por inducción sobre la estructura de la función $<$ y por los teoremas 4.4 y 4.13. □

4.4.1. Inmersión de los polinomios en los ϵ_0-ordinales

Si $p = m_1 + \cdots + m_n$ está normalizado, entonces por definición de forma normal se tiene que $m_n <_M \cdots <_M m_1$, y por tanto que:

$$M_{\epsilon_0}(m_n) <_{\epsilon_0} \cdots <_{\epsilon_0} M_{\epsilon_0}(m_1),$$

Esto se sigue del teorema que establece que $<_M$ es una relación bien fundamentada. De esta forma, la inmersión de los polinomios en los ϵ_0-ordinales se lleva a cabo de la siguiente forma.

Definición 4.15. Sea $p = m_1 + \cdots + m_n$ un polinomio normalizado y $m_i = c_i \cdot X^{[a_{i1},\ldots,a_{ik}]}$, $1 \leq i \leq n$ sus monomios. El ϵ_0-ordinal asociado con p, $P_{\epsilon_0}(p)$, será el siguiente.

$$P_{\epsilon_0}(p) = \sum_{i=1}^{n} \omega^{M_{\epsilon_0}(m_i)} = \sum_{i=1}^{n} \omega^{\sum_{j=1}^{k} \omega^{\omega^{k-j+1}+a_{ij}}}.$$

Una vez que se tiene (se definió en la sección anterior) la función que realiza la inmersión de monomios a los ϵ_0-ordinales se define en ACL2 la correspondiente a polinomios mediante la función `polinomio->e0-ordinal`:

```
(defun polinomio->e0-ordinal (p)
  (cond ((nulop p)
         0)
        (t
         (cons (MON::monomio->e0-ordinal (primero p))
               (polinomio->e0-ordinal (resto p))))))
```

4.4.2. Buena fundamentación

Al intentar demostrar la buena fundamentación del orden de polinomios, nos damos cuenta de un problema de la definición que se ha tomado: si M_{ϵ_0} devuelve 0, la función P_{ϵ_0} no construye un ϵ_0-ordinal apropiado en forma normal de Cantor. Una posible solución es incrementar los ordinales asociados a cada término por 1.

Sea $a = [a_1, \ldots, a_k] \in T[X]$. El ϵ_0-ordinal asociado con a, $T_{\epsilon_0}(a)$ será el siguiente.

$$T_{\epsilon_0}(a) = \begin{cases} 1, & \text{si } n = 0 \\ \omega^{\omega^n + a_1} + \cdots + \omega^{\omega + a_n} + 1, & \text{e.o.c.} \end{cases}$$

En ACL2 la definición de términos quedaría de la siguiente forma.

```
(defun termino->e0-ordinal (a)
  (if (endp a)
      1                              ; inicialmente 0
    (cons (cons (len a) (first a))
          (termino->e0-ordinal (rest a)))))
```

Obviamente, este cambio no altera la buena fundamentación del orden de monomios. Con esta pequeña modificación, los ordinales se asignan a los polinomios como sigue.

Sea $p = m_1 + \cdots + m_n$ un polinomio normalizado y $m_i = c_i \cdot X^{[a_{i1},\ldots,a_{ik}]}$, $1 \leq i \leq n$ sus monomios. El ϵ_0-ordinal asociado con p, $P_{\epsilon_0}(p)$, será el siguiente.

$$[m_1, \ldots, m_n] \longmapsto \sum_{i=1}^{n} \omega^{\sum_{j=1}^{k} \omega^{\omega^{k-j+1} + a_{ij}} + 1}.$$

A continuación se puede ya proceder a demostrar la buena fundamentación de este orden sobre polinomios.

Teorema 4.16. *El orden sobre polinomios, $<$, está bien fundamentado.*

Sean p, $q \in C[X]$, $p < q$ y $<_{\epsilon_0}$ el orden sobre los ϵ_0-ordinales entonces

$$P_{\epsilon_0}(p) <_{\epsilon_0} P_{\epsilon_0}(q)$$

En ACL2,

```
(defthm buena-fundamentacion-<
  (and (implies (polinomiop p)
                (e0-ordinalp (polinomio->e0-ordinal p)))
       (implies (and (polinomiop p) (polinomiop q) (< p q))
                (e0-ord-< (polinomio->e0-ordinal p)
                          (polinomio->e0-ordinal q))))
                          :rule-classes :well-founded-relation)
```

Demostración. Por inducción según el siguiente esquema. Sea $S(p,q) \equiv P_{\epsilon_0}(p) <_{\epsilon_0} P_{\epsilon_0}(q)$:

1. $q = 0 \rightarrow S(p,q)$
2. $(q \neq 0 \wedge S(resto(p), resto(q))) \rightarrow S(p,q)$

Caso base: $q = 0$

Inmediata por la definición de $<$, $<_{\epsilon_0}$ y y P_{ϵ_0}.

Paso de inducción:

Por hipótesis de inducción:

$$P_{\epsilon_0}(resto(p)) <_{\epsilon_0} P_{\epsilon_0}(resto(q))$$

entonces

$P_{\epsilon_0}(p) =$	
$M_{\epsilon_0}(mp(p)) + P_{\epsilon_0}(resto(p)) <_{\epsilon_0}$	(def. P_{ϵ_0})
$M_{\epsilon_0}(mp(p)) + P_{\epsilon_0}(resto(q)) <_{\epsilon_0}$	(hip. ind. y def. $<_{\epsilon_0}$)
$M_{\epsilon_0}(mp(q)) + P_{\epsilon_0}(resto(q)) =$	($p < q$, teo. 4.10 y def. $<_{\epsilon_0}$)
$P_{\epsilon_0}(q)$	(def. P_{ϵ_0})

□

Una alternativa a la modificación de `termino->e0-ordinal` consiste en cambiar `polinomio->e0-ordinal` adecuadamente. Esta última posibilidad tiene la ventaja de separar dos conceptos: el desarrollo de órdenes de monomios y el desarrollo de órdenes polinómicos inducidos.

```
(defun polinomio->e0-ordinal (p)
  (if (endp p)
      0
    (cons (incrementa-ordinal (monomio->e0-ordinal (primero p)))
          (polinomio->e0-ordinal (resto p)))))

(defun incrementa-ordinal (a)
  (if (consp a)
      (if (and (atom (rest a)) (integerp (rest a)))
          (cons (first a) (+ (rest a) 1))
        (cons (first a) (incrementa-ordinal (rest a))))
    (if (and (integerp a) (<= 0 a))
        (+ a 1)
      a)))
```

Como se puede ver, el ordinal devuelto por la función `monomio->e0-ordinal` se incrementa por la función `incrementa-ordinal`. Se demuestra que el incremento con esta función de un ordinal es también un ordinal y que no produce 0.

```
(defthm |e0-ordinalp(a) => e0-ordinalp(incrementa-ordinal(a))|
  (implies (e0-ordinalp a)
           (e0-ordinalp (incrementa-ordinal a))))

(defthm |~(incrementa-ordinal(a) = 0)|
  (not (equal (incrementa-ordinal a) 0)))
```

La propiedad fundamental demostrada sobre esta función es que si un ordinal es menor que otro, entonces seguirá siendo menor cuando se incrementen.

```
(defthm |a <e0 b => a + 1 <e0 b + 1|
  (implies (and (e0-ordinalp a) (e0-ordinalp b)
                (e0-ord-< a b))
           (e0-ord-< (incrementa-ordinal a)
                     (incrementa-ordinal b))))
```

Una vez que se demuestran estos teoremas, estamos en condición de establecer la corrección de la inmersión ordinal de los polinomios y, por tanto, la buena fundamentación del orden de polinomios.

Por último, se demuestra que no hay ningún ordinal entre un ordinal y su incremento.

```
(defthm |~(a <e0 b & b <e0 a + 1)|
  (implies (and (e0-ordinalp a) (e0-ordinalp b))
           (not (and (e0-ord-< a b)
                     (e0-ord-< b (incrementa-ordinal a))))))
```

4.5. Resumen

En este capítulo:

- Hemos desarrollado en ACL2 un orden sobre polinomios.
- Hemos demostrado que este orden está bien fundamentado.
- Hemos obtenido, como subproducto de la demostración de buena fundamentación, una inmersión de los polinomios en los ϵ_0-ordinales.

La única forma de demostrar en ACL2 la buena fundamentación de una relación es mediante inmersión ordinal. La buena fundamentación de los ϵ_0-ordinales con su orden habitual es un resultado metateórico en ACL2.

El orden sobre los polinomios se ha desarrollado de manera modular elevando las propiedades de buena fundamentación desde los términos a los monomios y de ahí a los polinomios. Los coeficientes quedan excluidos del proceso: realmente, estamos formalizando la noción matemática de orden lexicográfico sobre polinomios. Hemos mostrado también cómo el desarrollo del orden sobre monomios y el de su orden polinómico inducido se pueden realizar independientemente.

La principal utilidad de este orden es la de constituir una base sólida para demostrar la terminación de las reducciones sobre polinomios que aparecen implicadas en el desarrollo del algoritmo de Buchberger. Más adelante, veremos que estas reducciones producen siempre polinomios más pequeños respecto del orden que acabamos de definir. Esto, unido a su buena fundamentación, nos permitirá demostrar la existencia de formas normales.

Capítulo 5

Polinomios racionales

5.1. Introducción

Como ya se apuntó en el capítulo 3, en nuestra experiencia, el conjunto de propiedades necesarias para verificar algoritmos de cierta entidad sobre polinomios (y el algoritmo de Buchberger es uno de ellos) supera con mucho a las propiedades básicas de anillo. Esto nos ha llevado a su extensión en diferentes aspectos.

Por un lado, es necesario instanciar el anillo de polinomios con coeficientes abstractos implementado en el capítulo 3 para obtener una implementación ejecutable. En concreto, obtendremos el anillo de polinomios sobre el cuerpo de los números racionales. ACL2 ya posee una formalización de $\mathbb{Q}$ por lo que emplear este cuerpo resulta una buena elección. Por ejemplo, el sistema contiene un procedimiento de decisión para la aritmética lineal, esto nos descarga en muchas ocasiones de la pesada tarea de demostrar un gran número de propiedades triviales sobre los números.

Estos polinomios racionales serán los que empleemos en la construcción del algoritmo de Buchberger. Nótese que no bastaría con un anillo de coeficientes, ya que el algoritmo original necesita polinomios definidos sobre un cuerpo de coeficientes.

Por otro lado, también va a ser necesario poder comprobar si un término es divisible por otro, poder realizar la división en tal caso, y calcular el máximo común divisor de dos términos. Será necesario extender los términos para contemplar dichas operaciones.

Conviene, del mismo modo, por motivos técnicos que explicaremos más adelante, que el conjunto soporte de las variables sea el mismo para todos los polinomios. Esto nos llevará a definir la noción de polinomio uniforme.

En este capítulo se presentan todas estas extensiones, sin las que no sería posible demostrar la corrección del algoritmo de Buchberger.

5.2. Anillos de polinomios racionales

El anillo de polinomios racionales se obtiene directamente y sin dificultad por instanciación funcional a partir del anillo de polinomios con coeficientes arbitrarios implementado en el capítulo 3. Su formalización se encuentra en el directorio `polinomios-racionales` y se compone de los siguientes ficheros: `racional.lisp` en el paquete `RAC`; `termino.lisp` en el paquete `RAC-TER`; `monomio.lisp` en `RAC-MON`; `polinomio.lisp`, `forma-normal.lisp`, `congruencias-suma.lisp`, `congruencias-producto.lisp`, `opuesto.lisp`, `suma.lisp` y `producto.lisp`, todos ellos en el paquete `RAC-POL`; por último, `orden.lisp` y `polinomio-normalizado.lisp` en el paquete `RAC-NPOL`.

Nótese que se cambia de paquete (del paquete `POL` al `RAC-POL`, del `NPOL` al `RAC-NPOL`, etc.) para poder seguir utilizando los mismos nombres en las operaciones de polinomios.

Para ello sólo es necesario proporcionar una definición apropiada para las operaciones de coeficientes y términos abstractas presentadas en los encapsulados de las secciones 3.3.1 y 3.3.2, y demostrar los axiomas correspondientes.

Dichas definiciones se construirán justamente con los testigos utilizados en la formalización. Dichos testigos utilizan precisamente las operaciones de los números racionales implementados en ACL2.

5.2.1. Cuerpo de racionales

Sobre el conjunto de coeficientes racionales son necesarias, además de las de anillos, las propiedades que lo estructuran como cuerpo.

Definición 5.1. Sea C un conjunto, $+$ y $\cdot$ operaciones binarias en C, $-$ una operación unaria en C, $0, 1 \in C$ y $^{-1}$ una operación unaria en $C \setminus \{0\}$. Decimos que $\langle C, +, -, \cdot, ^{-1}, 0, 1\rangle$ es un cuerpo si:

- $\langle C, +, -, \cdot, 0, 1\rangle$ es un anillo conmutativo con identidad.

- $\langle C \setminus \{0\}, \cdot, ^{-1}, 1\rangle$ es un grupo.

Por tanto, para estructurar a nuestros coeficientes como cuerpo, se definirá la función de división de dos racionales utilizando la que ya está implementada en el sistema.

```
(defun / (a b)
  (LISP::/ a b))
```

Finalmente lo único que tenemos que añadir para tener el cuerpo de racionales es la propiedad de cancelación del producto respecto de la división. Para todo $c \in C \setminus \{0\}$, $c \cdot c^{-1} = 1$. El teorema ACL2 correspondiente es el siguiente.

```
(defthm |a * (1 / a) = 1|
  (implies (and (racionalp a)
                (not (= a (nulo))))
           (= (* a (/ (identidad) a)) (identidad))))
```

Algunas de las propiedades adicionales que se demuestran sobre la función división con sus teoremas ACL2 correspondientes son las siguientes:

- $a \neq 0 \in \mathbb{Q}$, $a/a = 1$

```
(defthm |a / a = 1|
  (implies (and (racionalp a)
                (not (= a (nulo))))
           (= (/ a a) (identidad))))
```

- $a \in \mathbb{Q}$, $a/1 = a$

```
(defthm |a / 1 = a|
  (implies (racionalp a)
           (equal (/ a (identidad)) a)))
```

- $a \neq 0$, $b \in \mathbb{Q}$, $(b \cdot a)/a = b$

```
(defthm |(b * a) / a = b|
  (implies (and (racionalp a) (racionalp b)
                (not (= a (nulo))))
           (= (/ (* b a) a) b)))
```

- $a \neq 0$, $b \in \mathbb{Q}$, $(b/a) \cdot a = (b \cdot a)/a$

```
(defthm |(b / a) * a = (b * a) / a|
  (implies (and (racionalp a) (racionalp b)
                (not (= a (nulo))))
           (= (* (/ b a) a) (/ (* b a) a))))
```

- a, $b \in \mathbb{Q}$, $a \cdot (1/b) = a/b$

```
(defthm |a * (1 / b) = a / b|
  (implies (and (racionalp a) (racionalp b))
           (equal (* a (/ (identidad) b))
                  (/ a b))))
```

- a, b, $c \in \mathbb{Q}$, $a \cdot (b/c) = (a \cdot b)/c$

```
(defthm |a * (b / c) = (a * b) / c|
  (implies (and (racionalp a) (racionalp b) (racionalp c))
           (equal (* a (/ b c)) (/ (* a b) c))))
```

Las demostraciones de todas estas propiedades en ACL2 son inmediata a partir de las definiciones de las operaciones sobre números y sus propiedades que están incorporadas en el sistema.

5.2.2. Anillos de polinomios racionales

Finalmente, por instanciación funcional, se obtienen todas las propiedades del anillo conmutativo con identidad de polinomios racionales. Los teoremas ACL2 correspondientes son los siguientes:

- $\langle Q[X], +, -, 0\rangle$ es un grupo conmutativo:
  - Asociativa: `|(p + q) + r = p + (q + r)|` en `suma.lisp`.
  - Conmutativa: `|p + q = q + p|` en `suma.lisp`.
  - Neutro: `|0 + p = p|` en `suma.lisp`.
  - Opuesto: `|p + (- p) = 0|` en `opuesto.lisp`.
- $\langle Q[X], \cdot, 1\rangle$ es un monoide conmutativo:
  - Asociativa: `|(p * q) * r = p * (q * r)|` en `producto.lisp`.

  - Conmutativa: `|p * q = q * p|` en `producto.lisp`.
  - Neutro: `|1 * p = p|` en `producto.lisp`.

- $\cdot$ distribuye respecto de $+$: `|p * (q + r) = (p * q) + (p * r)|` en `producto.lisp`.

Además se demuestra que la relación de orden está bien fundamentada:

- Irreflexiva: `|~(p < p)|` en `orden.lisp`.
- Transitiva: `|p < q & q < r => p < r|` en `orden.lisp`.
- Bien fundamentada: `buena-fundamentacion-<` en `orden.lisp`.

Por último en el fichero `polinomio-normalizado` se introducen las correspondientes operaciones normalizadas de los polinomios racionales y se demuestran sus propiedades de anillo por instanciación funcional.

Estos polinomios racionales normalizados obtenidos son los que constituyen la base para el algoritmo de Buchberger. A partir de ahora cuando se hable de polinomios nos referiremos a $Q[X]$ representado por polinomios normalizados.

5.3. Divisibilidad

Al abordar el problema de la parada del algoritmo de Buchberger interviene, entre otros, el concepto de divisibilidad entre términos. A continuación procedemos a definir esta relación entre términos que está formalizada en `termino-division.lisp`.

Definición 5.2. Sean $a = X^{[a_1,\ldots,a_k]}$, $b = X^{[b_1,\ldots,b_k]} \in T[X]$ entonces se dice que b es divisible por a o que a divide a b, y se notará por $a \mid b$, si $a_i \leq b_i$ para todo $1 \leq i \leq k$.

En ACL2,

```
(defun dividep (a b)
  (cond ((and (not (terminop a)) (not (terminop b)))
         t)
        ((not (terminop a)) t)
```

```
((not (terminop b)) nil)
((endp a) t)
((endp b) (endp a))
(t (and (LISP::<= (first a) (first b))
        (dividep (rest a) (rest b))))))
```

Nótese que la función se define de manera que trate los casos anómalos de forma razonable.

La relación de divisibilidad tiene las propiedades reflexiva, antisimétrica y transitiva.

Teorema 5.3 (reflexividad). *Sea $a \in T[X]$. Entonces, a divide a a. En* ACL2*,*

```
(defthm |dividep(a, a)|
  (dividep a a))
```

Teorema 5.4 (antisimetría). *Sean a, $b \in T[X]$. Si a divide a b y b divide a a, entonces $a = b$. En* ACL2*,*

```
(defthm |dividep(a, b) & dividep(b, a) => a = b|
  (implies (and (terminop a) (terminop b))
           (implies (and (dividep a b) (dividep b a))
                    (= a b))))
```

Teorema 5.5 (transitividad). *Sean a, b, $c \in T[X]$. Si a divide a b y b divide a c, entonces a divide a c. En* ACL2*,*

```
(defthm |dividep(a, b) & dividep(b, c) => dividep(a, c)|
  (implies (and (dividep a b) (dividep b c))
           (dividep a c)))
```

Otra propiedad que también será empleada en varias ocasiones es que si un término divide a otro, entonces también divide al producto de ese término con otro:

```
(defthm |dividep(a, b) => dividep(a, c * b)|
  (implies (dividep a b)
           (dividep a (* c b))))
```

Una vez definido el concepto de divisibilidad, se define la función de división entre dos términos y la de mínimo común múltiplo.

Definición 5.6. Sean $a = X^{[a_1,\ldots,a_k]}$, $b = X^{[b_1,\ldots,b_k]} \in T[X]$ tal que a es divisible por b. Entonces la división entre términos, notada por $/$, se define de la siguiente manera:

$$a/b = X^{[a_1,\ldots,a_k]}/X^{[b_1,\ldots,b_k]} = X^{[a_1-b_1,\ldots,a_k-b_k]}$$

En ACL2,

```
(defun / (a b)
  (cond ((and (not (terminop a)) (not (terminop b)))
         (uno))
        ((not (terminop a)) (uno))
        ((not (terminop b)) a)
        ((endp a) b)
        ((endp b) a)
        (t
         (cons (LISP::- (first a) (first b))
               (/ (rest a) (rest b))))))
```

Algunas de las propiedades adicionales que se demuestran sobre la función división con sus teoremas ACL2 correspondientes son las siguientes:

- $a, b \in T[X]$, $a \mid b \implies a \cdot (b/a) = b$

  ```
  (defthm |a * (b / a) = b|
    (implies (and (terminop a) (terminop b) (dividep a b))
             (equal (* a (/ b a)) b)))
  ```

  La demostración se realiza por inducción siguiendo el siguiente esquema. Sea $P(a, b) \equiv a \mid b \implies a \cdot (b/a) = b$:

  1. $b = [] \rightarrow P(a, b)$
  2. $(b \neq [] \wedge P(resto(a), resto(b))) \rightarrow P(a, b)$

- $a, b, c \in T[X]$, $b \mid a \implies (a/b) \cdot c = (a \cdot c)/b$

  ```
  (defthm |(a / b) * c = (a * c) / b|
    (implies (and (terminop b) (terminop c) (dividep b a))
             (equal (* (/ a b) c)
                    (/ (* a c) b))))
  ```

La demostración se realiza por inducción siguiendo el siguiente esquema. Sea $P(a,b,c) \equiv b \mid a \implies (a/b)\cdot c = (a\cdot c)/b$:

1. $(c = [] \vee a/b = []) \rightarrow P(a,b,c)$
2. $(c \neq [] \wedge a/b \neq [] \wedge P(resto(a), resto(b), resto(c))) \rightarrow P(a,b,c)$

- $a,b,c \in T[X],\ c \mid b \implies a\cdot(b/c) = (a\cdot b)/c$

```
(defthm |a * (b / c) = (a * b) / c|
  (implies (and (terminop a) (terminop b)
                (terminop c) (dividep c b))
           (equal (* a (/ b c))
                  (/ (* a b) c))))
```

La demostración se realiza por inducción siguiendo el siguiente esquema. Sea $P(a,b,c) \equiv c \mid b \implies a\cdot(b/c) = (a\cdot b)/c$:

1. $(b = [] \vee c = []) \rightarrow P(a,b,c)$
2. $(b \neq [] \wedge c \neq [] \wedge P(resto(a), resto(b), resto(c))) \rightarrow P(a,b,c)$

Definición 5.7. Sean $X^{[a_1,\ldots,a_k]}, X^{[b_1,\ldots,b_k]} \in T[X]$. El mínimo común múltiplo de ambos términos se define por:

$$mcm(X^{[a_1,\ldots,a_k]}, X^{[b_1,\ldots,b_k]}) = X^{[max(a_1,b_1),\ldots,max(a_k,b_k)]}$$

donde la función *max* calcula el máximo de dos números naturales. En ACL2,

```
(defun mcm (a b)
  (cond ((and (not (terminop a)) (not (terminop b)))
         (uno))
        ((not (terminop a)) b)
        ((not (terminop b)) a)
        ((endp a) b)
        ((endp b) a)
        (t
         (cons (max (first a) (first b))
               (mcm (rest a) (rest b))))))
```

Algunas de las propiedades necesarias sobre esta función son las siguientes:

- $a,b \in M_Q[X],\ mcm(a,b) = mcm(b,a)$

```
(defthm |mcm(a, b) = mcm(b, a)|
   (equal (mcm a b) (mcm b a)))
```

La demostración de esta propiedad se realiza por inducción siguiendo el siguiente esquema. Sea $P(a,b) \equiv mcm(a,b) = mcm(b,a)$:

1. $(a = [] \vee b = []) \rightarrow P(a,b)$
2. $(a \neq [] \wedge b \neq [] \wedge P(resto(a), resto(b))) \rightarrow P(a,b)$

- $a, b \in M_Q[X]$, $a \mid mcm(a,b)$

```
(defthm |dividep(a, mcm(a, b))|
  (implies (and (terminop a) (terminop b))
           (dividep a (mcm a b))))
```

La demostración se realiza por inducción siguiendo el siguiente esquema. Sea $P(a,b) \equiv a \mid mcm(a,b)$:

1. $a = [] \rightarrow P(a,b)$
2. $(a \neq [] \wedge primero(mcm(a,b)) < primero(a)) \rightarrow P(a,b)$
3. $(a \neq [] \wedge primero(a) \leq primero(mcm(a,b)) \wedge P(resto(a), resto(b))) \rightarrow P(a,b)$

- $a, b, c \in M_Q[X]$, $a \mid c \wedge b \mid c \implies mcm(a,b) \mid c$

```
(defthm |dividep(a, c) & dividep(b, c) => dividep(mcm(a, b),c)|
  (implies (and (terminop c) (dividep a c) (dividep b c))
           (dividep (mcm a b) c)))
```

La demostración se realiza por inducción siguiendo el siguiente esquema. Sea $P(a,b,c) \equiv a \mid c \wedge b \mid c \implies mcm(a,b) \mid c$:

1. $(a = [] \vee b = []) \rightarrow P(a,b,c)$
2. $(a \neq [] \wedge b \neq [] \wedge P(resto(a), resto(b), resto(c))) \rightarrow P(a,b,c)$

5.4. Pertenencia

Además será también necesario definir operaciones de pertenencia de monomios y términos a polinomios.

Definición 5.8. Sean m un monomio y $p = m_1 + \ldots + m_n$ un polinomio, entonces $m \in p$ si, y sólo si, existe un i $(1 \leq i \leq n)$ tal que $m = m_i$. En ACL2,

```
(defun en-monomio (m p)
  (cond ((nulop p)
         nil)
        ((equal m (primero p))
         t)
        (t
         (en m (resto p)))))
```

Algunas de las propiedades necesarias sobre la función anterior son las siguientes:

- $p \in Q[X] \wedge tp(m_1) \neq tp(m_2) \implies (m_1 \in p \iff m_1 \in (m_2 +_M^< p))$

```
(defthm |m1 en p & tp(m1) != tp(m2) => m1 en m2 +M p|
   (implies (and (polinomiop p) (en-monomio m1 p)
                 (not (equal (termino m2) (termino m1))))
            (en-monomio m1 (RAC-POL::+-monomio m2 p))))

(defthm |tp(m1) != tp(m2) & m1 en (m2 +M p) => m1 en p|
   (implies (and (not (equal (termino m1) (termino m2)))
                 (en-monomio m1 (RAC-POL::+-monomio m2 p)))
            (en-monomio m1 p)))
```

El primer teorema ACL2 (correspondiente a la implicación a la derecha) se demuestra siguiendo el siguiente esquema. Sea $P(m_1, m_2, p) \equiv m_1 \in p \wedge tp(m_1) \neq tp(m_2) \implies m_1 \in (m_2 +_M^< p)$):

1. $(m_2 = 0 \vee p = 0) \rightarrow P(m_1, m_2, p)$
2. $(m_2 \neq 0 \wedge p \neq 0 \wedge tp(p) \leq tp(m_2)) \rightarrow P(m_1, m_2, p)$
3. $(m_2 \neq 0 \wedge p \neq 0 \wedge tp(m_2) < tp(p) \wedge P(m_1, m_2, resto(p))) \rightarrow P(m_1, m_2, p)$

El segundo teorema (correspondiente a la implicación a la izquierda) se demuestra siguiente el siguiente esquema. Sea $P(m_1, m_2, p) \equiv tp(m_1) \neq tp(m_2) \wedge m_1 \in (m_2 +_M^< p) \implies m_1 \in p$:

1. $m_1 = mp(p) \rightarrow P(m_1, m_2, p)$

2. $(m_1 \neq mp(p) \wedge P(m_1, m_2, resto(p))) \rightarrow P(m_1, m_2, p)$

- $m \in p \implies (-m) \in (-p)$

```
(defthm |m en p => (- m) en (- p)|
  (implies (and (polinomiop p) (en-monomio m p))
           (en-monomio (RAC-MON::- m) (- p))))
```

La demostración se realiza por inducción siguiendo el siguiente esquema. Sea $P(m, p) \equiv m \in p \implies (-m) \in (-p)$:

1. $p = 0 \rightarrow P(m, p)$
2. $(p \neq 0 \wedge P(m, resto(p))) \rightarrow P(m, p)$

- $p, q \in Q[X] \wedge m_1 \in p \wedge m_2 \in q \wedge tp(m_1) = tp(m_2) \wedge cp(m_1) + cp(m_2) \neq 0 \implies (m_1 + m_2) \in (p + q)$

```
(defthm |m1 en p & m2 en q & m1 + m2 != 0 => (m1+m2) en (p+q)|
  (let ((c (RAC::+ (coeficiente m1) (coeficiente m2))))
    (implies (and (polinomiop p) (polinomiop q)
                  (en-monomio m1 p) (en-monomio m2 q)
                  (equal (termino m1) (termino m2))
                  (monomiop m1) (monomiop m2)
                  (not (equal c (RAC::nulo))))
             (en-monomio (RAC-MON::+ m1 m2) (+ p q)))))
```

La demostración se realiza por inducción siguiendo el siguiente esquema. Sea $P(m_1, m_2, p, q) \equiv m_1 \in p \wedge m_2 \in q \wedge tp(m_1) = tp(m_2) \wedge cp(m_1) + cp(m_2) \neq 0 \implies (m_1 + m_2) \in (p + q)$:

1. $p = 0 \rightarrow P(m_1, m_2, p, q)$
2. $(p \neq 0 \wedge m_1 = mp(p)) \rightarrow P(m_1, m_2, p, q)$
3. $(p \neq 0 \wedge m_1 \neq mp(p) \wedge P(m_1, m_2, resto(p), q)) \rightarrow P(m_1, m_2, p, q)$

Nótese que, en general, para demostrar estos teoremas hacen falta diversos lemas. En concreto, en el caso del último teorema (que se utiliza en la demostración del caso 3 del teorema 7.16) es necesario aplicar propiedades tales como:

- Sean $m \in M_Q[X]$ y $p \in Q[X]$; si $tp(m) \notin p$ entonces $m \notin p$.
- Sean $m \in M_Q[X]$ y $p \in Q[X]$; si $m \in q$ y m es mayor que los monomios de p entonces $m \in (p + q)$

- Sean m_1, $m_2 \in M_Q[X]$ y $p \in Q[X]$; si $m_1 \in p$ y $tp(m_1) \neq tp(m_2)$ entonces $m_1 \in (m_2 +^{<}_{M} p)$.
- Sean m_1, $m_2 \in M_Q[X]$ y $p \in Q[X]$; si $m_1 \in p$, $tp(m_1) = tp(m_2)$ y $cp(m_1) + cp(m_2) \neq 0$ entonces $(m_1 + m_2) \in (m_2 +^{<}_{M} p)$.
- Sean $m \in M_Q[X]$ y $p \in Q[X]$; si $tp(p) < tp(m)$ entonces m es un monomio mayor a todos los monomios de p.

Para la formulación de algunas de esas propiedades es necesario el concepto de pertenencia de un término a un polinomio.

Definición 5.9. Sean t un término y $p = m_1 + \ldots + m_n$ un polinomio, entonces $t \in p$ si, y sólo si, existe i $(1 \leq i \leq n)$ tal que $t = tp(m_i)$. En ACL2,

```
(defun en-termino (te p)
  (cond ((nulop p)
         nil)
        ((equal te (termino (primero p)))
         (primero p))
        (t (en-termino te (resto p)))))
```

Nótese que `en-termino` devuelve falso si el término que recibe como primer parámetro no está en el polinomio que recibe como segundo parámetro. En caso de que esté entonces devuelve el monomio correspondiente del polinomio.

De la misma forma que con la función `en-monomio` presentaremos sólo algunos de las propiedades que se necesitan demostrar sobre esta función.

- $p \in Q[X]$, $m \in M_Q[X]$ y p no tiene ningún término igual a $tp(m)$ entonces $m \notin p$.

  ```
  (defthm |!(tp(m) en p) => !(m en p)|
    (implies (and (polinomiop p)
                  (not (en-termino (termino m) p)))
             (not (en-monomio m p))))
  ```

  La demostración se realiza por inducción siguiendo el siguiente esquema. Sea $P(m,p) \equiv tp(m) \notin p \implies m \notin p$:

  1. $p = 0 \rightarrow P(m,p)$

2. $(p \neq 0 \wedge m = mp(p)) \rightarrow P(m,p)$
3. $(p \neq 0 \wedge m \neq mp(p) \wedge P(m, resto(p))) \rightarrow P(m,p)$

- $p, q \in Q[X] \wedge m \in p \wedge tp(m) \notin q \implies m \in (p+q)$

```
(defthm |m en p & !(tp(m) en q) => m en (p + q)|
  (implies (and (polinomiop p) (polinomiop q) (en-monomio m p)
                (not (en-termino (termino m) q)))
           (en-monomio m (+ p q))))
```

La demostración se realiza por inducción siguiendo el siguiente esquema. Sea $P(m,p) \equiv m \in p \wedge tp(m) \notin q \implies m \in (p+q)$:

1. $p = 0 \rightarrow P(m,p,q)$
2. $(p \neq 0 \wedge m = mp(p)) \rightarrow P(m,p,q)$
3. $(p \neq 0 \wedge m \neq mp(p) \wedge P(m, resto(p), q)) \rightarrow P(m,p,q)$

- $tp(m) \in p \implies tp(m_p) = tp(m)$, donde m_p es el monomio de p con el mismo término que el monomio m.

```
(defthm |tp(m(p) en p) => tp(m)|
  (implies (en-termino (termino m) p)
           (equal (termino (en-termino (termino m) p))
                  (termino m))))
```

La demostración se realiza por inducción siguiendo el siguiente esquema. Sea $P(m,p) \equiv tp(m) \in p \implies tp(m_p) = tp(m)$:

1. $p = 0 \rightarrow P(m,p)$
2. $(p \neq 0 \wedge m_p = mp(p)) \rightarrow P(m,p)$
3. $(p \neq 0 \wedge m_p \neq mp(p) \wedge P(m, resto(p))) \rightarrow P(m,p)$

- $tp(m) \in p \implies m_p \in p$, donde m_p es el monomio de p con el mismo término que el monomio m.

```
(defthm |tp(m) en p => m(p) en p|
  (implies (en-termino (termino m) p)
           (en-monomio (en-termino (termino m) p) p)))
```

La demostración se realiza por inducción siguiendo el siguiente esquema. Sea $P(m,p) \equiv tp(m) \in p \implies m_p \in p$:

1. $p = 0 \rightarrow P(m, p)$
2. $(p \neq 0 \wedge m_p = mp(p)) \rightarrow P(m, p)$
3. $(p \neq 0 \wedge m_p \neq mp(p) \wedge P(m, resto(p))) \rightarrow P(m, p)$

- $p \in Q[X] \wedge -m \in p \implies tp(m) \notin m +^{<}_{M} p$

```
(defthm |- m en p => !(tp(m) en (m +M p)|
  (implies (and (monomiop m) (polinomiop p)
                (en-monomio (RAC-MON::- m) p))
           (not (en-termino (termino m)
                            (RAC-POL::+-monomio m p))))))
```

La demostración se realiza por inducción siguiendo el siguiente esquema. Sea $P(m, p) \equiv -m \in p \implies tp(m) \notin m +^{<}_{M} p$:

1. $(p = 0 \vee m = 0) \rightarrow P(m, p)$
2. $(p \neq 0 \wedge m \neq 0 \wedge tp(p) \leq tp(m)) \rightarrow P(m, p)$
3. $(p \neq 0 \wedge \wedge m \neq 0 \wedge tp(m) < tp(p) \wedge P(m, resto(p))) \rightarrow P(m, p)$

Por último, también definimos la pertenencia de un polinomio a un conjunto finito de polinomios.

Definición 5.10. Sean p un polinomio y $F = \{f_1, \ldots, f_n\}$ un conjunto de polinomios, entonces $p \in F$ si, y sólo si, existe un i $(1 \leq i \leq n)$ tal que $p = f_i$. En ACL2,

```
(defun en (p F)
  (cond ((atom F)
         nil)
        ((equal p (first F))
         t)
        (t
         (en p (rest F)))))
```

Hay que resaltar la cantidad de lemas auxiliares necesarios sobre las operaciones de términos, monomios y polinomios. Entre ellos están los teoremas de tipos que también son necesarios sobre la mayoría de las operaciones.

En concreto, y por poner un ejemplo de este tipo de teoremas, nótese que sobre la función anterior se tiene que:

- Si F es un conjunto finito de polinomios y $p \in F$, entonces $p \in Q[X]$

```
(defthm |polinomiosp(F) & en(p, F) => polinomiop(p)|
  (implies (and (polinomiosp F) (en p F))
           (polinomiop p)))
```

donde `polinomiosp` se define de la siguiente forma:

```
(defun polinomiosp (F)
  (if (atom F)
      (equal F nil)
    (and (polinomiop (first F))
         (polinomiosp (rest F)))))
```

La demostración se realiza por inducción siguiendo el siguiente esquema. Sea $P(p, F) \equiv F \subset Q[X] \wedge p \in F \implies p \in Q[X]$:

1. $F = [] \rightarrow P(p, F)$
2. $(F \neq [] \wedge p = primero(F)) \rightarrow P(m, p)$
3. $(F \neq [] \wedge p \neq primero(F) \wedge P(p, resto(F))) \rightarrow P(p, F)$

- Si F es un conjunto finito de polinomios y $p \neq 0 \in F$, entonces $mp(p) \in M_Q[X]$

```
(defthm |polinomiosp(F) & p en F & p != 0 => monomiop(mp(p))|
  (implies (and (polinomiosp F) (en p F) (not (nulop p)))
           (monomiop (primero p))))
```

La demostración ACL2 de este teorema es inmediata utilizando la propiedad anterior y que el monomio principal de un polinomio no nulo es un monomio.

5.5. Uniformidad

Otro problema se presenta en el hecho de que la formalización en la que nos apoyamos para demostrar la parada del algoritmo de Buchberger (véase el capítulo 8) trabaja con secuencias de términos *uniformes*, esto es, secuencias de términos con el mismo número de variables. Esto hace que haya que introducir el concepto de polinomio uniforme y demostrar que cada una de las operaciones sobre polinomios preservan la uniformidad. Su formalización se encuentra en el fichero `k-polinomio.lisp`.

Nótese que la implementación hecha en el capítulo 3 de los polinomios es más flexible y permite que los términos, y consecuentemente los monomios, que formen un determinado polinomio puedan tener un número diferente de variables. Es en la implementación de las operaciones que trabajan con polinomios donde se tiene en cuenta esta posibilidad, extendiéndose sus comportamientos de manera adecuada para que mantengan las propiedades de anillo.

La uniformidad de la representación y el hecho de que el lenguaje sea aplicativo (es decir, no hay variables globales) introducen la necesidad de un parámetro adicional en cada función que opere con polinomios. Este parámetro puede obviarse acordando de antemano un número fijo de variables. De aquí en adelante, notaremos por k ese número fijo de variables.

5.5.1. Polinomios uniformes

El concepto de polinomio uniforme de k variables se implementa en ACL2 con la función `k-uniformep` y el de polinomio normalizado uniforme de k variables con la macro `k-polinomiop`. Recuérdese que `polinomiop` es el reconocedor de polinomios normalizados que lo que hace es comprobar si el objeto que se le pasa es una lista de monomios no nulos en un orden estrictamente decreciente.

```
(defun<k> k-uniformep (p)
  (if (nulop p)
      (equal p (nulo))
    (and (k-monomiop (primero p))
         (k-uniformep (resto p)))))

(defmacro k-polinomiop (p)
  `(and (polinomiop ,p)
        (k-uniformep ,p)))
```

En estas definiciones se exige que explícitamente aparezcan k variables en los monomios de los polinomios. Para ello se ha creado un nuevo evento en el sistema, `defun<k>`, con el que se introducen nuevos nombres de función fijando el número de variables a k. En concreto, el evento `defun<k>` anterior se transforma en los dos siguientes, permitiendo no tener que estar pasando la variable extra k cada vez que se haga referencia a estas funciones. Con esto se consigue una mayor claridad en la exposición de los teoremas y legibilidad del código resultante.

```
(defmacro k-uniformep (p)
  `(k-uniformep-k ,p k))

(defun k-uniformep-k (p k)
  (if (nulop p)
      (equal p (nulo))
    (and (k-monomiop (primero p))
         (k-uniformep-k (resto p) k))))
```

Por otro lado, un monomio es uniforme de k variables si es un monomio cuyo término es uniforme de k variables.

```
(defmacro k-monomiop (m)
  `(and (monomiop ,m)
        (k-terminop (termino ,m))))
```

Por último, `k-terminop` comprueba si el objeto que se le pasa es un término de k variables. Este reconocedor se implementa en ACL2 mediante la macro siguiente.

```
(defmacro k-terminop (te)
  `(k-terminop-k ,te k))

(defun k-terminop-k (te k)
  (cond ((zp k) (null te))
        ((not (naturalp (first te))) nil)
        (T (k-terminop-k (rest te) (1- k)))))
```

Nótese que aquí no podemos emplear el evento `defun<k>` ya que la función `k-terminop-k` modifica k en cada llamada recursiva y, por tanto, no sigue el patrón para poder emplear `defun<k>`.

Lo mismo ocurre cuando se define el término nulo de de k variables:

```
(defmacro k-termino-nulo ()
  `(k-termino-nulo-k k))

(defun k-termino-nulo-k (k)
  (cond ((not (naturalp k)) nil)
        ((zp k) nil)
        (t (cons 0 (k-termino-nulo-k (ACL2::- k 1))))))
```

Esta macro se utiliza para definir el monomio identidad de k variables que, a su vez, se usa para definir el polinomio identidad de k variables.

```
(defun<k> k-monomio-identidad ()
  (monomio (RAC::identidad) (k-termino-nulo)))

(defun<k> k-identidad ()
  (polinomio (k-monomio-identidad)))
```

Además como habitualmente trabajamos con secuencias de polinomios, definimos también la secuencia de polinomios uniformes de k variables de la siguiente forma.

```
(defun<k> k-uniformesp (F)
  (if (endp F)
      (equal F nil)
    (and (k-uniformep (first F))
         (k-uniformesp (rest F)))))

(defmacro k-polinomiosp (F)
  `(and (polinomiosp ,F)
        (k-uniformesp ,F)))
```

Nótese que a partir de ahora trabajaremos con polinomios normalizados y uniformes de k variables sobre el cuerpo de coeficientes de los números racionales.

Por tanto, $mp(p)$ hará referencia al monomio principal de p con respecto a $<_M$ y $resto(p)$ hará referencia al resto del polinomio.

5.5.2. Propiedades

Demostraremos ahora que cada una de las operaciones sobre polinomios preservan la uniformidad. Los teoremas fundamentales sobre las operaciones del anillo de polinomios se exponen a continuación:

- Uniformidad de la forma normal de un polinomio uniforme.

```
(defthm |k-uniformep(p) => k-uniformep(fn(p))|
  (implies (k-uniformep p)
           (k-uniformep (fn p))))
```

La demostración se realiza por inducción siguiendo el siguiente esquema. Sea $P(p) \equiv k\text{-}uniformep(p) \implies k\text{-}uniformep(fn(p))$:

1. $p = 0 \rightarrow P(p)$
2. $(p \neq 0 \wedge P(resto(p))) \rightarrow P(p)$

- Uniformidad de la suma desnormalizada de dos polinomios uniformes.

```
(defthm |k-uniformep(p) & k-uniformep(q)=> k-uniformep(p +D q)|
  (implies (and (k-uniformep p) (k-uniformep q))
           (k-uniformep (RAC-POL::+ p q))))
```

La demostración se realiza por inducción siguiendo el siguiente esquema. Sea $P(p,q) \equiv k\text{-}uniformep(p) \wedge k\text{-}uniformep(q) \Rightarrow k\text{-}uniformep(p +_d q)$:

1. $p = 0 \rightarrow P(p,q)$
2. $(p \neq 0 \wedge P(resto(p),q)) \rightarrow P(p,q)$

- Uniformidad de la suma de dos polinomios uniformes.

```
(defthm |k-uniformep(p) & k-uniformep(q)=> k-uniformep(p + q)|
  (implies (and (k-uniformep p) (k-uniformep q))
           (k-uniformep (+ p q))))
```

La demostración en ACL2 de esta propiedad es inmediata utilizando las dos propiedades anteriores.

- Uniformidad del producto desnormalizado de dos polinomios uniformes.

```
(defthm |k-uniformep(p) & k-uniformep(q) => k-uniformep(p *D q)|
  (implies (and (k-uniformep p) (k-uniformep q))
           (k-uniformep (RAC-POL::* p q))))
```

La demostración se realiza por inducción siguiendo el siguiente esquema. Sea $P(p,q) \equiv k\text{-}uniformep(p) \wedge k\text{-}uniformep(q) \Rightarrow k\text{-}uniformep(p \cdot_d q)$:

1. $p = 0 \rightarrow P(p,q)$
2. $(p \neq 0 \wedge P(resto(p),q)) \rightarrow P(p,q)$

- Uniformidad del producto de dos polinomios uniformes.

```
(defthm |k-uniformep(p) & k-uniformep(q) => k-uniformep(p * q)|
  (implies (and (k-uniformep p) (k-uniformep q))
           (k-uniformep (* p q))))
```

La demostración en ACL2 de esta propiedad es inmediata utilizando la propiedad anterior y la uniformidad de la forma normal de un polinomio uniforme.

- Uniformidad del opuesto sin normalización de un polinomio uniforme.

```
(defthm |k-uniformep(p) => k-uniformep(-D p)|
  (implies (k-uniformep p)
           (k-uniformep (RAC-POL::- p))))
```

La demostración se realiza por inducción siguiendo el siguiente esquema. Sea $P(p) \equiv k\text{-}uniformep(p) \implies k\text{-}uniformep(-_d\ p)$:

1. $p = 0 \rightarrow P(p)$
2. $(p \neq 0 \wedge P(resto(p))) \rightarrow P(p)$

- Uniformidad del opuesto de un polinomio uniforme.

```
(defthm |k-uniformep(p) => k-uniformep(- p)|
  (implies (k-uniformep p)
           (k-uniformep (- p))))
```

La demostración en ACL2 de esta propiedad es inmediata utilizando la propiedad anterior y la uniformidad de la forma normal de un polinomio uniforme.

Estos teoremas se demuestran por inducción sobre la estructura de las operaciones y ayudados por algunos lemas.

Como adicionalmente utilizaremos las operaciones de producto, división y mínimo común múltiplo de términos, así como el opuesto y la división de monomios será necesario demostrar también que estas funciones preservan la uniformidad:

- Uniformidad de la división de dos términos uniformes y divisibles.

```
(defthm |k-termsp(t1 & t2) & divp(t2, t1) => k-termp(t1 / t2)|
  (implies (and (k-terminop t1) (k-terminop t2)
                (dividep t2 t1))
           (k-terminop (/ t1 t2))))
```

La demostración se realiza por inducción siguiendo el siguiente esquema. Sea $P(t_1, t_2) \equiv k\text{-}terminop(t_1) \wedge k\text{-}terminop(t_2) \wedge t_2 \mid t_1 \implies k\text{-}terminop(t_1/t_2)$:

1. $(t_1 = [] \vee t_2 = []) \rightarrow P(t_1, t_2)$
2. $(t_1 \neq [] \wedge t_2 \neq [] \wedge P(resto(t_1), resto(t_2))) \rightarrow P(t_1, t_2)$

- Uniformidad del mínimo común múltiplo de dos términos uniformes.

```
(defthm |k-terminop(mcm(t1, t2))|
  (implies (and (k-terminop t1) (k-terminop t2))
           (k-terminop (mcm t1 t2))))
```

La demostración se realiza por inducción siguiendo el siguiente esquema. Sea $P(t_1, t_2) \equiv k\text{-}terminop(t_1) \wedge k\text{-}terminop(t_2) \wedge t_2 \mid t_1 \implies k\text{-}terminop(mcm(t_1, t_2))$:

1. $(t_1 = [] \vee t_2 = []) \rightarrow P(t_1, t_2)$
2. $(t_1 \neq [] \wedge t_2 \neq [] \wedge P(resto(t_1), resto(t_2))) \rightarrow P(t_1, t_2)$

- Uniformidad del opuesto de la división de dos monomios uniformes y divisibles.

```
(defthm |k-monsp(m1 & m2) & divp(tm2, tm1) => k-monp(-m1/m2))|
  (implies (and (k-monomiop m1) (k-monomiop m2)
                (dividep (termino m2) (termino m1)))
           (k-monomiop (- (/ m1 m2)))))
```

La demostración en ACL2 de esta última propiedad es inmediata utilizando la propiedad anterior y que el término de un monomio es igual al de su opuesto.

5.6. Resumen

En este capítulo:

- Hemos instanciado el anillo de polinomios con coeficientes abstractos para obtener una implementación ejecutable y verificada de los polinomios racionales.

- Hemos extendido los términos para incluir conceptos relacionados con la división.
- Hemos desarrollado el concepto de polinomio uniforme como una forma de asegurar que se emplean siempre las mismas variables.
- Hemos desarrollado funciones para comprobar la pertenencia de términos y monomios a polinomios y de polinomios a listas de polinomios, demostrando algunas de sus propiedades más importantes.
- Hemos demostrado que la uniformidad se preserva por todas las operaciones polinómicas desarrolladas.

Por otro lado, nótese que los polinomios abstractos construidos y que, en este capítulo, se han instanciados para obtener los polinomios racionales se pueden utilizar para desarrollar cualquier otro anillo de polinomios.

En concreto, esta formalización se ha instanciado también para obtener polinomios booleanos [35, 67], que se han utilizado con éxito para verificar un procedimiento de decisión para la lógica proposicional basado en polinomios [68].

Parte III

Ideales y reducciones polinómicas

Capítulo 6

Ideales polinómicos

6.1. Introducción

La siguiente estructura algebraica a formalizar en nuestro camino hacia la implementación y verificación del algoritmo de Buchberger es la de ideal polinómico. Esta estructura juega un papel muy importante en la verificación de este algoritmo, ya que una propiedad que ha de permanecer invariante a lo largo de su ejecución es que no se altere el ideal de la base inicial.

Nótese que los polinomios racionales normalizados obtenidos en el capítulo anterior son los que constituyen la base para formalizar estos ideales polinómicos.

En primer lugar, se presentará el problema más importante que resuelven las bases de Gröbner: el de la pertenencia al ideal.

A continuación se define en ACL2 el concepto de ideal polinómico y se demuestra que cumple sus propiedades fundamentales. La formalización se encuentra en el fichero `ideal.lisp`.

6.2. Pertenencia a ideales polinómicos

Definición 6.1. Sea $\langle R, +, \cdot, 0, 1\rangle$ un anillo e $I \subseteq R$. Se dice que I es un *ideal* de R si permanece cerrado bajo la suma y bajo el producto por elementos de

R. Es decir:

$$\begin{aligned} a, b \in I &\implies a + b \in I \\ a \in I \wedge b \in R &\implies a \cdot b \in I \end{aligned}$$

Definición 6.2. Sea $a \in R$ y $B, C \subseteq R$. Se dice que a es *combinación lineal* de los elementos de B con coeficientes en C, si existen $b_1, \ldots, b_n \in B$ y $c_1, \ldots, c_n \in C$ tales que

$$a = \sum_{i=1}^{n} c_i \cdot b_i$$

Definición 6.3. Sea $B \subseteq R$. El *ideal generado* por B (notado por $\langle B \rangle$), es el conjunto de las combinaciones lineales de los elementos de B con coeficientes en R.

Definición 6.4. Sea $I \subseteq R$ un ideal. Decimos que $B \subseteq R$ es una base de I si $I = \langle B \rangle$.

Definición 6.5. Se dice que el ideal I está *finitamente generado* si tiene una base finita.

En lo sucesivo, sea p un polinomio y C y F secuencias de polinomios.

Definición 6.6 (combinación lineal). Se define la combinación lineal de los elementos de $F = \langle f_1, \ldots, f_n \rangle$ con coeficientes en $C = \langle c_1, \ldots, c_n \rangle$, $cl(C, F)$, como sigue.

$$cl(C, F) = c_1 \cdot f_1 + \ldots + c_n \cdot f_n = \sum_{i=1}^{n} c_i \cdot f_i$$

En ACL2,

```
(defun combinacion-lineal (C F)
  (if (or (atom C) (atom F))
      (nulo)
    (+ (* (first C) (first F))
       (combinacion-lineal (rest C) (rest F)))))
```

Definición 6.7 (pertenencia a un ideal). Se define la pertenencia de p al ideal engendrado por F, $\langle F \rangle$, mediante el siguiente predicado, $\in$:

$$p \in \langle F \rangle \equiv \exists C \; p = cl(C, F)$$

Nótese que se podría haber ocultado la aparición del cuantificador existencial empleando en lugar de $\in$ el siguiente predicado, $\in_C$:

$$p \in_C \langle F \rangle \equiv p = cl(C, F)$$

Decimos que C en la definición anterior es un «testigo» de la pertenencia de p a $\langle F \rangle$. Ciertamente, se cumple que:

$$p \in \langle F \rangle \equiv \exists C \; p \in_C \langle F \rangle.$$

Para la formalización en ACL2 introducimos una función de Skolem, restringida por axiomas a devolver una lista de coeficientes testigo de la pertenencia al ideal.

```
(defun-sk<k> en-ideal (p F)
  (exists (C) (and (k-polinomiosp C)
                   (equal p (combinacion-lineal C F)))))
```

En esta definición se exige que explícitamente aparezcan k variables en los monomios de los polinomios. Para ello se ha creado un nuevo evento en el sistema, `defun-sk<k>`, con el que se pueden introducir funciones de Skolem análogamente a como lo hace `defun-sk` pero añadiendo un parámetro adicional, k.

Más precisamente, el evento `defun-sk<k>` anterior equivale a un encapsulado en el que se define mediante `defchoose` una función de elección restringida por axiomas a devolver un testigo de la existencia de los polinomios en `C`. Esta función se emplea a través de una macro, `en-ideal-witness`, que recibe `p` y `F` como parámetros y que emplea la función de elección para obtener el testigo de la pertenencia al ideal. Es decir, afirmar que $p \in \langle F \rangle$ equivale a decir que $p = cl(C, F)$ donde $C = \textit{en-ideal-witness}(p, F)$.

6.3. Estabilidad del ideal respecto a las operaciones

En esta sección se demuestra la estabilidad del ideal respecto a las operaciones del anillo de polinomios. Esta propiedad nos será de utilidad en los capítulos sucesivos. Empezaremos, en primer lugar, por la operación suma. Se demuestra que la suma de polinomios preserva la pertenencia al ideal con la ayuda del lema 6.9.

Definición 6.8. Sean $C_p = \langle p_1, \ldots, p_n \rangle$ y $C_q = \langle q_1, \ldots, q_n \rangle$ secuencias de polinomios de la misma longitud n. Se define la suma de secuencias de polinomios, notada por C_+, de la siguiente forma.

$$C_+(C_p, C_q) = \langle p_1 + q_1, \ldots, p_n + q_n \rangle$$

En ACL2,

```
(defun C+ (Cp Cq)
  (declare (xargs :measure (LISP::+ (len Cp) (len Cq))))
  (if (and (atom Cp) (atom Cq))
      nil
    (cons (+ (if (atom Cp) (nulo) (first Cp))
             (if (atom Cq) (nulo) (first Cq)))
          (C+ (rest Cp) (rest Cq)))))
```

Nótese que para que esta función sea admitida por el sistema ha sido necesario proporcionarle explícitamente la medida que decrece en cada llamada recursiva de la función. Con la medida suministrada, ACL2 demuestra la parada de la función sin ningún problema.

Lema 6.9. *Sean C_p, C_q y F secuencias de polinomios de la misma longitud. Entonces,*

$$cl(C_+(C_p, C_q), F) = cl(C_p, F) + cl(C_q, F)$$

En ACL2,

```
(defthm |cl(C+(Cp, Cq), F) = cl(Cp, F) + cl(Cq, F)|
  (equal (combinacion-lineal (C+ Cp Cq) F)
         (+ (combinacion-lineal Cp F) (combinacion-lineal Cq F))))
```

Demostración. Por inducción según el siguiente esquema. Sea $P(C_p, C_q, F) \equiv cl(C_+(C_p, C_q), F) = cl(C_p, F) + cl(C_q, F)$:

1. $(C_p = [] \vee C_q = []) \rightarrow P(C_p, C_q, F)$
2. $(C_p \neq [] \wedge C_q \neq [] \wedge P(resto(C_p), resto(C_q), resto(F))) \rightarrow P(C_p, C_q, F)$

Caso base: $C_p = [] \vee C_q = []$

Inmediata por la definición de C_+ y cl y porque $0 + p = p$.

Paso de inducción:

Por hipótesis de inducción:

$$cl(C_+(resto(C_p), resto(C_q)), resto(F)) = \\ cl(resto(C_p), resto(F)) + cl(resto(C_q), resto(F))$$

Sean $C_p = \langle p_1, \ldots, p_n \rangle$, $C_q = \langle q_1, \ldots, q_n \rangle$ y $F = \langle f_1, \ldots, f_n \rangle$ entonces

$$cl(C_+(C_p, C_q), F) = \\ (p_1 + q_1) \cdot f_1 + cl(C_+(resto(C_p), resto(C_q)), resto(F)) = \\ (p_1 + q_1) \cdot f_1 + cl(resto(C_p), resto(F)) + cl(resto(C_q), resto(F)) = \\ p_1 \cdot f_1 + q_1 \cdot f_1 + cl(resto(C_p), resto(F)) + cl(resto(C_q), resto(F)) = \\ p_1 \cdot f_1 + cl(resto(C_p), resto(F)) + q_1 \cdot f_1 + cl(resto(C_q), resto(F)) = \\ cl(C_p, F) + cl(C_q, F)$$

por definición de C_+ y cl, por hipótesis de inducción y por propiedades de polinomios.

□

Para demostrar este teorema en ACL2 es necesario suministrarle el esquema de inducción que debe utilizar.

```
(defun induccion (Cp Cq F)
  (if (and (atom Cp) (atom Cq))
      F
    (induccion (rest Cp) (rest Cq) (rest F))))
```

La suma de polinomios preserva la pertenencia al ideal.

Teorema 6.10 (clausura del ideal respecto a la suma). *Sean p y q polinomios, y F una secuencia de polinomios. Entonces,*

$$p \in \langle F \rangle \wedge q \in \langle F \rangle \implies p + q \in \langle F \rangle$$

En ACL2,

```
(defthm |p en <F> & q en <F> => p + q en <F>|
  (implies (and (en-ideal p F) (en-ideal q F))
           (en-ideal (+ p q) F)))
```

Demostración.

$$
\begin{array}{ll}
p \in \langle F \rangle \wedge q \in \langle F \rangle & \\
\quad \Longrightarrow \exists C_p\ p = cl(C_p, F) \wedge \exists C_q\ q = cl(C_q, F) & \text{(def. pertenencia a ideal)} \\
\quad \Longrightarrow p + q = cl(C_+(C_p, C_q), F) & \text{(lema 6.9)} \\
\quad \Longrightarrow p + q \in \langle F \rangle & \text{(def. pertenencia a ideal)}
\end{array}
$$

□

Nótese que $C_+(C_p, C_q)$ es el testigo de la pertenencia de $p+q$ al ideal generado por F.

A continuación se demuestra que el producto de un polinomio del ideal por otro polinomio preserva la pertenencia al ideal. Para la operación producto se sigue el mismo esquema que para la suma. En primer lugar se define una función que calcula el testigo correspondiente de la pertenencia al ideal, se demuestra el lema 6.12 y finalmente se demuestra el teorema deseado.

Definición 6.11. Sean p un polinomio y $C = \langle c_1, \ldots, c_n \rangle$ una secuencia de polinomios. Se define el producto de un polinomio por una secuencia de polinomios de la siguiente forma.

$$C_*(p, C) = \langle p \cdot c_1, \ldots, p \cdot c_n \rangle$$

En ACL2,

```
(defun C* (p C)
  (declare (xargs :verify-guards nil))
  (if (atom C)
      nil
    (cons (* p (first C)) (C* p (rest C)))))
```

Lema 6.12. *Sean p un polinomio y C y F secuencias de polinomios de la misma longitud. Entonces,*

$$cl(C_*(p, C), F) = p \cdot cl(C, F)$$

En ACL2,

```
(defthm |cl(C*(p, C)) = p * cl(C, F)|
  (equal (combinacion-lineal (C* p C) F)
         (* p (combinacion-lineal C F))))
```

Demostración. Por inducción según el esquema siguiente. Sea $P(C, F, p) \equiv cl(C_*(p, C), F) = p \cdot cl(C, F)$:

1. $(C = [] \vee F = []) \rightarrow P(C, F, p)$
2. $(C \neq [] \wedge F \neq [] \wedge P(resto(C), resto(F), p)) \rightarrow P(C, F, p)$

Caso base: $C = [] \vee F = []$

Inmediata por la definición de C_* y cl y porque $0 \cdot p = 0$ y $p \cdot q = q \cdot p$.

Paso de inducción:

Por hipótesis de inducción:

$$cl(C_*(p, resto(C)), resto(F)) = p \cdot cl(resto(C), resto(F))$$

Sean $C = \langle c_1, \ldots, c_n \rangle$ y $F = \langle f_1, \ldots, f_n \rangle$ entonces

$$\begin{array}{ll}
cl(C_*(p, C), F) = & \\
(p \cdot c_1) \cdot f_1 + cl(C_*(p, resto(C)), resto(F)) = & (\text{def. } C_* \text{ y } cl) \\
(p \cdot c_1) \cdot f_1 + p \cdot cl(resto(C), resto(F)) = & (\text{hip. inducción}) \\
p \cdot (c_1 \cdot f_1) + p \cdot cl(resto(C), resto(F)) = & (\text{prop. asociativa de } \cdot) \\
p \cdot (c_1 \cdot f_1 + cl(resto(C), resto(F))) = & (\text{prop. distributiva}) \\
p \cdot cl(C, F) & (\text{def. } cl)
\end{array}$$

□

El producto de un polinomio del ideal por otro polinomio preserva la pertenencia al ideal.

Teorema 6.13 (clausura del ideal respecto al producto). *Sean p y q polinomios y F una secuencia de polinomios. Entonces,*

$$q \in \langle F \rangle \implies p \cdot q \in \langle F \rangle$$

En ACL2,

```
(defthm |q en <F> => p * q en <F>|
  (implies (and (k-polinomiop p)
                (en-ideal q F))
           (en-ideal (* p q) F)))
```

Demostración.

$$
\begin{array}{ll}
q \in \langle F \rangle & \\
\implies \exists C\ q = cl(C, F) & \text{(def. pertenencia a ideal)} \\
\implies p \cdot q = p \cdot cl(C, F) = cl(C_*(p, C), F) & \text{(lema 6.12)} \\
\implies p \cdot q \in \langle F \rangle & \text{(def. pertenencia a ideal)}
\end{array}
$$

□

Nótese que $C_*(p, C)$ es el testigo de la pertenencia de $p \cdot q$ al ideal generado por F.

6.4. Inclusión de la base

A continuación se prueba que $\langle F \rangle$ contiene a F.

Definición 6.14. Sean $F = \langle f_1, \ldots, f_n \rangle$ una secuencia de polinomios y $p \in F$. Entonces, $\exists i\ p = f_i$, $1 \leq i \leq n$.

Los coeficientes testigos de la pertenencia de p al ideal generado por F, $C_b(p, F) = \langle c_1, \ldots, c_n \rangle$, vienen dados como sigue:

$$c_i = 1 \wedge c_j = 0, \forall j \neq i, 1 \leq j \leq n$$

En ACL2,

```
(defun<k> Cb (p F)
  (cond ((atom F)
         nil)
        ((equal p (first F))
         (cons (k-identidad) (C0 (rest F))))
        (t
         (cons (nulo) (Cb p (rest F))))))
```

donde la función `C0` se define de la siguiente forma:

```
(defun C0 (F)
 (if (atom F)
     nil
   (cons (nulo) (C0 (rest F)))))
```

Lema 6.15. *Sea F una secuencia de polinomios. La combinación lineal de una lista de polinomios con coeficientes nulos es siempre el polinomio nulo. Es decir:*

$$cl(C_0(F), F) = 0$$

donde $C_0(F)$ es una secuencia de la misma longitud que F de polinomios nulo.

En ACL2,

```
(defthm |cl(C0(F), F) = 0|
  (equal (combinacion-lineal (C0 F) F) (nulo)))
```

Demostración. Por inducción sobre la estructura de definición de C_0.

Caso base: $F = []$

Inmediata por la definición de C_0 y cl.

Paso de inducción:

Por hipótesis de inducción:

$$cl(C_0(resto(F)), resto(F)) = 0$$

Sean $C = \langle c_1, \ldots, c_n \rangle$ y $F = \langle f_1, \ldots, f_n \rangle$ entonces

$$\begin{array}{ll}
cl(C_0(F), F) = & \\
0 \cdot f_1 + cl(C_0(resto(F)), resto(F)) = & (\text{def. } cl) \\
0 \cdot f_1 + 0 = & (\text{hip. inducción}) \\
0 & (0 + p = p \text{ y } 0 \cdot p = 0)
\end{array}$$

□

Lema 6.16. *Sean p un polinomio, F una secuencia de polinomios y $p \in F$. Entonces,*

$$cl(C_b(p, F), F) = p$$

En ACL2,

```
(defthm |p en F => cl(Cb(p, F), F) = p|
  (implies (and (k-polinomiop p)
                (en p F))
           (equal (combinacion-lineal (Cb p F) F) p)))
```

Demostración. Por inducción sobre la estructura de la definición de C_b.

Caso base 1: $F = []$

Imposible porque $p \in F$.

Caso base 2: $p = f_1$

$$\begin{aligned}
&cl(C_b(p, F), F) \\
&= 1 \cdot f_1 + cl(C_0(resto(F)), resto(F)) && \text{(def. } C_b \text{ y } cl) \\
&= 1 \cdot f_1 + 0 && \text{(lema 6.15)} \\
&= f_1 && \text{(prop. polinomios)} \\
&= p && (p = f_1)
\end{aligned}$$

Paso de inducción:

$F \neq []$, $p \neq f_1$ y por hipótesis de inducción:

$$cl(C_b(p, resto(F)), resto(F)) = p$$

Sea $F = \langle f_1, \ldots, f_n \rangle$ entonces

$$\begin{aligned}
&cl(C_b(p, F), F) \\
&= 0 \cdot f_1 + cl(C_b(p, resto(F)), resto(F)) && \text{(def. } C_b \text{ y } cl) \\
&= 0 + cl(C_b(p, resto(F)), resto(F)) && (0 \cdot p = 0) \\
&= 0 + p && \text{(hip. inducción)} \\
&= p && (0 + p = p)
\end{aligned}$$

□

Teorema 6.17. *Cualquier elemento de la base pertenece al ideal.*

Sean p un polinomio y F una secuencia de polinomios.

$$p \in F \implies p \in \langle F \rangle$$

En ACL2,

```
(defthm |p en F => p en <F>|
  (implies (and (k-polinomiop p)
                (en p F))
           (en-ideal p F))
```

Demostración. Inmediata por la definición de pertenencia a ideal y por el lema 6.16. □

6.5. Congruencia inducida por el ideal

En el fichero `congruencia-ideal.lisp` se formaliza la congruencia inducida por un ideal.

Definición 6.18. Dado un anillo de polinomios conmutativo R, la *congruencia* $\equiv_{\langle F \rangle}$ inducida por el ideal $\langle F \rangle$ se define por,

$$p \equiv_{\langle F \rangle} q \iff p - q \in \langle F \rangle$$

En ACL2,

```
(defun<k> =<> (p q F)
  (en-ideal (+ p (- q)) F))
```

6.6. Resumen

En este capítulo:

- Hemos introducido la noción de ideal y formalizado la pertenencia de un polinomio a un ideal.

- Hemos presentado la formalización en ACL2 del concepto de ideal generado por un conjunto finito de polinomios.
- Hemos definido la congruencia inducida por un ideal.
- Hemos comprobado que nuestra definición de ideal en ACL2 cumple con la habitual al demostrar que es cerrado bajo las operaciones de anillo.

Capítulo 7

Reducciones polinómicas

7.1. Introducción

En este capítulo presentamos un tipo particular de reducción sobre polinomios y su formalización en ACL2.

El objetivo es definir ciertas condiciones bajo las cuales se decide $\equiv_{\langle F \rangle}$. Daremos una visión alternativa de $\equiv_{\langle F \rangle}$ basada en reducciones que nos permitirá aplicarle resultados generales de reducciones abstractas y, en particular, permite reutilizar los resultados de [85, 86, 87].

Principalmente, se demuestra que la reducción definida es noetheriana respecto al orden de polinomios subyacente y que la relación de equivalencia inducida coincide con la congruencia inducida por el ideal. Además, se lleva a cabo el cálculo de formas normales y la demostración de propiedades tales como que los ideales permanecen cerrados bajo ese cálculo.

7.2. Reducciones abstractas

Para fijar la notación, definiremos, en primer lugar, algunos conceptos básicos, sobre reducciones abstractas. Consúltese [1] para una descripción detallada.

Definición 7.1. Una relación de reducción sobre un conjunto A es una relación binaria sobre A.

Las relaciones de reducción suelen representarse en notación de flecha, de-

jando el conjunto implícito. En particular, escribiremos $a \to b$ para expresar que $(a, b) \in \to \subseteq A \times A$, para un cierto conjunto A deducible por el contexto. Decimos entonces que a *se reduce a* b (a través de $\to$).

Nota. Dadas dos relaciones de reducción, $\to_1$ y $\to_2$, escribimos $a \to_1 b \to_2 c$ para indicar que $a \to_1 b$ y $b \to_2 c$.

Definición 7.2. Dadas dos relaciones de reducción, $\to_1$ y $\to_2$ sobre un conjunto A, se define su composición, $\to_1 \circ \to_2$, como:

$$\to_1 \circ \to_2 = \{(a, b) \in A \times A \mid \exists c \in A \; a \to_1 c \to_2 b\}$$

Definición 7.3. Dada una relación de reducción $\to$ definimos su n-ésima potencia, con $n \in N$, como:

$$\to^n = \begin{cases} \{(a, a) \in A \times A\}, & \text{si } n = 0 \\ \to \circ \to^{n-1}, & \text{si } n > 0 \end{cases}$$

Definición 7.4. Dada una relación de reducción $\to$ definimos las siguientes relaciones de reducción derivadas:

$$\begin{aligned} \to^+ &= \bigcup_{n>0} \to^n && \text{(clausura transitiva)} \\ \to^* &= \bigcup_{n\geq 0} \to^n = \to^0 \cup \to^+ && \text{(clausura reflexiva y transitiva)} \\ \leftarrow &= \{(a, b) \in A \times A \mid b \to a\} && \text{(inversa)} \\ \leftrightarrow &= \leftarrow \cup \to && \text{(clausura simétrica)} \\ \leftrightarrow^* &= (\leftrightarrow)^* && \text{(clausura de equivalencia)} \end{aligned}$$

Cuando $a \to b$, decimos que a se reduce a b en *un* paso de reducción. Intuitivamente, cuando $a \to^n b$, ocurre que a se reduce a b a través de $n - 1$ elementos intermedios mediante pasos unitarios de reducción: decimos entonces que a se reduce a b en n pasos de reducción. Por otro lado, cuando $a \to^* b$, a se reduce a b en cero o más pasos de reducción.

La clausura de equivalencia juega un papel muy importante en este trabajo. Intuitivamente, cuando $a \leftrightarrow^* b$, ambos elementos están conectados por cero o más pasos de reducción, cada uno de los cuales puede ser directo ($\to$) o inverso ($\leftarrow$).

Definición 7.5. Sean a y b tales que $a \leftrightarrow^* b$. Una secuencia $\langle a_0, \ldots, a_n \rangle$ es una *prueba* de la equivalencia de a y b (respecto de $\to$) si $a = a_0 \leftrightarrow a_1 \leftrightarrow \cdots \leftrightarrow a_n = b$.

Definición 7.6. Diremos que una prueba es un pico local si es de la forma $a \leftarrow c \rightarrow b$. Diremos que es un valle si es de la forma $a \rightarrow^* c \leftarrow^* b$, y lo notamos por $a \downarrow^* b$.

Nota. Por el contexto, se distinguirá la «prueba» obtenida con ACL2 de estas *pruebas*.

7.3. Reducciones polinómicas

Estudiaremos a continuación un tipo particular de reducción definida sobre conjuntos de polinomios, así como su formalización en la lógica de ACL2.

Definición 7.7. Sea f un polinomio no nulo. La relación de reducción $\rightarrow_f$ sobre polinomios inducida por f se define de manera que $p \rightarrow_f q$ si existe $m \in M[X]$ no nulo tal que:

1. $m \in p$
2. $\exists c \in M[X]\ m = -c \cdot mp(f)$.
3. $q = p + c \cdot f$.

Al monomio m se le llamará *monomio de reducción* y al monomio c, *factor de reducción*.

Definición 7.8. Sea $F = \{f_1, \ldots, f_k\}$ un conjunto finito de polinomios no nulos. La relación de reducción $\rightarrow_F$ inducida por F se define como sigue:

$$\rightarrow_F \ = \bigcup_{i=1}^{k} \rightarrow_{f_i}$$

Este proceso de reducción se puede ver como un paso en una división generalizada, la cual se establece como sigue. Sean p un polinomio y $F = \{f_1, \ldots, f_k\}$ un conjunto finito de polinomios, entonces existen $c_1, \ldots, c_n$ y r polinomios tales que

$$p = c_1 f_1 + \ldots + c_n f_n + r$$

donde $r = 0$ o r está completamente reducido con respecto a F.

Con el algoritmo de *división generalizada* se pueden calcular estos c_i y r (nótese que éstos no son únicos, si se reorganizan los f_i se obtienen unos c_i

y r diferentes). Este algoritmo es una forma generalizada del algoritmo de división tradicional de una variable.

Nótese, también, que con este algoritmo no se resuelve completamente el problema de la pertenencia al ideal. De hecho, si después de la división de p por F se obtiene $r = 0$ entonces $p = c_1 f_1 + \ldots + c_n f_n$ y, por tanto, pertenece al ideal generado por F. Así, $r = 0$ es condición suficiente para la pertenencia al ideal. Sin embargo, no es necesaria.

En este punto, surge la pregunta de si dado un ideal habría alguna base de ese ideal para la cual la condición de $r = 0$ es equivalente a la pertenencia al ideal. La respuesta es sí: las bases de Gröbner. Estas bases se estudiarán en el capítulo siguiente.

A continuación se presenta la formalización de esta reducción polinómica como una instancia concreta del marco suministrado por [87] para las relaciones de reducción abstractas en ACL2. La idea es que, posteriormente, esto nos permitirá traspasar mediante instanciación funcional propiedades bien conocidas de las reducciones abstractas al caso particular de las reducciones polinómicas, sin necesidad de demostrarlas partiendo de cero. El principal resultado que usaremos será el hecho de que una relación de reducción convergente tiene clausura de equivalencia decidible.

En [87] se formalizan las reducciones abstractas en la lógica de ACL2 como funciones binarias que a partir de un objeto de un cierto *dominio* y de un *operador* devuelven otro objeto del mismo dominio, llevando a cabo un *paso de reducción*. Un operador no puede ser aplicado a cualquier objeto, por lo que se introduce también un predicado binario que indica cuándo es *válida* dicha aplicación.

Por lo tanto, este enfoque requiere definir tres funciones: un predicado unario que especifica el dominio sobre el cual pretendemos que esté definida la reducción, un predicado binario que comprueba si es válido aplicar un operador a un objeto y, por último, una función binaria que permite aplicar un operador a un objeto. Y además, una representación concreta de los operadores.

En el caso que nos ocupa, el dominio de los objetos es el de los anillos de polinomios racionales de k variables, $Q[x_1, \ldots, x_k]$. Por tanto, el predicado unario que especifica el dominio es `polinomiop`. A continuación, procederemos a definir la noción concreta de reducción sobre polinomios siguiendo el esquema anterior.

Definamos a continuación qué entendemos por operador en nuestro caso.

Definición 7.9. Un operador es una tripleta $\langle m, c, f\rangle$ donde:

1. m es el monomio de reducción,
2. c es el factor de reducción y
3. f es el polinomio por el que se va a reducir.

Nótese que el valor concreto de c es deducible de m y de f, que es de lo que disponemos. Sin embargo, conviene no obviarlo para que el desarrollo posterior sea más claro. Esto hace también que no sea necesario calcular su valor cada vez que se realice un paso de reducción.

La función ACL2 que construye un operador recibe m, c y f y crea una lista que contiene los tres valores.

```
(defun operador (m c f)
  (list m c f)))
```

Las funciones accesoras permiten extraer cada uno de los valores que constituyen el operador.

```
(defmacro o-monomio (o)
  `(first ,o))

(defmacro o-factor (o)
  `(second ,o))

(defmacro o-polinomio (o)
  `(third ,o))
```

Definición 7.10. Decimos que es *válido* aplicar un operador, $\langle m, c, f\rangle$, a un polinomio p respecto de un conjunto finito de polinomios F, si m es un monomio que pertenece al polinomio p, f es un polinomio no nulo perteneciente a F, $mp(f)$ divide a m y $c = -m/mp(f)$.

Definamos ahora este concepto en el sistema. Como ACL2 no distingue entre mayúsculas y minúsculas, de aquí en adelante, utilizaremos `fi` para denotar al polinomio f y `F` para el conjunto de polinomios F en los casos en los que haga falta esta distinción.

```
(defun valido (p o F)
  (let ((m (o-monomio o))
        (c (o-factor o))
        (fi (o-polinomio o)))
    (and (polinomiop p)
         (monomiop m) (en-monomio m p)
         (polinomiop fi)
         (not (nulop fi)) (en fi F)
         (dividep (termino (primero fi)) (termino m))
         (equal (polinomio (RAC-MON::- (RAC-MON::/ m (primero fi))))
                c))))
```

Nótese que, en la formalización, el factor de reducción se interpreta como un polinomio (que consta de un único monomio). Esto nos va a permitir utilizar el producto de polinomios durante el cálculo de la reducción y evitará la necesidad de utilizar una operación distinta (el producto de monomio por polinomio) en el resto del trabajo.

Ejemplo 7.11. Si el polinomio $x+1$ pertenece al conjunto de polinomios F, entonces sería válido aplicar al polinomio x^2+xy el operador $\langle x^2, -x, x+1\rangle$ respecto de F.

En ACL2, se comprobaría la validez como sigue. Nótese que en este caso tratamos con monomios y polinomios de dos variables, $\{x, y\}$. Por tanto, el monomio x^2 es $1x^2y^0$, `(1 2 0)` en ACL2; de esta forma, el polinomio x^2+xy sería en ACL2, `((1 2 0) (1 1 1))`, y el polinomio $x+1$, `((1 1 0) (1 0 0))`. Consideremos también que se ha definido en ACL2 una variable `F` con al menos el polinomio $x+1$.

```
(valido '((1 2 0) (1 1 1))
        (operador '(1 2 0) '((-1 1 0)) '((1 1 0) (1 0 0)))
        F)
```

El polinomio que se obtiene tras un paso de reducción se calcula de la siguiente forma.

Definición 7.12. Sea $o = \langle m, c, f\rangle$ un operador. Entonces:

$$\textit{reducción}(p, o) = p + c \cdot f$$

En ACL2,

```
(defun reduccion (p o)
  (let* ((c (o-factor o))
         (f (o-polinomio o)))
   (+ p (* c f))))
```

Ejemplo 7.13. Siguiendo con el ejemplo anterior, el polinomio resultante de aplicar a $x^2 + xy$ el operador $\langle x^2, -x, x+1\rangle$ respecto de F sería $xy - x$. En ACL2, se calcula por

```
(reduccion '((1 2 0) (1 1 1))
           (operador '(1 2 0) '((-1 1 0)) '((1 1 0) (1 0 0))))
```

A continuación procederemos, en primer lugar, a construir la clausura de equivalencia, $\leftrightarrow_F^*$, y luego particularizaremos ésta para obtener $\rightarrow_F^*$, $\rightarrow_F$ y otras relaciones derivadas de $\rightarrow_F$ de uso común.

En principio, la función ACL2 que formaliza la relación $\leftrightarrow_F^*$ tiene tres parámetros, que representan a p, q (los polinomios relacionados) y a F, respectivamente. Siguiendo a [87] añadimos como un parámetro extra la prueba de la relación entre p y q a través de F y evita la aparición de un cuantificador existencial. Este nuevo parámetro tiene como cometido almacenar la secuencia de pasos de reducción que se realizan.

Formalizaremos en ACL2 el concepto de prueba como una lista de pasos de prueba (posiblemente vacía), cada uno de los cuales representa un paso de reducción.

Un *paso de prueba* será una estructura que constará de cuatro campos: los elementos que se conectan, el operador que se aplica y un marcador que indica si el paso es del primer elemento hacia el segundo (paso directo) o al contrario (paso inverso).

Cada paso de prueba se representa en ACL2 mediante una estructura [10] `r-step` [1] que consta de cuatro campos: `elt1` y `elt2` (los elementos que se conectan), `operator` (el operador que se aplica) y `direct` (un booleano que indica si el paso es del primer elemento hacia el segundo o al contrario). Diremos que un paso de prueba es *inverso* si su campo `direct` es `nil`, y *directo* en caso contrario.

Para que un paso de prueba sea válido, la aplicación (en el sentido indicado) del operador a uno de los elementos de la estructura debe producir como resultado el otro elemento. Pero esto no basta, además hay que comprobar

[1] Definida en `abstract-proofs.lisp` del directorio `relaciones`.

que realmente dicho operador es aplicable al elemento sobre el que se lleva a cabo la reducción.

```
(defun paso-valido (paso F)
  (let ((p (elt1 paso))
        (q (elt2 paso))
        (operador (operator paso))
        (directo (direct paso)))
    (and (r-step-p paso)
         (implies directo
                  (and (valido p operador F)
                       (equal (reduccion p operador) q)))
         (implies (not directo)
                  (and (valido q operador F)
                       (equal (reduccion q operador) p))))))
```

Esto es, un paso de prueba es un paso válido respecto de un conjunto finito de polinomios, `F`, si es válido aplicar a su primer elemento (`elt1`) su operador (`operator`) respecto de `F`, en caso de que sea un paso directo, o si es válido aplicar a su segundo elemento (`elt2`) su operador respecto de `F`, en caso contrario. Y además la aplicación de ese operador válido produce el otro elemento.

Ejemplo 7.14. La estructura compuesta por $x^2 + xy$ (como primer elemento), $xy - x$ (como segundo elemento), $\langle x^2, -x, x + 1\rangle$ (como operador) y el booleano t (indicando que el paso es directo) sería un paso de prueba válido.

Una vez que tenemos la función que comprueba la validez de un paso de prueba, podemos definir qué entendemos por equivalencia entre dos polinomios respecto a la reducción polinómica que hemos definido (es decir, $\leftrightarrow_F^*$).

```
(defun<k> <->* (p q prueba F)
  (let ((r (elt2 (first prueba))))
    (and (k-polinomiop p)
         (if (endp prueba)
             (equal p q)
           (and (<-> p r (first prueba) F)
                (<->* r q (rest prueba) F))))))
```

donde `<->` se define por.

```
(defmacro <-> (p q paso F)
  `(and (paso-valido ,paso ,F)
        (equal ,p (elt1 ,paso))
        (equal ,q (elt2 ,paso))))
```

Como se observa, el parámetro `prueba` es una lista que contiene la concatenación de pasos de prueba que atestiguan que $p \leftrightarrow_F^* q$.

Veamos a continuación cómo esta formalización de $\leftrightarrow_F^*$ nos permite definir en ACL2 algunos casos particulares de relaciones.

Así, por ejemplo, si tenemos que $p \rightarrow_f q$, la prueba estaría compuesta por un único *paso* de prueba.

`prueba` = `(list paso)`

Este paso de prueba sería una estructura `r-step` con `direct` igual a `t`, `elt1` igual a p, `elt2` igual a q y `operator` igual a `(operador` m c f`)`. Aquí, m sería el monomio de p por el que se realiza la reducción y c sería el factor de reducción correspondiente.

```
paso = (make-r-step
          :direct t
          :elt1 p
          :elt2 q
          :operator (operador m c f))
```

donde `make-r-step` es la función constructora la estructura `r-step`.

En el caso de tener $p \rightarrow_{f_1} r \leftarrow_{f_2} q$, la prueba estaría compuesta por dos pasos de prueba.

`prueba` = `(list` paso_1 paso_2`)`

Considerando que en el primer paso, m_1 es el monomio de p por el que se hace la reducción y c_1 el factor de reducción, y que en el segundo paso, m_2 es el monomio de q por el que se hace la reducción (ya que la reducción es de q a r, en vez de r a q) y c_2 el factor de reducción, se tiene que

```
paso1= (make-r-step
         :direct t
         :elt1 p
         :elt2 r
         :operator (operador m1 c1 f1))

paso2= (make-r-step
         :direct nil
         :elt1 r
         :elt2 q
         :operator (operador m2 c2 f2))
```

Nótese cómo en el segundo paso el campo `direct` está a `nil`, indicando que la reducción se produce de q a r.

Una vez construida la función `<->*` se definen `->*` y `->`. Para la primera de ellas basta exigir que la prueba sea *descendente*. Para la segunda, debe existir un único paso de prueba directo.

```
(defmacro ->* (p q prueba F)
  `(and (<->* ,p ,q ,prueba ,F)
        (descendentep ,prueba)))

(defmacro -> (p q prueba F)
  `(and (<->* ,p ,q ,prueba ,F)
        (direct (first ,prueba))
        (not (consp (rest ,prueba)))))
```

El predicado `descendentep` expresa el hecho de que una prueba está formada únicamente por pasos de prueba directos. Se define como una macro utilizando la definición de `steps-down` de la formalización de reducciones abstractas de en `abstract-proofs.lisp`.

```
(defmacro descendentep (prueba)
  `(steps-down ,prueba))

(defun steps-down (p)
   (if (endp p)
       t
     (and
      (direct (car p))
      (steps-down (cdr p)))))
```

También se pueden definir las clausuras positivas `<->+` y `->+` exigiendo que exista, al menos, un paso de prueba.

```
(defmacro <->+ (p q prueba F)
  `(and (<->* ,p ,q ,prueba ,F)
        (consp ,prueba)))

(defmacro ->+ (p q prueba F)
  `(and (->* ,p ,q ,prueba ,F)
        (consp ,prueba)))
```

Al igual que las pruebas descendentes, existen pruebas con formas especiales que son útiles en la formalización de propiedades de confluencia sobre las relaciones de reducción. Veamos el caso de las pruebas con forma de *pico local* y de *valle*, que aparecen de manera natural al definir la noción de confluencia local (en el capítulo siguiente).

Una prueba con forma de pico local se compone únicamente de dos pasos de prueba: uno inverso seguido de otro directo. Por ejemplo, si tenemos que $p \leftarrow_F r \rightarrow_F q$, existe una prueba de que $p \leftrightarrow_F^* q$ con forma de pico local. El predicado `<-x->` comprueba este tipo de relación.

```
(defmacro <-x-> (p q prueba F)
  `(and (<->* ,p ,q ,prueba ,F)
        (pico-localp ,prueba)))
```

El predicado `pico-localp` se define como una macro utilizando la definición de `local-peak-p` de la formalización de reducciones abstractas.

```
(defmacro pico-localp (prueba)
  `(local-peak-p ,prueba))

(defun local-peak-p (p)
   (and (consp p)
        (consp (cdr p))
        (atom (cddr p))
        (not (direct (car p)))
        (direct (cadr p))))
```

Una prueba con forma de valle se compone de una secuencia de pasos de prueba directos seguida de una secuencia de pasos de prueba inversos (ambas

secuencias pueden estar vacías). Si entre p y q existe una prueba en forma de valle, lo notamos por $p \downarrow_F^* q$. Por ejemplo, si tenemos que $p \to_F r \to_F s \leftarrow_F q$, existe una prueba de que $p \leftrightarrow_F^* q$ con forma de valle. El predicado `->*<-` implementa esta relación.

```
(defmacro ->*<- (p q prueba F)
  `(and (<->* ,p ,q ,prueba ,F)
        (vallep ,prueba)))
```

El predicado `vallep` se define como una macro utilizando la definición de `steps-valley` de la formalización de reducciones abstractas.

```
(defmacro vallep (prueba)
  `(steps-valley ,prueba))

(defun steps-valley (p)
   (cond ((endp p) t)
         ((direct (car p)) (steps-valley (cdr p)))
         (t (steps-up (cdr p)))))

(defun steps-up (p)
   (if (endp p)
       t
     (and
      (not (direct (car p)))
      (steps-up (cdr p)))))
```

7.4. Equivalencia y congruencia inducida

A continuación, se demuestra que la relación de equivalencia coincide con la congruencia inducida por el ideal (teorema 7.18).

Este resultado es importante ya que unifica dos visiones sobre una misma noción de equivalencia polinómica. De un lado, la visión proporcionada por la clausura de equivalencia de una relación de reducción construida a partir de la base de un ideal. Del otro, la visión dada por la pertenencia de la diferencia de polinomios al ideal generado.

La formalización de los resultados de esta sección se encuentra en el fichero `congruencia-ideal.lisp` del directorio `Buchberger`.

Los tres lemas siguientes son fundamentales para la demostración del teorema principal.

Lema 7.15. *Sean F un conjunto finito de polinomios, p un polinomio y m un monomio no nulo. Entonces:*

$$p \in F \implies m \cdot p \to_F 0$$

Demostración. La clave está en considerar que cualquier polinomio se puede reducir por su monomio líder o principal. Sea m_1 el monomio líder de p, entonces el polinomio $m \cdot p$ tendrá como monomio líder $m \cdot m_1$. Sea $c = -(m \cdot m_1)/m_1 = -m$. Por definición de la relación de reducción, $m \cdot p \to_p m \cdot p + (-m) \cdot p = 0$. Por tanto, como $p \in F$ se tiene que $m \cdot p \to_F 0$. □

El teorema se puede formalizar en ACL2 de la siguiente forma. Sean `m` un monomio no nulo, `p` un polinomio y `F` una lista de polinomios, y supongamos que `p` pertenece a `F`. Se ha de demostrar que `m * p` se reduce a 0 respecto a `F`. Para ello, hay que encontrar una prueba de dicha relación.

Como ya se comentó en la sección anterior, el parámetro extra del predicado `->` representa la prueba de la relación entre p y q como un paso de reducción.

```
(defthm |p en F => m * p ->F 0|
  (let ((prueba (prueba-|m * p ->p 0| m p)))
    (implies (and (k-monomiop m) (not (RAC-MON::nulop m))
                  (k-polinomiop p)
                  (k-polinomiosp F)
                  (en p F))
             (-> (* (polinomio m) p) (nulo) prueba F))))
```

Nótese que la prueba que justifica el paso de reducción se construye mediante la función `prueba-|m * p ->p 0|` que la obtiene a partir de m y p.

Por tanto, debemos en primer lugar definir esa función, que devuelve la secuencia de pasos de reducción que hay que realizar para reducir $m \cdot p$ a 0. Puede verse cómo es suficiente con un único paso de reducción. El monomio de reducción será el producto de m por el monomio principal de p, el factor de reducción será $-m$ y el polinomio de reducción, p.

```
(defun prueba-|m * p ->p 0| (m p)
  (if (nulop p)
```

```
  nil
(list (make-r-step
       :direct t
       :elt1 (* (polinomio m) p)
       :elt2 (nulo)
       :operator (operador (RAC-MON::* m (primero p))
                           (polinomio (RAC-MON::- m))
                           p)))))
```

Para la demostración del teorema anterior, basta probar que si p pertenece a F entonces el único paso de la prueba que devuelve la función anterior es realmente un paso válido de reducción respecto de F.

```
(defthm |p en F => paso-valido(first(prueba-m * p ->p 0))|
  (implies (and (k-monomiop m) (not (RAC-MON::nulop m))
                (k-polinomiop p) (not (nulop p))
                (en p F))
           (paso-valido (first (prueba-|m * p ->p 0| m p)) F)))
```

Lema 7.16. *Sean F un conjunto finito de polinomios y p, q y r polinomios. Entonces:*

$$p \to_F q \implies p + r \downarrow_F^* q + r$$

Demostración. Supongamos que $f \in F$ es el polinomio que se utiliza para reducir p a q, es decir, $p \to_f q$. Sea $m = c_m \cdot t_m$ el monomio de p sobre el que se aplica la reducción, donde c_m y t_m son, respectivamente, el coeficiente y el término de m. De acuerdo con la definición de la relación de reducción, $q = p - m/mp(f) \cdot f$.

Sea c_r el coeficiente del monomio de r cuyo término es t_m. Distinguimos tres casos según c_r y $c_m + c_r$ sean o no nulos.

Caso 1: $c_r = 0$.

El monomio con término t_m en $p + r$ es, precisamente, m. Por lo tanto, se puede emplear como monomio de reducción para reducir, en un paso, $p + r$

a $q + r$ mediante f, con lo que $p + r \downarrow_F^* q + r$. En efecto:

$$
\begin{array}{lll}
c_r = 0 & & \\
\Longrightarrow\ p + r \rightarrow_f p + r - \dfrac{m}{mp(f)} \cdot f & & \text{(def. de } \rightarrow\text{)} \\
\Longrightarrow\ p + r \rightarrow_f q + r & & (q = p - \dfrac{m}{mp(f)} \cdot f) \\
\Longrightarrow\ p + r \rightarrow_F q + r & & \text{(ya que } f \in F\text{)} \\
\Longrightarrow\ p + r \downarrow_F^* q + r & & \text{(def. de } \downarrow_F^*\text{)}
\end{array}
$$

Caso 2: $c_r \neq 0 \wedge c_m + c_r = 0$.

El monomio con término t_m en $q + r$ es $c_r \cdot t_m$, ya que el coeficiente de t_m en q es 0. Por lo tanto, se puede emplear como monomio de reducción para reducir, en un paso, $q + r$ a $p + r$ mediante f, con lo que $p + r \downarrow_F^* q + r$. En efecto:

$$
\begin{array}{lll}
c_r \neq 0 \wedge c_m + c_r = 0 & & \\
\Longrightarrow\ q + r \rightarrow_f q + r - \dfrac{c_r \cdot t_m}{mp(f)} \cdot f & & \text{(def. de } \rightarrow\text{)} \\
\Longrightarrow\ q + r \rightarrow_f p - \dfrac{c_m \cdot t_m}{mp(f)} \cdot f + r - \dfrac{c_r \cdot t_m}{mp(f)} \cdot f & & (q = p - \dfrac{c_m \cdot t_m}{mp(f)} \cdot f) \\
\Longrightarrow\ q + r \rightarrow_f p + r & & \text{(ya que } c_m + c_r = 0\text{)} \\
\Longrightarrow\ q + r \rightarrow_F p + r & & \text{(ya que } f \in F\text{)} \\
\Longrightarrow\ q + r \downarrow_F^* p + r & & \text{(def. de } \downarrow_F^*\text{)}
\end{array}
$$

Caso 3: $c_r \neq 0 \wedge c_m + c_r \neq 0$.

El monomio con término t_m en $p + r$ es $(c_m + c_r) \cdot t_m$, ya que $c_m + c_r \neq 0$. Por lo tanto, se puede emplear como monomio de reducción para reducir, en un paso, $p + r$ a $p + r - [(c_m + c_r) \cdot t_m / mp(f)] \cdot f$ mediante f. En efecto:

$$
\begin{array}{lll}
c_r \neq 0 \wedge c_m + c_r \neq 0 & & \\
\Longrightarrow\ p + r \rightarrow_f p + r - \dfrac{(c_m + c_r) \cdot t_m}{mp(f)} \cdot f & & \text{(def. de } \rightarrow\text{)} \\
\Longrightarrow\ p + r \rightarrow_F p + r - \dfrac{(c_m + c_r) \cdot t_m}{mp(f)} \cdot f & & \text{(ya que } f \in F\text{)}
\end{array}
$$

El monomio con término t_m en $q + r$ es $c_r \cdot t_m$, ya que el coeficiente de t_m en q es 0. Por lo tanto, se puede emplear como monomio de reducción para

reducir, en un paso, $q + r$ a $p + r - [(c_m + c_r) \cdot t_m / mp(f)] \cdot f$ mediante f. En efecto:

$$
\begin{aligned}
& c_r \neq 0 \wedge c_m + c_r \neq 0 \\
& \quad \Longrightarrow\ q + r \to_f q + r - \frac{c_r \cdot t_m}{mp(f)} \cdot f && \text{(def. de } \to\text{)} \\
& \quad \Longrightarrow\ q + r \to_f p - \frac{c_m \cdot t_m}{mp(f)} \cdot f + r - \frac{c_r \cdot t_m}{mp(f)} \cdot f && \left(q = p - \frac{c_m \cdot t_m}{mp(f)} \cdot f\right) \\
& \quad \Longrightarrow\ q + r \to_f p + r - \frac{(c_m + c_r) \cdot t_m}{mp(f)} \cdot f && \text{(arit. polinómica)} \\
& \quad \Longrightarrow\ q + r \to_F p + r - \frac{(c_m + c_r) \cdot t_m}{mp(f)} \cdot f && \text{(ya que } f \in F\text{)}
\end{aligned}
$$

Uniendo ambos resultados, se obtiene que $p + r \downarrow_F^* q + r$.

□

El teorema puede expresarse en ACL2 de la siguiente forma.

```
(defthm |p ->F q => p + r ->*<-F q + r|
  (let ((valle (prueba-|p ->F q => p + r ->*<-F q + r|
                p q prueba r)))
    (implies (and (k-polinomiop p)
                  (k-polinomiop q)
                  (k-polinomiop r)
                  (k-polinomiosp F)
                  (-> p q prueba F))
             (->*<- (+ p r) (+ q r) valle F))))
```

Describamos a continuación los pasos fundamentales de la demostración ACL2.

Nuevamente, al igual que ocurría en el lema anterior, debemos en primer lugar definir una función que construya la secuencia de pasos de reducción que justifica que $p + r \downarrow_F^* q + r$. Esta secuencia se halla a partir de la prueba de la reducción de p a q siguiendo la idea proporcionada por la demostración anterior. Nótese que se están construyendo pruebas a partir de otras.

```
(defun prueba-|p ->F q => p + r ->*<-F q + r| (p q prueba r)
  (let* ((o (operator (first prueba)))
```

```
        (m (o-monomio o))
        (mr (en-termino (termino m) r))
        (crm (RAC::+ (coeficiente m) (coeficiente mr))))
   (cond ((equal p q)
          nil)
         ((equal mr nil)
          (prueba-caso-1 p q prueba r))
         (t
          (if (RAC::= crm (RAC::nulo))
              (prueba-caso-2 p q prueba r)
            (prueba-caso-3 p q prueba r))))))
```

Recordemos que la función `en-termino` recibe un término y un polinomio, y devuelve, si existe, el monomio asociado con ese término. Si no existe devuelve falso.

En consonancia con la demostración anterior, la función que construye la prueba buscada se divide en tres casos:

Caso 1: No existe ningún monomio con término t_m en r, es decir, $c_r = 0$.

En este caso, la prueba que conecta p con q es una secuencia que, por hipótesis, consta de un único paso de reducción. Sólo habría que cambiar en dicho paso los dos elementos que se conectan, dejando inalterado el operador y la dirección. Esto es lo que hace la siguiente función.

```
(defun prueba-caso-1 (p q prueba r)
  (list (make-r-step
         :direct (direct (first prueba))
         :elt1 (+ p r)
         :elt2 (+ q r)
         :operator (operator (first prueba)))))
```

A continuación, se demuestra que si $p \rightarrow_F q$ entonces la prueba que calcula la función anterior es valle y conecta $p + r$ con $q + r$.

```
(defthm |cr = 0 & p ->F q => p + r ->*<- q + r|
  (let ((m (o-monomio (operator (first prueba)))))
    (implies (and (-> p q prueba F)
                  (k-polinomiop r)
                  (not (en-termino (termino m) r)))
             (->*<- (+ p r) (+ q r) (prueba-caso-1 p q prueba r)
                    F))))
```

Caso 2: Existe un monomio con término t_m en r y se anula con el monomio de reducción, es decir, $c_r \neq 0$ y $c_m + c_r = 0$.

En este caso, el factor de reducción es $-c_r \cdot t_m / mp(f)$, ya que al anularse el monomio de q con término t_m, el coeficiente del monomio con término t_m en $q + r$ es c_r.

La función `prueba-caso-2` devuelve la prueba asociada, partiendo de la prueba que conecta p con q, que consiste en un único paso de reducción. El paso se modifica de la siguiente forma:

- Los dos elementos conectados p y q deberán ser ahora $p + r$ y $q + r$.
- La dirección será `nil` indicando que el paso es inverso.
- El operador tendrá como monomio de reducción el monomio de r con término t_m (llamémosle m_r, es decir, $m_r = c_r \cdot t_m$), como factor de reducción $-m_r / mp(f)$ (calculado por la función `f-reduccion`) y como polinomio de reducción, f.

```
(defun f-reduccion (mr mfi)
  (polinomio (RAC-MON::- (RAC-MON::/ mr mfi))))

(defun prueba-caso-2 (p q prueba r)
  (let* ((o (operator (first prueba)))
         (m (o-monomio o))
         (mr (en-termino (termino m) r))
         (fi (o-polinomio o)))
    (list (make-r-step
           :direct nil
           :elt1 (+ p r)
           :elt2 (+ q r)
           :operator
           (operador mr (f-reduccion mr (primero fi)) fi)))))
```

El teorema siguiente establece que si $p \rightarrow_F q$ entonces la prueba que calcula la función anterior es valle y conecta $p + r$ con $q + r$.

```
(defthm |cr + cm = 0 & p ->F q => p + r ->*<- q + r|
  (let* ((m (o-monomio (operator (first prueba))))
         (mr (en-termino (termino m) r))
         (crm (RAC::+ (coeficiente m) (coeficiente mr))))
    (implies (and (-> p q prueba F) (not (equal p q))
                  (k-polinomiop r)
                  (k-polinomiosp F)
```

```
                mr
                (equal crm (RAC::nulo)))
          (->*<- (+ p r) (+ q r) (prueba-caso-2 p q prueba r)
                 F))))
```

Para su demostración, es necesario comprobar que m_r está en la suma de q y r, dado que $p \rightarrow_F q$ y que $c_r \neq 0$:

```
(defthm |p ->F q => mr en (q + r)|
  (let* ((m (o-monomio (operator (first prueba))))
         (mr (en-termino (termino m) r)))
    (implies (and (-> p q prueba F) (not (equal p q))
                  (polinomiop r)
                  (k-polinomiosp F)
                  mr)
             (en-monomio mr (+ q r)))))
```

Caso 3: Existe un monomio con término t_m en r y no se anula con el monomio de reducción: $c_m + c_r \neq 0$ y $c_r \neq 0$. En este caso, hay dos factores de reducción. Uno es $-(c_m + c_r) \cdot t_m/mp(f)$ (para $p + r$) y el otro es $-c_r \cdot t_m/mp(f)$ (para $q + r$).

- $p + r \rightarrow_F p + r - [(c_m + c_r) \cdot t_m/mp(f)] \cdot f$. Nótese que el coeficiente de t_m en $p + r$ es $c_m + c_r$.
- $q + r \rightarrow_F p + r - [(c_m + c_r) \cdot t_m/mp(f)] \cdot f$. Nótese que el coeficiente de t_m en $q + r$ es c_r.

Por lo tanto, en este caso son necesarios dos pasos de prueba, uno directo y otro inverso. La función `paso-1-caso-3` calcula el paso de prueba directo.

```
(defun paso-1-caso-3 (p paso r)
  (let* ((o (operator paso))
         (m (o-monomio o))
         (mr (en-termino (termino m) r))
         (mrm (RAC-MON::+ m mr))
         (fi (o-polinomio o)))
    (make-r-step
     :direct t
     :elt1 (+ p r)
     :elt2 (+ r (+ p (* (f-reduccion mrm (primero fi)) fi)))
     :operator
     (operador mrm (f-reduccion mrm (primero fi)) fi))))
```

Es necesario demostrar que lo que devuelve esta función es un paso de prueba válido y directo.

```
(defthm |p ->F q => paso-valido(paso-1-caso-3)|
  (let* ((o (operator (first prueba)))
         (m (o-monomio o))
         (mr (en-termino (termino m) r))
         (crm (RAC::+ (coeficiente m) (coeficiente mr)))
         (paso (paso-1-caso-3 p (first prueba) r)))
  (implies (and (-> p q prueba F)
                (k-polinomiop r)
                (not (equal p q))
                (en-termino (termino m) r)
                (not (equal crm (RAC::nulo))))
           (and (paso-valido paso F)
                (direct paso)))))
```

Para esto, también se necesita saber que $(c_m + c_r) \cdot t_m$ está en $p + r$.

```
(defthm |(cm + cr)tm en p + r|
  (let* ((m (o-monomio (operator (first prueba))))
         (mr (en-termino (termino m) r))
         (crm (RAC::+ (coeficiente m) (coeficiente mr))))
    (implies (and (-> p q prueba F) (not (equal p q))
                  (polinomiop r)
                  (en-termino (termino m) r)
                  (not (equal crm (RAC::nulo))))
             (en-monomio (RAC-MON::+ m mr) (+ p r)))))
```

La siguiente función `paso-2-caso-3` calcula el paso de prueba inverso.

```
(defun paso-2-caso-3 (p q paso r)
  (let* ((o (operator paso))
         (m (o-monomio o))
         (mr (en-termino (termino m) r))
         (mrm (RAC-MON::+ m mr))
         (fi (o-polinomio o)))
    (make-r-step
     :direct nil
     :elt1 (+ r (+ p (* (f-reduccion mrm (primero fi)) fi)))
     :elt2 (+ q r)
     :operator
     (operador mr (f-reduccion mr (primero fi)) fi))))
```

En este caso, también es necesario demostrar que lo que devuelve esta función es un paso de prueba válido.

```
(defthm |p ->F q => paso-valido(paso-2-caso-3)|
  (let* ((o (operator (first prueba)))
         (m (o-monomio o))
         (mr (en-termino (termino m) r))
         (crm (RAC::+ (coeficiente m) (coeficiente mr)))
         (paso (paso-2-caso-3 p q (first prueba) r)))
    (implies (and (-> p q prueba F)
                  (k-polinomiop p)
                  (k-polinomiop q)
                  (k-polinomiop r)
                  (k-polinomiosp F)
                  (not (equal p q))
                  (en-termino (termino m) r)
                  (not (equal crm (RAC::nulo))))
             (paso-valido paso F))))
```

Seguidamente, antes de poder unir los dos pasos de reducción previamente obtenidos, necesitamos comprobar que el primer elemento del paso inverso coincide con el segundo elemento del paso directo.

```
(defthm |elt1(paso-2-caso-3) = elt2(paso-1-caso-3)|
  (equal (elt1 (paso-2-caso-3 p q paso r))
         (elt2 (paso-1-caso-3 p paso r))))
```

Por último, se construye la prueba del último caso.

```
(defun prueba-caso-3 (p q prueba r)
  (list (paso-1-caso-3 p (first prueba) r)
        (paso-2-caso-3 p q (first prueba) r)))
```

Finalmente, se establece que si $p \rightarrow_F q$ entonces la prueba que calcula la función anterior es valle y conecta $p + r$ con $q + r$.

```
(defthm |cr + cm != 0 & p ->F q => p + r ->*<- q + r|
  (let* ((m (o-monomio (operator (first prueba))))
         (mr (en-termino (termino m) r))
         (crm (RAC::+ (coeficiente m) (coeficiente mr))))
    (implies (and (-> p q prueba F) (not (equal p q))
                  (k-polinomiop r)
                  (k-polinomiosp F)
```

```
        (en-termino (termino m) r)
        (not (equal crm (RAC::nulo))))
  (->*<- (+ p r) (+ q r) (prueba-caso-3 p q prueba r)
         F)))
```

Así, se ha demostrado la existencia de una prueba valle en los tres posibles casos. La consideración de estos tres casos lleva a la demostración del teorema principal.

Veamos a continuación como si $p \leftrightarrow_F q$ con un único paso de prueba entonces $p \equiv_{\langle F \rangle} q$.

Lema 7.17. *Si F es un conjunto finito de polinomios e $I = \langle F \rangle$ entonces*

$$p \leftrightarrow_F q \implies p \equiv_I q$$

Demostración.

Caso 1: $p \to_F q$. Sea f el polinomio de F por el cual se hace la reducción, es decir, $p \to_f q$, y sea m el monomio de p que se utiliza en la reducción. Entonces

$$\begin{aligned}
p \to_f q &\implies q = p - (m/mp(f)) \cdot f && \text{(def. de } \to_F\text{)} \\
&\implies p - q = (m/mp(f)) \cdot f && \text{(prop. polinómicas)} \\
&\implies p - q \in \langle F \rangle && \text{(def. de } \langle F \rangle\text{)} \\
&\implies p \equiv_I q && \text{(def. de } \equiv_I\text{)}
\end{aligned}$$

Caso 2: $q \to_F p$. Análogamente, tomando el signo opuesto.

□

El teorema ACL2 correspondiente es el siguiente.

```
(defthm |p <->F q => p =<F> q|
  (implies (and (k-polinomiop p)
                (k-polinomiop q)
                (k-polinomiosp F)
                (<-> p q paso F))
           (=<> p q F)))
```

Para demostrarlo en ACL2 se debe suministrar un testigo de la pertenencia de $p - q$ al ideal generado por F (es decir, una secuencia de coeficientes).

La siguiente función `C-|p <->F q => p =<F> q|` es precisamente el testigo de la pertenencia de $p - q$ al ideal generado por F.

```
(defun C-|p <->F q => p =<F> q| (paso F)
  (let* ((o (operator paso))
         (fi (o-polinomio o))
         (ci (o-factor o)))
    (cond ((endp F) nil)
          ((equal fi (first F))
           (if (direct paso)
               (cons (- ci) (C0 (rest F)))
             (cons ci (C0 (rest F)))))
          (t
           (cons (nulo) (C-|p <->F q => p =<F> q|
                        paso (rest F)))))))
```

Su corrección viene dada por el siguiente teorema.

```
(defthm |p - q = cl(C-p <->F q => p =<F> q, F)|
  (implies (and (k-polinomiosp F)
                (<-> p q paso F))
           (equal (+ p (- q))
                  (combinacion-lineal
                    (|C-p <->F q => p =<F> q| paso F) F)))
```

Los tres lemas anteriores, 7.15, 7.16 y 7.17, son fundamentales para demostrar el siguiente teorema, que establece que la congruencia del ideal coincide con la relación de equivalencia. Las funciones definidas anteriormente para la construcción de las pruebas se emplearán también ahora. En particular, la función `prueba-|p ->F q => p + r ->*<-F q + r|` jugará un papel muy importante.

Teorema 7.18. *Si F es un conjunto finito de polinomios e $I = \langle F \rangle$ entonces*

$$p \equiv_I q \iff p \leftrightarrow_F^* q$$

Demostración. $\Rightarrow$: Sea $F = \{f_1, \ldots, f_n\}$

$$p \equiv_I q \implies p - q \in I \qquad \text{(def. de } \equiv_I\text{)}$$

$$\implies \exists c_1, \ldots, c_n \; p - q = \sum_{i=1}^{n} c_i f_i \qquad \text{(def. de pertenencia a ideal)}$$

Sea $c_i = m_{i1} + \cdots + m_{il_i}$. Al descomponer cada polinomio c_i en la suma de sus monomios y aplicar distributividad, se obtiene lo siguiente:

$$p = q + \sum_{i=1}^{n} \sum_{j=1}^{l_i} m_{ij} f_i$$

Demostremos ahora $p \leftrightarrow_F^* q$ por inducción sobre el número total de monomios $l = \sum_{i=1}^{n} l_i$.

Caso base:

Si $l = 0$ entonces $p = q$ y, por lo tanto, $p \leftrightarrow_F^* q$.

Paso de inducción:

Si $l > 0$, definimos

$$r = q + \sum_{i=1}^{n} \sum_{j=1}^{l_i} m_{ij} f_i - m_{11} f_1$$

Como evidentemente $r \equiv_I q$, aplicamos hipótesis de inducción para deducir que $r \leftrightarrow_F^* q$.

Por el lema 7.15 se tiene que $m_{11} f_1 \rightarrow_F 0$ y, por el lema 7.16, que $p = r + m_{11} f_1 \downarrow_F^* r$. Esto unido a que $r \leftrightarrow_F^* q$ implica $p \leftrightarrow_F^* q$.

$\Leftarrow$: Por inducción sobre la definición de $\leftrightarrow_F^*$.

Caso base: $p = q$

$$\begin{aligned} p = q &\implies p - q = 0 && (p - p = 0) \\ &\implies p - q \in \langle F \rangle && (0 \in \langle F \rangle) \\ &\implies p \equiv_I q && (\text{def. de } \equiv_I) \end{aligned}$$

Paso de inducción: $p \neq q$

Entonces existe un r tal que $p \leftrightarrow_F r \leftrightarrow_F^* q$ y por hipótesis de inducción,

$$r \leftrightarrow_F^* q \implies r \equiv_I q$$

Por el lema 7.17 $p \equiv_I r$ y por hipótesis de inducción $r \equiv_I q$, entonces $p \equiv_I q$ por definición de $\equiv_I$.

□

El teorema se formaliza en ACL2 mediante los dos siguientes eventos. Cada uno de ellos establece una de las implicaciones.

```
(defthm |p =<F> q => p <->F* q|
  (let ((prueba (prueba-|p =<F> q => p <->F* q| p q F)))
    (implies (and (k-polinomiop p)
                  (k-polinomiop q)
                  (k-polinomiosp F)
                  (=<> p q F))
             (<->* p q prueba F)))

(defthm |p <->F* q => p =<F> q|
  (implies (and (k-polinomiop p)
                (k-polinomiop q)
                (k-polinomiosp F)
                (<->* p q prueba F))
           (=<> p q F)))
```

Demostración del teorema `|p =<F> q => p <->F* q|`:

Siguiendo la demostración a mano, comenzamos por el primer teorema ACL2.

Sean `p` y `q` polinomios y `F` una lista de polinomios. Supuesto que `p` y `q` son congruentes respecto al ideal de `F`, se ha de demostrar que `p` y `q` están relacionados respecto a `F`. Para ello, hay que encontrar una prueba de dicha relación.

La función `prueba-|p =<F> q => p <->F* q|` construye una prueba que demuestra que `p` y `q` están relacionados mediante `<->F*`.

```
(defun<k> prueba-|p =<F> q => p <->F* q| (p q F)
  (let* ((C (en-ideal-witness (+ p (- q)) F))
         (pe (pares-especiales C F)))
    (prueba-|pep(pe, F) => cle(q, pe) <->F* q| q pe)))
```

Recordemos que la función `en-ideal-witness`, que aparece en el encapsulado correspondiente a la función de Skolem `en-ideal`, está restringida por axiomas a devolver la lista de coeficientes que atestiguan la pertenencia de un polinomio a un ideal.

Por consiguiente, la expresión `(en-ideal-witness (+ p (- q)) F)` es una lista `C` que representa a los c_i del esquema de demostración anterior, y puede ser utilizada para construir lo que denominaremos *pares especiales* de `C` y `F`. Estos pares especiales no son más que los pares (m_{ij}, f_i) del sumatorio $\sum_{i=1}^{n} \sum_{j=1}^{l_i} m_{ij} f_i$, que se obtienen a partir de los c_i y los f_i.

En primer lugar, hay que descomponer los c_i en sumas de monomios, para así poder expresar p como la suma de q con una combinación lineal de los f_i con los m_{ij}. De esto se encarga la función `pares-especiales`.

La función `pares-especiales` recibe la lista de coeficientes polinómicos `C` y la lista de polinomios `F`. Ambas listas pueden considerarse como una representación abstracta del sumatorio $\sum_{i=1}^{n} c_i f_i$, de manera que lo que se devuelve es la lista de pares (m_{ij}, f_i) que se obtiene al descomponer los c_i en sus monomios y aplicar reiteradamente la propiedad distributiva dentro del sumatorio.

```
(defun pares-especiales (C F)
  (if (or (endp C) (endp F))
      nil
    (append (pares-especiales-aux (first C) (first F))
            (pares-especiales (rest C) (rest F)))))
```

Esta función se apoya en la función auxiliar `pares-especiales-aux` que concatena recursivamente los pares (m_{ij}, f_i) correspondientes a cada par (c_i, f_i).

```
(defun pares-especiales-aux (c f)
  (if (nulop c)
      nil
    (cons (cons (primero c) (list f))
          (pares-especiales-aux (resto c) f))))
```

El problema se reduce ahora a construir una prueba que utilice q y los pares especiales en lugar de p, q y F. Más adelante, veremos que si $p - q \in \langle F \rangle$ entonces se cumple que $p = q + \sum_{i=1}^{n} \sum_{j=1}^{l_i} m_{ij} f_i$.

```
(defun prueba-|pep(pe, F) => cle(q, pe) <->F* q| (q pe)
 (if (endp pe)
     nil
  (append (prueba-|cle(q, pe) <->*F cle(q, rest(pe))| q pe)
          (prueba-|pep(pe, F) => cle(q, pe) <->F* q| q (rest pe)))))
```

Como se puede observar, esta función recursiva realiza su cometido concatenando dos pruebas en el caso no trivial. La primera de ellas relaciona $p = q + \sum_{i=1}^{n} \sum_{j=1}^{l_i} m_{ij} f_i$ y $r = p - m_{11} f_1$. La segunda prueba relaciona r y q, y se obtiene por recursión.

Por último, veamos cómo se construye la prueba que relaciona p con r. La idea consiste en utilizar las funciones que se emplearon para demostrar los lemas 7.15 y 7.16. La función del lema 7.15 se utiliza para construir la prueba de la reducción de $m_{11} f_1$ a 0. Esta prueba se emplea a su vez junto a la función del lema 7.16 para crear una prueba de que $p = r + m_{11} f_1$ y r están relacionados.

```
(defun prueba-|cle(q, pe) <->*F cle(q, rest(pe))| (q pe)
  (if (endp pe)
      nil
    (let ((par (first pe)))
      (prueba-|p ->F q => p + r ->*<-F q + r|
              (* (polinomio (first par)) (second par))
              (nulo)
              (prueba-|m * p ->p 0| (first par) (second par))
              (combinacion-lineal-especial q (rest pe))))))
```

Para poder hacer esto, tenemos que definir el concepto de combinación lineal especial, que es el resultado de sumar un polinomio `q` y los productos de una lista `pe` de pares especiales. Esta función juega el papel en la demostración de calcular $q + \sum_{i=1}^{n} \sum_{j=1}^{l_i} m_{il_j} f_i$ a partir de q y de la lista de pares (m_{il_j}, f_i).

```
(defun combinacion-lineal-especial (q pe)
  (if (endp pe)
      q
    (let ((par (first pe)))
      (+ (* (polinomio (first par)) (second par))
       (combinacion-lineal-especial q (rest pe))))))
```

Ahora, hemos de demostrar que realmente la prueba devuelta por la función `prueba-|cle(q, pe) <->*F cle(q, rest(pe))|` se construye correctamente. Para ello hay que comprobar que cuando `pe` contiene pares especiales de `F`, su sumatorio asociado está relacionado con el que se obtiene al eliminar el primer par. Es decir:

$$q + \sum_{i=1}^{n} \sum_{j=1}^{l_i} m_{ij} f_i \longleftrightarrow_F^* q + \sum_{i=1}^{n} \sum_{j=1}^{l_i} m_{ij} f_i - m_{11} f_1$$

El teorema que obtenemos, y que se presenta a continuación, se demuestra con ayuda de los lemas 7.15 y 7.16, y es clave en la demostración del paso inductivo del resultado final.

```
(defthm |cle(q, pe) <->*F cle(q, rest(pe))|
  (let ((prueba (prueba-|cle(q, pe) <->*F cle(q, rest(pe))| q pe)))
    (implies (and (k-polinomiop q)
                  (k-polinomiosp F)
                  (pares-especialesp pe F))
             (<->* (combinacion-lineal-especial q pe)
                   (combinacion-lineal-especial q (rest pe))
                   prueba
                   F))))
```

Como se observa, es necesario definir un reconocedor para los pares especiales, al que hemos llamado `pares-especialesp`. Esta función recibe dos parámetros, `pe` y `F`, y determina si `pe` es una lista de pares especiales de `F`.

```
(defun<k> pares-especialesp (pe F)
  (if (endp pe)
      (equal pe nil)
    (let ((par (first pe)))
      (and (consp par)
           (equal (len par) 2)
           (k-monomiop (first par))
           (not (RAC-MON::nulop (first par)))
           (k-polinomiop (second par))
           (en (second par) F)
           (pares-especialesp (rest pe) F)))))
```

El siguiente teorema establece que si $p - q = \sum_{i=1}^{n} c_i f_i$ entonces $p = q + \sum_{i=1}^{n} \sum_{j=1}^{l_i} m_{il_j} f_i$. La demostración ACL2 de este teorema es complicada y requiere algunos lemas además de muchas de las propiedades de anillo.

```
(defthm |p - q = cl(C, F) => cle(q, pe(C, F)) = p|
  (let ((pe (pares-especiales C F)))
    (implies (and (equal (+ p (- q)) (combinacion-lineal C F))
                  (k-polinomiop p)
                  (k-polinomiop q)
                  (k-polinomiosp C))
             (equal (combinacion-lineal-especial q pe) p))))
```

En este punto, podemos ya demostrar que si p es congruente con q módulo $\langle F \rangle$, entonces p tiene que ser igual a la combinación lineal especial formada por q y por los pares especiales que se obtienen de los testigos de la pertenencia de $p - q$ a $\langle F \rangle$.

```
(defthm |p =<F> q => cle(q, pe(en-ideal-witness(p - q, F))) = p|
  (let* ((C (en-ideal-witness (+ p (- q)) F))
         (pe (pares-especiales C F)))
    (implies (and (k-polinomiop p)
                  (k-polinomiop q)
                  (=<> p q F))
             (equal (combinacion-lineal-especial q pe) p))))
```

Finalmente, obtenemos el teorema siguiente que demuestra que si `pe` es una lista de pares especiales de `F` entonces la combinación lineal especial de `q` y `pe` está relacionada con `q` a través de una prueba. Su demostración se realiza por inducción sobre la estructura de la función que obtiene la prueba.

```
(defthm |pep(pe, F) => cle(q, pe) <->F* q|
  (let ((prueba (prueba-|pep(pe, F) => cle(q, pe) <->F* q| q pe)))
    (implies (and (k-polinomiop q)
                  (k-polinomiosp F)
                  (pares-especialesp pe F))
             (<->* (combinacion-lineal-especial q pe) q prueba F))))
```

Componiendo los dos últimos teoremas junto con algunos lemas de clausura se obtiene el teorema `|p =<F> q => p <->F* q|`.

Demostración del teorema `|p <->F* q => p =<F> q|`:

Procedamos ahora a demostrar `|p <->F* q => p =<F> q|`. Este teorema dice que si $p \leftrightarrow_F^* q$ entonces $p \equiv_I q$, o lo que es equivalente, que $p - q$ pertenece al ideal generado por F.

Para demostrar esto en ACL2 se debe suministrar un testigo de la pertenencia de $p - q$ al ideal generado por F.

La siguiente función `C-|p <->F* q => p =<F> q|` es precisamente ese testigo, y usa la función introducida en el lema 7.17 para obtener la combinación lineal a partir de un paso individual.

```
(defun C-|p <->F* q => p =<F> q| (prueba F)
  (if (endp prueba)
```

```
      nil
    (C+ (C-|p <->F q => p =<F> q| (first prueba) F)
        (C-|p <->F* q => p =<F> q| (rest prueba) F))))
```

Su corrección viene dada por el siguiente teorema.

```
(defthm |p - q = cl(C-p <->F* q => p =<F> q, F)|
  (implies (and (k-polinomiop p)
                (k-polinomiop q)
                (k-polinomiosp F)
                (<->* p q prueba F))
           (equal (+ p (- q))
                  (combinacion-lineal
                    (|C-p <->F* q => p =<F> q| prueba F) F))))
```

Este teorema implica trivialmente el teorema `|p <->F* q => p =<F> q|`.

La importancia de este último resultado radica en que se unifican dos visiones sobre una misma noción de equivalencia polinómica. De un lado, la visión proporcionada por la clausura de equivalencia de una relación de reducción construida a partir de la base de un ideal. Del otro, la visión dada por la pertenencia de la diferencia de polinomios al ideal generado.

La formalización en ACL2 conduce en ambos casos, al empleo de testigos. En el caso de la relación de equivalencia, el testigo, que exponemos siempre explícitamente, constituye un objeto de prueba que demuestra la conexión existente entre los polinomios relacionados. En cuanto a la congruencia inducida, el testigo, que mantenemos oculto bajo una función de Skolem apropiada, muestra los coeficientes polinómicos implicados en la pertenencia al ideal del polinomio diferencia.

Como hemos demostrado, ambos testigos están relacionados y las funciones diseñadas para la elaboración de la prueba final nos permiten construir uno cualquiera de ellos a partir del otro.

7.5. Noetherianidad de las reducciones polinómicas

Definición 7.19. Sea $\rightarrow$ una relación de reducción sobre un conjunto A. Decimos que $\rightarrow$ es *noetheriana* si no existe una secuencia infinita $\{a_i\}_{i\in\mathbb{N}}$ tal que $a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow \cdots$.

En esta sección se muestra la formalización de la noetherianidad de la relación de reducción polinómica realizada en `noetherianidad.lisp`. Para la formalización de esta propiedad en ACL2, nos basaremos en el siguiente resultado.

Teorema 7.20. *Una reducción es noetheriana si, y sólo si, está contenida en la inversa de una relación bien fundamentada.*

Por lo tanto, basta probar que existe una relación bien fundamentada tal que cuando se aplica cualquier paso válido de reducción a un elemento en el dominio de definición, se obtiene otro elemento menor respecto de tal relación.

Recordemos (capítulo 4) que empleamos $<$ para referirnos al orden bien fundamentado sobre los polinomios, y que dados dos polinomios p y q, $p < q$ si se cumple alguna de las siguientes condiciones:

1. $p = 0 \wedge q \neq 0$.
2. $p \neq 0 \wedge q \neq 0 \wedge tp(p) <_T tp(q)$.
3. $p \neq 0 \wedge q \neq 0 \wedge tp(p) = tp(q) \wedge resto(p) < resto(q)$.

Ya que este orden está bien fundamentado, se puede utilizar para establecer la noetherianidad de la reducción polinómica previamente presentada. Simplemente hay que expresar que mediante la aplicación válida de un operador a un polinomio se obtiene otro menor que él con respecto a $<$.

Teorema 7.21. *Sean p un polinomio, F un conjunto finito de polinomios y o un operador. Entonces,*

$$válido(p, o, F) \implies reducción(p, o) < p$$

En ACL2,

```
(defthm |valido(p, o, F) => reduccion(p, o) < p|
  (implies (and (k-polinomiop p)
                (k-polinomiosp F)
                (valido p o F))
           (< (reduccion p o) p)))
```

Este teorema no es sencillo de demostrar porque involucra bastantes propiedades de los polinomios (aparte de las de anillo). La demostración se apoya, fundamentalmente, en el corolario del lema 7.23 para el que necesitaremos el lema 7.22 que demostraremos a continuación.

Lema 7.22. *Sean p y q dos polinomios. Si m es un monomio de p, $mp(q) = -m$ y $tp(p+q) \neq tp(p)$ entonces $tp(p+q) < tp(p)$.*

En ACL2,

```
(defthm |m en p & mp(q)=-m & tp(p + q)!=tp(p) => tp(p + q) < tp(p)|
  (implies (and (polinomiop p) (polinomiop q) (en-monomio m p)
                (not (equal (termino (primero (+ p q)))
                            (termino (primero p))))
                (equal (primero q) (RAC-MON::- m)))
           (RAC-TER::< (termino (primero (+ p q)))
                       (termino (primero p)))))
```

Demostración. Sea t_m el término de m. Como m es un monomio de p y $mp(q) = -m$ se tiene que $t_m \leq tp(p)$ y $tp(q) = t_m \leq tp(p)$. Por lo tanto, al ser $tp(p+q) \leq \text{máx}\{tp(p), tp(q)\}$, necesariamente $tp(p+q) \leq tp(p)$ y, como son distintos por hipótesis, $tp(p+q) < tp(p)$. □

Lema 7.23. *Sean p y q dos polinomios. Si m es un monomio de p y $mp(q) = -m$ entonces $p + q < p$.*

En ACL2,

```
(defthm |m en p & mp(q) = - m => p + q < p|
  (implies (and (k-polinomiop p)
                (k-polinomiop q)
                (k-monomiop m) (en-monomio m p)
                (equal (primero q) (RAC-MON::- m)))
           (< (+ p q) p)))
```

Demostración. Por inducción sobre la definición del predicado $<$.

Caso 1: $p + q = 0 \vee p = 0$.

Inmediato, ya que $p \neq 0$.

Caso 2: $p + q \neq 0 \wedge p \neq 0 \wedge tp(p+q) \neq tp(p)$.

En este caso, $tp(p+q) < tp(p)$ por el lema 7.22. Luego $p + q < p$.

Paso inductivo: $p + q \neq 0 \wedge p \neq 0 \wedge tp(p + q) = tp(p)$.

Por hipótesis de inducción se tiene que si m es un monomio de $resto(p)$ y $mp(q) = -m$ entonces

$$resto(p) + q < resto(p)$$

En este caso, sea t_m el término de m; como m es un monomio de p y $mp(q) = -m$, se tiene que $t_m \leq tp(p)$ y $tp(q) = t_m \leq tp(p)$. Por lo tanto, al ser $tp(p) = tp(p + q)$, necesariamente $tp(q) < tp(p)$ porque si $tp(q) = tp(p)$ entonces $mp(p) = m$ y $tp(p + q) < tp(p)$, ya que $mp(q) = -m$. De aquí se obtiene que $t_m \neq tp(p)$.

Por lo tanto, $m \neq mp(p)$ y, como m ha de estar en p, formará parte de $resto(p)$.

Como m es un monomio de $resto(p)$, $mp(q) = -m$ y $tp(p + q) = tp(p)$ entonces $resto(p + q) = resto(p) + q$ y, aplicando hipótesis de inducción, $resto(p + q) < resto(p)$. Luego $p + q < p$.

□

Nota. En la demostración anterior, utilizamos implícitamente el hecho de que si m es un monomio de p cuyo término es t_m y $tp(p) = t_m$ entonces $mp(p) = m$. Recuérdese que esto es así porque estamos considerando una representación normalizada, donde un polinomio no puede contener dos monomios con el mismo término.

Lema 7.24. *Sean p y q dos polinomios no nulos. Si m es un monomio de p divisible por el monomio principal de q y $c = -m/mp(q)$ entonces $p + c \cdot q < p$.*

En ACL2,

```
(defthm |m en p & c = - m / mp(q) => p + c * q < p|
  (let ((c (polinomio (RAC-MON::- (RAC-MON::/ m (primero q))))))
    (implies (and (k-polinomiop p)
                  (k-polinomiop q) (not (nulop q))
                  (k-monomiop m) (en-monomio m p)
                  (dividep (termino (primero q)) (termino m)))
             (< (+ p (* c q)) p))))
```

Demostración. Nótese que $mp(c \cdot q) = (-m/mp(q)) \cdot mp(q) = -m$ y por tanto, podemos aplicar el lema 7.23. □

7.6. Formas normales

Como consecuencia de la sección anterior, es posible definir el concepto de forma normal de un polinomio. Su formalización se encuentra en el fichero `forma-normal.lisp` del directorio `Buchberger`.

Nótese que esta forma normal no tiene nada que ver con la forma normal de un polinomio definida en el capítulo 3.

Definición 7.25. Dada una reducción $\rightarrow$ sobre un conjunto A, se dice que x está en *forma normal* o que es *irreducible* respecto de $\rightarrow$ si no existe ningún $y \in A$ tal que $x \rightarrow y$. En caso contrario, se dice que x es *reducible*.

Definición 7.26. Dada una reducción $\rightarrow$ sobre un conjunto A, se dice que y es una forma normal de x respecto de $\rightarrow$ si $x \rightarrow^* y$ y y es irreducible respecto de A. La notación $x\downarrow$ denota una forma normal de x.

Veamos a continuación como, dado un polinomio, se puede llevar a cabo el cálculo de una forma normal respecto de la reducción asociada a una secuencia finita de polinomios. Nótese, que para formalizar la noción de forma normal, será fundamental definir un *test de reducibilidad*.

Definición 7.27. Dada una reducción $\rightarrow$ sobre un conjunto A, se dice que una función, *reducible*, es un *test de reducibilidad* para $\rightarrow$, si es una función computable $reducible : A \rightarrow A \cup \{x\}$, (donde x es un elemento cualquiera que no pertenece a A) tal que, para todo a reducible, $a \rightarrow reducible(a)$ y, en caso contrario, $reducible(a) = x$.

Para el caso de las reducciones polinómicas, se dice que un polinomio p es reducible por otro f, si existe un monomio en p que es divisible por el monomio principal de f.

7.6.1. Reducibilidad

La función `reducible` implementa este concepto de reducibilidad en ACL2, devolviendo `nil` si no es reducible y un operador en caso contrario.

```
(defun reducible (p f)
  (let ((c (polinomio (RAC-MON::- (RAC-MON::/ (primero p)
                                              (primero f))))))
```

```
(cond ((or (not (polinomiop p))
           (not (polinomiop f))
           (nulop p) (nulop f))
       nil)
      ((dividep (termino (primero f)) (termino (primero p)))
       (operador (primero p) c f))
      (t (reducible (resto p) f))))
```

Teorema 7.28. *Sean p un polinomio, F un conjunto finito de polinomios y $f \in F$.*

$$\mathit{reducible}(p, f) \implies \mathit{válido}(p, \mathit{reducible}(p, f), F)$$

En ACL2,

```
(defthm |reducible(p, fi) => valido(p, reducible(p, fi), F)|
  (implies (and (reducible p fi) (en fi F))
           (valido p (reducible p fi) F)))
```

Demostración. Por inducción sugerida por la función *reducible*.

Caso base: $tp(f) \mid tp(p)$

Inmediato por la definición de *reducible* y *válido*.

Paso de inducción:

La hipótesis de inducción es

$$\mathit{reducible}(\mathit{resto}(p), f) \implies \mathit{válido}(\mathit{resto}(p), \mathit{reducible}(\mathit{resto}(p), f), F)$$

$$\begin{aligned}
&\mathit{reducible}(p, f) \wedge \neg(tp(f) \mid tp(p)) \\
&\implies \mathit{reducible}(p, f) = \mathit{reducible}(\mathit{resto}(p), f) && \text{(def. de } \mathit{reducible}\text{)} \\
&\implies \mathit{válido}(\mathit{resto}(p), \mathit{reducible}(p, f), F) && \text{(hipótesis de inducción)} \\
&\implies \mathit{válido}(p, \mathit{reducible}(p, f), F) && \text{(def. de válido)}
\end{aligned}$$

□

La función `reducible-F`, que se define a continuación, realiza la comprobación de reducibilidad respecto de un conjunto de polinomios. Para esto se necesita la función `reducible`. En caso de que sí sea reducible, la función devuelve el operador asociado. En caso negativo devuelve `nil`.

```
(defun reducible-F (p F)
  (cond ((endp F)
         nil)
        ((reducible p (first F))
         (reducible p (first F)))
        (t
         (reducible-F p (rest F)))))
```

La propiedad fundamental que debe satisfacer la función `reducible-F` es que para todo polinomio `p`, `(reducible-F p)` devuelve un operador aplicable a `p`, siempre que `p` no esté ya en forma normal, y `nil` en caso contrario. La formalización en ACL2 es la siguiente.

```
(defthm |reducible(p, F) => valido(p, reducible(p, F), F)|
  (implies (reducible-F p F)
           (valido p (reducible-F p F) F)))

(defthm |~reducible(p, F) => ~valido(p, o, F)|
  (implies (not (reducible-F p F))
           (not (valido p o F))))
```

7.6.2. Forma normal a partir del test de reducibilidad

Con este test de reducibilidad, calculamos una forma normal, con la siguiente función.

```
(defun<k> fn-F (p F)
  (declare (xargs :measure (RAC-POL::fn p)
                  :well-founded-relation <))
  (if (and (k-polinomiop p) (k-polinomiosp F))
      (let ((red (reducible-F p F)))
        (if red
            (fn-F (reduccion p red) F)
            p))
    p))
```

Nótese que es posible definir la función anterior en ACL2 (es decir, probar su terminación) porque se ha demostrado previamente la buena fundamentación de la relación $<$ sobre los polinomios normalizados (teorema 4.16) y que la reducción es noetheriana (teorema 7.21). En concreto con `declare` se le

indica que, para demostrar la parada de la función, tome como medida la forma normal del polinomio que recibe como primer parámetro y que use como relación bien fundamentada el orden de polinomios.

Teorema 7.29. *Dado un conjunto finito de polinomios F, la función fn_F calcula una forma normal. Es decir, dado un polinomio p*

- $p \rightarrow_F^* fn_F(p)$
- $fn_F(p)$ *es irreducible*

Demostración. Por inducción sobre la definición de fn_F.

Caso base: p es irreducible.

Por definición de fn_F y $\rightarrow_F^*$ se tiene, ya que $p \rightarrow_F^* p = fn_F(p)$ y p es irreducible.

Paso de inducción: p no es irreducible.

En este caso, sea $o = reducible_F(p)$ el operador respecto del cual p se puede reducir y $q = reduccion(p, o)$ el polinomio resultante de reducir p utilizando ese operador. Por definición de $\rightarrow_F$ se tiene que $p \rightarrow_F q$.

Por hipótesis de inducción se tiene que $q \rightarrow_F^* fn_F(q)$ y que $fn_F(q)$ es irreducible. Con lo cual, $p \rightarrow_F^* fn_F(q)$.

Por definición de fn_F, $fn_F(p) = fn_F(q)$ y, por tanto, se tiene que $p \rightarrow_F^* fn_F(p)$ y que $fn_F(p)$ es irreducible.

□

El teorema ACL2 correspondiente se presenta a continuación. Puesto que en nuestra formalización toda equivalencia debe ser justificada dando una prueba, para demostrar la propiedad de normalización, debemos definir una función `prueba-forma-normal` que recibirá un elemento del dominio de definición y devolverá una prueba que justifica la equivalencia de ese elemento con otro en forma normal que es el mismo elemento que calcula la función `fn-F` dada anteriormente. En nuestro caso, para saber si un polinomio es irreducible basta con comprobar que no es válido aplicar ningún operador a ese polinomio.

Estos son los dos teoremas ACL2 que establecen que `(fn-F p F)` es una forma normal de `p`.

```
(defthm |p ->F* fn(p, F)|
  (implies (and (k-polinomiop p)
                (k-polinomiosp F))
           (->* p (fn-F p F) (prueba-forma-normal p F) F)))

(defthm |fn(p, F) irreducible|
  (implies (and (k-polinomiop p)
                (k-polinomiosp F))
           (not (valido (fn-F p F) o F))))
```

La función `prueba-forma-normal`, que nos devuelve una prueba que nos conduce desde un polinomio hasta su forma normal, será obtenida aplicando pasos de reducción hasta que se llegue a un elemento irreducible. Nótese que en la demostración de terminación de esta función (al igual que pasaba con la función `fn-F`) se necesita la buena fundamentación de la relación $<$ entre polinomios y la noetherianidad de la reducción.

```
(defun<k> prueba-forma-normal (p F)
  (declare (xargs :measure (RAC-POL::fn p)
                  :well-founded-relation <))
  (let ((red (reducible-F p F)))
    (if (and red (k-polinomiop p) (k-polinomiosp F))
        (cons (make-r-step
                :direct t
                :elt1 p
                :elt2 (reduccion p red)
                :operator red)
              (prueba-forma-normal (reduccion p red) F))
      nil)))
```

Cada paso de reducción está constituido por una estructura `r-step` que contiene un paso directo de prueba que conecta `p` con `(reduccion p red)`, donde `red` contendrá el operador correspondiente a la reducción. Dicho operador se obtiene con la función `reducible-F`.

Para que la secuencia de tales pasos constituya una prueba, debemos demostrar que todo operador, al aplicarse válidamente sobre un elemento del dominio de definición de la reducción, obtiene un elemento en el mismo dominio de definición.

```
(defthm |valido(p, o, F) => k-polinomiop(reduccion(p, o))|
  (implies (and (k-polinomiop p) (k-polinomiosp F)
```

```
        (valido p o F))
    (k-polinomiop (reduccion p o))))
```

Con todo esto se obtiene, finalmente, la demostración de los teoremas ACL2 `|p ->F* fn(p, F)|` y `|fn(p, F) irreducible|`.

7.7. Cálculo de formas normales

Como se ha visto en la sección anterior, es posible definir una función que calcula formas normales usando el test de reducibilidad. Aunque tal función es idónea para el razonamiento (en concreto, es muy útil para poder obtener el resultado de la sección 8.3.2 mediante instanciación funcional), no es adecuada para usarla en el algoritmo de Buchberger, ya que maneja el concepto «teórico» de operador.

En esta sección nos olvidamos momentáneamente de los operadores y definimos las funciones de reducción que se utilizarán en la implementación del algoritmo de Buchberger. Posteriormente se prueba la relación entre estas funciones y las de operadores. Su formalización se encuentra en el fichero `calculo-forma-normal.lisp`.

La noción de reducción polinómica que se utiliza en el algoritmo de Buchberger puede ser modelada mediante la siguiente función.

Definición 7.30 (reducción polinómica). Sean p, f dos polinomios. Se define $red(p, f)$ de la siguiente forma:

$$red(p, f) = p + cr(p, f) \cdot f$$

donde cr se define como sigue:

- Si no existe ningún monomio en p divisible por $mp(f)$, $cr(p, f) = 0$.
- En caso contrario, $cr(p, f) = -m/mp(f)$ donde m es el mayor monomio de p divisible por $mp(f)$.

De esta forma, la noción de reducción polinómica que se utiliza en el algoritmo de Buchberger puede ser modelada en ACL2 mediante la siguiente función:

```
(defun red (p f)
  (+ p (* (cr p f) f)))
```

donde la función `cr` calcula el *factor de reducción* entre `p` y `q`, y que coincide con un posible c en la definición 7.7.

```
(defun cr (p f)
  (second (crp p f)))
```

Para ello, utiliza la función `crp` que devuelve un par con un booleano que indica si existe o no factor de reducción, y el factor de reducción si existe, (o el polinomio nulo, si no existe).

```
(defun crp (p f)
  (let ((p1 (primero p))
        (f1 (primero f)))
    (cond ((or (nulop p) (not (polinomiop p))
               (nulop f) (not (polinomiop f)))
           (list nil (nulo)))
          ((dividep (RAC-MON::termino f1) (RAC-MON::termino p1))
           (list t (polinomio (RAC-MON::- (RAC-MON::/ p1 f1)))))
          (t
           (crp (resto p) f)))))
```

A continuación se demuestra en el teorema 7.34 una propiedad fundamental: el polinomio reducido es menor que el original (que se demuestra con la ayuda de los teoremas 7.21 y 7.28, y los dos lemas 7.31 y 7.33 que se demuestran a continuación).

Para ello, en primer lugar, se define la función `redp` que reimplementa el concepto de reducibilidad en ACL2. Esta función se define con la ayuda de la función `crp`. Devuelve `t` si `p` es reducible por `f` y `nil` en caso contrario.

```
(defun redp (p f)
  (first (crp p f)))
```

Nótese que la diferencia entre los dos tests de reducibilidad definidos es que *reducible* trabaja con operadores mientras que *redp* no lo hace.

Lema 7.31. *Sean p y f dos polinomios.*

$$redp(p, f) \iff reducible(p, f)$$

En ACL2,

```
(defthm |redp(p, f) <=> reducible(p, f)|
  (iff (redp p f) (reducible p f)))
```

Demostración. Por inducción sobre la definición de *reducible* □

Lema 7.32. *Sean p, f polinomios y c el factor de reducción del operador devuelto por $reducible(p, f)$. Entonces $c = cr(p, f)$.*

```
(defthm |o-factor(reducible(p, f)) = cr(p, f)|
  (implies (reducible p f)
           (equal (o-factor (reducible p f)) (cr p f))))
```

Demostración. Por inducción sobre la definición de *reducible*. □

Lema 7.33. *Sean p y f dos polinomios.*

$$reducible(p, f) \implies reducción(p, reducible(p, f)) = red(p, f)$$

En ACL2,

```
(defthm |reduccion(p, reducible(p, f)) = red(p, f)|
  (implies (reducible p f)
           (equal (reduccion p (reducible p f)) (red p f))))
```

Demostración.

Sea c el factor de reducción del operador devuelto por $reducible(p, f)$. Entonces:

$$\begin{array}{lll} reducible(p, f) \implies & & \\ reducción(p, reducible(p, f)) & = & p + c \cdot f \qquad \text{(def. de } reducción\text{)} \\ & = & p + cr(p, f) \cdot f \qquad \text{(lema 7.32)} \\ & = & red(p, f) \qquad \text{(def. de } red\text{)} \end{array}$$

□

Teorema 7.34. *Sean p y f dos polinomios.*

$$redp(p, f) \implies red(p, f) < p$$

En ACL2,

```
(defthm |redp(p, f) => red(p, f) < p|
  (implies (and (k-polinomiop p)
                (k-polinomiop f)
                (redp p f))
           (< (red p f) p)))
```

Demostración. Sea F un conjunto finito de polinomios donde $f \in F$.

$$
\begin{array}{ll}
& redp(p, f) \\
\Longrightarrow & reducible(p, f) & \text{(lema 7.31)} \\
\Longrightarrow & reducible(p, f) \wedge válido(p, reducible(p, f), F) & \text{(lema 7.28)} \\
\Longrightarrow & reducible(p, f) \wedge reducción(p, reducible(p, f)) < p & \text{(lema 7.21)} \\
\Longrightarrow & red(p, f) < p & \text{(lema 7.33)}
\end{array}
$$

□

Definimos a continuación la función `redp-F`, que reimplementa el concepto de reducibilidad respecto de un conjunto de polinomios en ACL2. Para esto se necesita la función `redp`. En caso de que sí sea reducible, la función devuelve un polinomio del conjunto por el que se puede reducir. En caso negativo devuelve `nil`.

```
(defun redp-F (p F)
  (cond ((endp F)
         nil)
        ((redp p (first F))
         (first F))
        (t
         (redp-F p (rest F)))))
```

Nótese que la diferencia entre los dos tests de reducibilidad sobre conjuntos definidos es que $reducible_F$ trabaja con operadores mientras que $redp_F$ no lo hace.

Lema 7.35.

$$redp_F(p) \iff reducible_F(p)$$

En ACL2,

```
(defthm |reducible(p, F) <=> redp(p, F)|
  (iff (reducible-F p F) (redp-F p F)))
```

Demostración. Por inducción sobre la definición de la función $redp_F$ y utilizando el teorema 7.31. □

El siguiente resultado se utiliza para demostrar la parada de la clausura de la función de reducción *red* respecto a un conjunto de polinomios.

Teorema 7.36. *Sean p un polinomio y F un conjunto de polinomios.*

$$redp_F(p) \implies red(p, redp_F(p)) < p$$

En ACL2,

```
(defthm |redp(p, F) => red(p, redp(p, F)) < p|
  (implies (and (k-polinomiop p)
                (k-polinomiosp F)
                (redp-F p F))
           (< (red p (redp-F p F)) p)))
```

Demostración. Por inducción sobre la definición de $redp_F$ y por el teorema 7.34. □

El siguiente lema expresa una propiedad útil sobre la función $redp_F$ que será utilizada en la sección siguiente.

Lema 7.37. *Sean p un polinomio y F un conjunto de polinomios. Entonces*

$$redp_F(p) \implies redp_F(p) \in F$$

En ACL2,

```
(defthm |redp(p, F) => redp(p, F) en F|
  (implies (and (k-polinomiosp F) (redp-F p F))
           (en (redp-F p F) F)))
```

Demostración. Por inducción sobre la definición de la función $redp_F$ y por el teorema 7.31. □

Y por último, se define la clausura de `red` respecto a un conjunto de polinomios que se nota por `red-F*`.

Definición 7.38. Sean p un polinomio y F conjunto finito de polinomios.

$$red^*_F(p) = \begin{cases} p, & \text{si } \neg redp_F(p) \\ red^*_F(red(p, redp_F(p))), & \text{e.o.c.} \end{cases}$$

donde $redp_F(p)$ indica si el polinomio p puede ser reducido utilizando algún polinomio del conjunto F. En ACL2,

```
(defun<k> red-F* (p F)
  (declare (xargs :measure (RAC-POL::fn p)
                  :well-founded-relation <))
  (if (and (k-polinomiop p) (k-polinomiosp F))
      (let ((reductor (redp-F p F)))
        (if reductor
            (red-F* (red p reductor) F)
          p))
    (nulo)))
```

Nótese que la parada de la función `red-F*` no es inmediata. Es necesario suministrar al sistema una medida apropiada. Para ello se hace uso de la propiedad de buena fundamentación del orden de polinomios demostrada en el capítulo 4 y del teorema 7.36.

Hemos visto que podemos calcular una forma normal sin hacer referencia a operadores. De hecho, red^*_F es una función que calcula formas normales y será la que usemos para computar. Probamos a continuación que fn_F y red^*_F son iguales. Para ello primero demostramos el siguiente lema.

Lema 7.39. *Sean p un polinomio y F un conjunto finito de polinomios.*

$$reducible_F(p) \implies reducción(p, reducible_F(p)) = red(p, redp_F(p))$$

En ACL2,

```
(defthm |r = reducible(p, F) => reduccion(p, r) = red(p, redp(p,F))|
  (implies (reducible-F p F)
           (equal (reduccion p (reducible-F p F))
                  (red p (redp-F p F)))))
```

Demostración. Por inducción sobre la definición de $redp_F$ y por el lema 7.33. □

Teorema 7.40. *Dado un polinomio p y un conjunto finito de polinomios F se tiene que*

$$fn_F(p) = red^*_F(p)$$

En ACL2,

```
(defthm |fn(p, F) = red*(p, F)|
  (implies (and (k-polinomiop p)
                (k-polinomiosp F))
           (equal (fn-F p F) (red-F* p F))))
```

Demostración. Por inducción sobre la definición de fn_F.

Caso base: p es irreducible.

Inmediato, por las definiciones de fn_F y de red^*_F y porque se cumple que $\neg reducible_F(p) \iff \neg redp_F(p)$.

Paso de inducción: p no es irreducible.

Sea $q = reduccion(p, reducible(p, F))$.

Por hipótesis de inducción, $fn_F(q) = red^*_F(q)$. Entonces el resultado se obtiene por los lemas 7.35 y 7.39, y por las definiciones de fn_F y red^*_F.

□

Por último, se demuestra la relación existente entre red^*_F y la reducción $\rightarrow_F$, demostrando que $p \rightarrow^*_F red^*_F(p)$. Esta propiedad será necesaria para aplicar los resultados que se obtengan sobre la relación de reducción al algoritmo de Buchberger, que utiliza la función de reducción red^*_F en lugar de fn_F.

Teorema 7.41. *Sean p un polinomio y F un conjunto finito de polinomios, se tiene que*

$$p \rightarrow^*_F red^*_F(p)$$

En ACL2,

```
(defthm |p ->F* red*(p, F)|
  (let ((prueba (prueba-forma-normal p F)))
    (implies (and (k-polinomiop p)
                  (k-polinomiosp F))
             (->* p (red-F* p F) prueba F))))
```

Demostración. Inmediato por los teoremas 7.29 y 7.40. □

7.8. Estabilidad del ideal respecto a las reducciones

En esta sección demostramos la estabilidad del ideal respecto a las funciones de reducción. Esto será necesario para, después, poder probar la estabilidad del ideal respecto del algoritmo de Buchberger, imprescindible para la demostración de corrección parcial del algoritmo. Su formalización se encuentra en el fichero `estabilidad-reduccion.lisp`.

Teorema 7.42 (clausura del ideal respecto a *red*). *La relación de reducción de polinomios red preserva la pertenencia al ideal.*

$$f \in F \implies (p \in \langle F\rangle \iff red(p,f) \in \langle F\rangle)$$

En ACL2,

```
(defthm |fi en F => (red(p, fi) en <F> <=> p en <F>)|
  (implies (and (k-polinomiop p)
                (k-polinomiop fi)
                (en fi F))
           (iff (en-ideal (red p fi) F) (en-ideal p F))))
```

Demostración. $\Rightarrow$:

$$\begin{array}{ll}
f \in F \wedge p \in \langle F\rangle & \\
\implies f \in \langle F\rangle \wedge p \in \langle F\rangle & (p \in F \Rightarrow p \in \langle F\rangle) \\
\implies f \cdot cr(p,f) \in \langle F\rangle \wedge p \in \langle F\rangle & (p \in \langle F\rangle \Rightarrow p \cdot q \in \langle F\rangle) \\
\implies p + cr(p,f) \cdot f \in \langle F\rangle & (p, q \in \langle F\rangle \Rightarrow p + q \in \langle F\rangle,\ p \cdot q = q \cdot p) \\
\implies red(p,f) \in \langle F\rangle & (\text{def. de } red)
\end{array}$$

$\Leftarrow$:

$$\begin{array}{ll}
f \in F \wedge red(p,f) \in \langle F\rangle & \\
\implies f \in \langle F\rangle \wedge red(p,f) \in \langle F\rangle & (p \in F \Rightarrow p \in \langle F\rangle) \\
\implies f \cdot cr(p,f) \in \langle F\rangle \wedge red(p,f) \in \langle F\rangle & (p \in \langle F\rangle \Rightarrow p \cdot q \in \langle F\rangle) \\
\implies -f \cdot cr(p,f) \in \langle F\rangle \wedge red(p,f) \in \langle F\rangle & (p \in \langle F\rangle \Rightarrow -p \in \langle F\rangle) \\
\implies red(p,f) - f \cdot cr(p,f) \in \langle F\rangle & (p, q \in \langle F\rangle \Rightarrow p + q \in \langle F\rangle) \\
\implies p \in \langle F\rangle & (\text{def. } red \text{ y arit. polinómica})
\end{array}$$

□

Teorema 7.43. *Si p es reducible respecto de un conjunto de polinomios F entonces,*

$$p \in \langle F \rangle \iff red(p, redp_F(p)) \in \langle F \rangle$$

En ACL2,

```
(defthm |redp(p, F) => (red(p, redp(p, F)) en <F> <=> p en <F>)|
  (implies (and (k-polinomiop p)
                (k-polinomiosp F)
                (redp-F p F))
           (iff (en-ideal (red p (redp-F p F)) F) (en-ideal p F))))
```

Demostración.

$$\begin{aligned}
&redp_F(p) \wedge p \in \langle F \rangle \\
&\iff redp_F(p) \in F \wedge p \in \langle F \rangle && \text{(lema 7.37)} \\
&\iff red(p, redp_F(p)) \in \langle F \rangle && \text{(teorema 7.42)}
\end{aligned}$$

□

Teorema 7.44 (clausura del ideal respecto a red^*_F). *La reducción de polinomios red^*_F preserva la pertenencia al ideal. Es decir:*

$$p \in \langle F \rangle \iff red^*_F(p) \in \langle F \rangle$$

En ACL2,

```
(defthm |red*(p, F) en <F> <=> p en <F>|
  (implies (and (k-polinomiop p)
                (k-polinomiosp F))
           (iff (en-ideal (red-F* p F) F) (en-ideal p F))))
```

Demostración. Por inducción en la función red^*_F.

Caso base: $\neg redp_F(p)$

Inmediato ya que por definición $red^*_F(p) = p$.

Paso de inducción: $redp_F(p)$

La hipótesis de inducción es

$$red(p, redp_F(p)) \in \langle F\rangle \iff red^*_F(red(p, redp_F(p))) \in \langle F\rangle$$

$$\begin{array}{lll} & redp_F(p) \wedge p \in \langle F\rangle & \\ \iff & red(p, redp_F(p)) \in \langle F\rangle & \text{(teorema 7.43)} \\ \iff & red^*_F(red(p, redp_F(p))) \in \langle F\rangle & \text{(hipótesis de inducción)} \\ \iff & red^*_F(p) \in \langle F\rangle & \text{(def. de } red^*_F\text{)} \end{array}$$

□

7.9. Resumen

En este capítulo:

- Hemos presentado un tipo particular de relación de reducción sobre polinomios y su formalización en ACL2 en el marco suministrado por las relaciones abstractas.
- Hemos demostrado que la relación de equivalencia generada por la reducción coincide con la congruencia inducida por el ideal.
- Hemos demostrado que dicha relación es noetheriana respecto al orden de polinomios subyacente.
- Se ha llevado a cabo el cálculo de formas normales.
- Hemos demostrado la relación existente entre red^*_F y la reducción $\rightarrow_F$, demostrando que red^*_F calcula una forma normal respecto de $\rightarrow_F$.
- Hemos demostrado que los ideales son cerrados bajo el cálculo de formas normales.

Parte IV

Bases de Gröbner: algoritmo de Buchberger

Capítulo 8

Bases de Gröbner

8.1. Introducción

En 1965, B. Buchberger descubrió que el problema de la pertenencia de un polinomio a un ideal finitamente generado por un conjunto de polinomios (la base del ideal), era equivalente a comprobar que el polinomio fuera reducible (utilizando la noción de reducción polinómica dada en el capítulo anterior) al nulo a través de un nuevo conjunto de polinomios derivados del conjunto original, que forman una nueva base para el ideal derivada a partir de la base original.

En honor a su director de tesis, W. Gröbner, que estimuló su interés por este problema, Buchberger bautizó estas bases con el nombre de *bases de Gröbner*.

En realidad, H. Hironaka había ya descubierto este tipo de bases con antelación, a las que llamó *bases estándar*. Sin embargo, aunque demostró su existencia, su demostración, que no era constructiva, no arrojaba luz sobre el problema de cómo calcularlas.

Buchberger, junto a su demostración, presentó un algoritmo que permitía construir una base de Gröbner a partir de un conjunto de polinomios dado.

Las bases de Gröbner, aunque inicialmente no tuvieron demasiada difusión, tuvieron finalmente un gran impacto en áreas muy diversas. Se emplean fundamentalmente para resolver el problema de la pertenencia al ideal en un anillo de polinomios y para decidir la relación de congruencia inducida por el ideal.

En este capítulo formalizaremos el concepto de bases de Gröbner en ACL2,

y demostraremos sus propiedades fundamentales.

8.2. Confluencia de reducciones abstractas

Para fijar la notación, definiremos primeramente algunos conceptos básicos, sobre confluencia de reducciones abstractas y enunciaremos sin demostración algunas de sus propiedades. Consúltese [1] para una descripción detallada.

Definición 8.1. Una reducción $\rightarrow$ en A se dice que tiene la *propiedad de Church-Rosser* si para todo $a, b \in A$ tal que $a \leftrightarrow^* b$, se tiene que $a \downarrow^* b$.

Teorema 8.2. *Si una reducción $\rightarrow$ en A tiene la propiedad de Church-Rosser y es noetheriana, entonces*

$$a \leftrightarrow^* b \iff a \downarrow = b \downarrow$$

Definición 8.3. Una reducción $\rightarrow$ se dice:

- *Confluente* si $a \leftarrow^* c \rightarrow^* b$ implica que $a \downarrow^* b$.
- *Localmente confluente* si $a \leftarrow c \rightarrow b$ implica que $a \downarrow^* b$.

El siguiente teorema establece que la confluencia es una propiedad equivalente a la propiedad de Church-Rosser.

Teorema 8.4. *Una reducción es confluente si y, sólo si, tiene la propiedad de Church-Rosser.*

El siguiente lema permite reducir la propiedad de Church-Rosser para reducciones noetherianas a la confluencia local.

Lema 8.5. *(de Newman) Una reducción noetheriana y localmente confluente es confluente.*

Combinado el lema de Newman y el lema 8.2 se tiene el siguiente teorema.

Teorema 8.6. *Sea A un conjunto cualquiera, $\equiv$ una relación de equivalencia sobre A y $\rightarrow$ una reducción noetheriana y localmente confluente sobre A tal que $\leftrightarrow^* = \equiv$. Supongamos que existe un test de reducibilidad para $\rightarrow$. Entonces $\equiv$ es decidible.*

Nótese, que la decidibilidad se tiene por comparación de formas normales.

8.3. Bases de Gröbner

Definición 8.7. G es una base de Gröbner del ideal $I = \langle G \rangle$ si

$$p \in I \iff p \to_G^* 0$$

Nótese que si encontramos una base de Gröbner de un ideal, entonces tenemos un procedimiento de decisión para la pertenencia al ideal.

En lo que sigue, veremos un resultado debido a Buchberger, que dice que para determinar si una base G es de Gröbner, en lugar de comprobar que $p \to_G^* 0$ para todo p de un ideal, basta ver que se reducen a 0 una cantidad finita de polinomios, llamados *s-polinomios*. La formalización se los *s-polinomios* se encuentra en `s-polinomio.lisp` y `estabilidad-s-polinomio.lisp`.

Definición 8.8. Sea $m = \text{mcm}(mp(f_i), mp(f_j))$ el mínimo común múltiplo de los monomios principales de los polinomios no nulos f_i y f_j, y m_i y m_j monomios tales que $m_i \cdot mp(f_i) = m = m_j \cdot mp(f_j)$. El *s-polinomio* inducido por f_i y f_j se define como:

$$s\text{-}polinomio(f_i, f_j) = m_i f_i - m_j f_j$$

En ACL2,

```
(defun s-polinomio (fi fj)
  (if (or (nulop fi) (nulop fj))
      (nulo)
    (+ (* (polinomio (cc (primero fi) (primero fj))) fi)
       (* (polinomio (RAC-MON::- (cc (primero fj) (primero fi))))
          fj))))
```

donde `cc` se encarga de calcular los coeficientes m_i y m_j que aparecen en el s-polinomio de f_i y f_j.

```
(defun cc (m1 m2)
  (monomio (RAC::/ (RAC::identidad) (coeficiente m1))
           (RAC-TER::/ (RAC-TER::mcm (termino m1) (termino m2))
                       (termino m1))))
```

Teorema 8.9.

$$f_i \in \langle F \rangle \wedge f_j \in \langle F \rangle \implies s\text{-}polinomio(f_i, f_j) \in \langle F \rangle$$

En ACL2,

```
(defthm |fi en <F> & fj en <F> => s-polinomio(fi, fj) en <F>|
  (implies (and (k-polinomiop fi)
                (k-polinomiop fj)
                (en-ideal fi F)
                (en-ideal fj F))
           (en-ideal (s-polinomio fi fj) F)))
```

Demostración. Inmediata. Ya se demostró que el ideal es cerrado bajo las operaciones polinómicas básicas, que son las que se emplean en la construcción del s-polinomio. □

Es posible dar una interpretación de los s-polinomios desde el punto de vista de la reducción $\rightarrow_F$. Nótese que los s-polinomios representan, de manera compacta, un tipo concreto de picos locales en $\rightarrow_F$, llamados picos locales críticos. Consideremos la siguiente situación crítica que se produce cuando reducimos $m = \text{mcm}(mp(f_i), mp(f_j))$ a la vez por $\rightarrow_{f_i}$ y $\rightarrow_{f_j}$:

$$m - m_i f_i \ {}_{f_i}\!\leftarrow m \rightarrow_{f_j} m - m_j f_j$$

Para representar, de manera compacta, el par crítico $(m - m_i f_i, m - m_j f_j)$ que provoca esta situación se emplea la diferencia entre sus dos componentes, lo que produce el siguiente s-polinomio:

$$\textit{s-polinomio}(f_i, f_j) = m - m_j f_j - (m - m_i f_i) = m_i f_i - m_j f_j$$

El resultado de Buchberger se puede interpretar diciendo que la confluencia de todas estas divergencias críticas asegura la confluencia local, como prueba el teorema 8.12 de la siguiente sección.

Como además la reducción polinómica es noetheriana, esto hace que por el lema de Newman se tenga que la reducción asociada a una base cuyos s-polinomios se reducen a 0 es confluente. Y esta última propiedad veremos que implica el hecho de que la base es de Gröbner y que, por tanto, dicha base proporciona un algoritmo de decisión de la pertenencia al ideal. En líneas generales, las siguientes secciones desarrollan este argumento.

8.3.1. La propiedad Φ y la confluencia local

Considérese la siguiente propiedad $\Phi(F)$ que expresa el hecho de que los s-polinomios formados a partir de polinomios de una base F se reducen a 0.

$$\Phi(F) \equiv \forall p, q \in F \; s\text{-}polinomio(p, q) \rightarrow_F^* 0$$

Nuestro objetivo es demostrar la confluencia local de la relación de reducción inducida por un conjunto de polinomios que verifica la propiedad Φ, tal y como establece el teorema 8.12:

$$\Phi(F) \implies \forall p, q, r \; (r \rightarrow_F p \wedge r \rightarrow_F q \implies p \downarrow_F^* q)$$

Antes, de pasar a la demostración del resultado principal, necesitamos probar los dos lemas siguientes. Su formalización se encuentra en `confluencia.lisp`.

Lema 8.10. *Sean p y q polinomios, y F un conjunto finito de polinomios.*

$$p - q \rightarrow_F^* 0 \implies p \downarrow_F^* q$$

Demostración. Por inducción sobre la longitud de la secuencia de reducción.

Caso base:

Si la longitud de la secuencia de reducción es 0, es porque $p - q = 0$, por lo que $p = q$ y, por tanto, $p \downarrow_F^* q$ trivialmente.

Paso de inducción:

Si la longitud de la secuencia de reducción es mayor que 0 podemos suponer que existe $f_i \in F$ tal que:

$$p - q \rightarrow_{f_i} r \rightarrow_F^* 0$$

Sea $m = c_m \cdot t_m$ el monomio de $p - q$ sobre el que se aplica la reducción, donde c_m y t_m son, respectivamente, el coeficiente y el término del monomio. De acuerdo con la definición de la relación de reducción, se ha de cumplir que $p - q \rightarrow_{f_i} r = (p - q) - (m/mp(f_i)) \cdot f_i$.

Sea c_p el coeficiente de t_m en p y c_q el coeficiente de t_m en q. Claramente, $c_m = c_p - c_q \neq 0$.

Dependiendo de si c_p es o no cero, tenemos que:

$$p = p - \frac{c_p \cdot t_m}{mp(f_i)} \cdot f_i \vee p \rightarrow_{f_i} p - \frac{c_p \cdot t_m}{mp(f_i)} \cdot f_i$$

Dependiendo de si c_q es o no cero, tenemos que:

$$q = q - \frac{c_q \cdot t_m}{mp(f_i)} \cdot f_i \vee q \rightarrow_{f_i} q - \frac{c_q \cdot t_m}{mp(f_i)} \cdot f_i$$

Puesto que $m = (c_p - c_q) \cdot t_m$, se tiene:

$$r = \left(p - \frac{c_p \cdot t_m}{mp(f_i)} \cdot f_i\right) - \left(q - \frac{c_q \cdot t_m}{mp(f_i)} \cdot f_i\right)$$

y se tiene, por hipótesis de inducción, que

$$p - \frac{c_p \cdot t_m}{mp(f_i)} \cdot f_i \downarrow_F^* q - \frac{c_q \cdot t_m}{mp(f_i)} \cdot f_i$$

Por tanto, $p \downarrow_F^* q$.

□

El teorema ACL2 se expresa de la siguiente forma.

```
(defthm |p - q ->*F 0 => p ->*F<- q|
  (let* ((valle (prueba-|p - q ->*F 0 => p ->*F<- q| p q prueba)))
    (implies (and (k-polinomiop p)
                  (k-polinomiop q)
                  (k-polinomiosp F)
                  (->* (+ p (- q)) (nulo) prueba F))
             (->*<- p q valle F))))
```

Nuevamente, como ocurría en teoremas demostrados anteriormente, debemos en primer lugar definir una función que construya la secuencia de pasos de reducción que hay que realizar para obtener $p \downarrow_F^* q$. Esto se hace partiendo de la prueba que contiene la secuencia de pasos de reducción de $p - q$ a 0.

Así, siguiendo la demostración que acabamos de describir, definimos la función `prueba-|p - q ->*F 0 => p ->*F<- q|`. Esta función recibirá los polinomios p y q junto con la prueba que demuestra que $p - q$ se reduce a 0 y calcula la prueba que reduce p a $p - (c_p \cdot t_m / mp(f_i)) \cdot f_i$ y la prueba que reduce q a $q - (c_q \cdot t_m / mp(f_i)) \cdot f_i$. Entonces, produce el resultado deseado recursivamente, empleando $p - (c_p \cdot t_m / mp(f_i)) \cdot f_i$ y $q - (c_q \cdot t_m / mp(f_i)) \cdot f_i$ como parámetros.

```
(defun prueba-|p - q ->*F 0 => p ->*F<- q| (p q prueba)
  (let* ((fi (o-polinomio (operator (first prueba))))
         (m (o-monomio (operator (first prueba))))
         (mp (en-termino (termino m) p))
         (crp (factor-reduccion mp (primero fi)))
         (mq (en-termino (termino m) q))
         (crq (factor-reduccion mq (primero fi))))
    (if (endp prueba)
        nil
      (append (prueba-|p -> p + crp * fi| p mp crp fi)
              (prueba-|p - q ->*F 0 => p ->*F<- q|
                      (+ p (* crp fi)) (+ q (* crq fi)) (rest prueba))
              (prueba-|q + crq * fi <- q| q mq crq fi)))))
```

La función auxiliar `factor-reduccion` recibe los dos monomios que intervienen en un paso de reducción (en concreto, $m_p = c_p \cdot t_m$ y $mp(f_i)$ en un caso y $m_q = c_q \cdot t_m$ y $mp(f_i)$ en el otro) y calcula el polinomio que se emplea como factor en la reducción. El primero de los monomios puede ser cero (siendo `nil` el valor que se le pasa a la función), en cuyo caso se devuelve el polinomio nulo para indicar que la reducción no se realiza.

```
(defun factor-reduccion (m mi)
  (if (not m)
      (nulo)
    (polinomio (RAC-MON::- (RAC-MON::/ m mi)))))
```

La función `prueba-|p -> p + crp * fi|` se encarga de calcular la prueba que reduce p a $p - (c_p \cdot t_m)/mp(f_i) \cdot f_i$. En este caso la prueba se compone de un único paso de prueba directo o de ninguno, dependiendo de si $c_p \neq 0$ o no.

```
(defun prueba-|p -> p + crp * fi| (p mp crp fi)
  (if mp
      (list (make-r-step
             :direct t
             :elt1 p
             :elt2 (+ p (* crp fi))
             :operator (operador mp crp fi)))
    nil))
```

La función `prueba-|q -> q + crq * fi|` se encarga de calcular la prueba que reduce q a $q - (c_q \cdot t_m)/mp(f_i) \cdot f_i$. Este caso es el análogo al anterior, pero cambiando el sentido.

```
(defun prueba-|q + crq * fi <- q| (q mq crq fi)
    (if mq
        (list (make-r-step
               :direct nil
               :elt1 (+ q (* crq fi))
               :elt2 q
               :operator (operador mq crq fi)))
      nil))
```

Para demostrar el teorema, necesitamos asegurarnos de que si se ha producido algún paso de prueba, éste es válido:

```
(defthm |p - q <->F+ 0 => paso-valido(prueba-p -> p + crp * fi)|
  (let* ((o (operator (first prueba)))
         (m (o-monomio o))
         (fi (o-polinomio o))
         (mp (en-termino (termino m) p))
         (crp (factor-reduccion mp (primero fi)))
         (rp (first (prueba-|p -> p + crp * fi| p mp crp fi))))
    (implies (and (k-polinomiop p)
                  (k-polinomiosp F)
                  (<->+ (+ p (- q)) (nulo) prueba F) rp)
             (and (paso-valido rp F)
                  (equal (elt1 rp) p)
                  (equal (elt2 rp) (+ p (* crp fi)))))))

(defthm |p - q <->F+ 0 => paso-valido(prueba-q + crq * fi <- q)|
  (let* ((o (operator (first prueba)))
         (m (o-monomio o))
         (fi (o-polinomio o))
         (mq (en-termino (termino m) q))
         (crq (factor-reduccion mq (primero fi)))
         (rq (first (prueba-|q + crq * fi <- q| q mq crq fi))))
    (implies (and (k-polinomiop q)
                  (k-polinomiosp F)
                  (<->+ (+ p (- q)) (nulo) prueba F) rq)
             (and (paso-valido rq F)
                  (equal (elt1 rq) (+ q (* crq fi)))
                  (equal (elt2 rq) q)))))
```

Para que se pueda aplicar la hipótesis de inducción hay que demostrar que, realmente, r se reduce a 0.

```
(defthm |p - q ->F+ 0 => p + crp * fi - (q + crq * fi) <->F* 0|
  (let* ((o (operator (first prueba)))
         (m (o-monomio o))
         (fi (o-polinomio o))
         (mp (en-termino (termino m) p))
         (crp (factor-reduccion mp (primero fi)))
         (mq (en-termino (termino m) q))
         (crq (factor-reduccion mq (primero fi))))
    (implies (and (polinomiop p)
                  (polinomiop q)
                  (polinomiosp F)
                  (->+ (+ p (- q)) (nulo) prueba F))
             (<->* (+ (+ p (* crp fi)) (- (+ q (* crq fi)))) (nulo)
                   (rest prueba) F))))
```

La descomposición de $r = (p - q) - m/mp(f_i) \cdot f_i$ en la diferencia de los polinomios $p - (c_p \cdot t_m)/mp(f_i) \cdot f_i$ y $q - (c_q \cdot t_m)/mp(f_i) \cdot f_i$ es muy técnica y requiere gran parte de las propiedades de polinomios desarrolladas, además del siguiente teorema:

```
(defthm |m en p - q => cr(mp, mi) - cr(mq, mi) = - m / mi|
  (let* ((mp (en-termino (termino m) p))
         (mq (en-termino (termino m) q)))
    (implies (and (monomiop m)
                  (polinomiop p) (polinomiop q)
                  (en-monomio m (+ p (- q)))
                  (monomiop mi) (not (RAC-MON::nulop mi))
                  (dividep (termino mi) (termino m)))
             (equal (+ (factor-reduccion mp mi)
                       (- (factor-reduccion mq mi)))
                    (polinomio (RAC-MON::- (RAC-MON::/ m mi)))))))
```

Por último, hemos de demostrar que la prueba obtenida es valle.

```
(defthm |valle(prueba-p - q ->*F 0 => p ->*F<- q)|
  (vallep (prueba-|p - q ->*F 0 => p ->*F<- q| p q prueba)))
```

Además del lema anterior es necesario también el siguiente.

Lema 8.11. *Sean m un monomio no nulo, p y q polinomios, y F un conjunto finito de polinomios.*

$$p \rightarrow_F^* q \implies m \cdot p \rightarrow_F^* m \cdot q$$

Demostración. Por inducción sobre la longitud de la secuencia de reducción.

Caso base:

Si la longitud de la secuencia de reducción es 0, es porque $p = q$ y $m{\cdot}p = m{\cdot}q$, resultando $m \cdot p \rightarrow_F^* m \cdot q$ trivialmente.

Paso de inducción:

Si la longitud de la secuencia de reducción es mayor que 0 podemos suponer que existe $f_i \in F$ tal que:

$$p \rightarrow_{f_i} r \rightarrow_F^* q$$

Sea m_p el monomio de p sobre el que se aplica la reducción. De acuerdo con la definición de la relación de reducción, se ha de cumplir que $p \rightarrow_{f_i} r = p - m_p/mp(f_i) \cdot f_i$.

Por hipótesis de inducción:

$$r \rightarrow_F^* q \implies m \cdot r \rightarrow_F^* m \cdot q$$

De esta forma, sólo resta demostrar que:

$$p \rightarrow_{f_i} r \implies m \cdot p \rightarrow_{f_i} m \cdot r$$

Como $m \cdot p$ contiene al monomio $m \cdot m_p$, con coeficiente no nulo, entonces $m{\cdot}r = m{\cdot}(p - m_p/mp(f_i){\cdot}f_i) = m{\cdot}p - (m{\cdot}m_p)/mp(f_i){\cdot}f_i$. Esto demuestra que $m \cdot p \rightarrow_{f_i} m \cdot p - (m \cdot m_p)/mp(f_i) \cdot f_i = m \cdot r$.

□

El teorema ACL2 se expresa de la siguiente forma. Nótese que m se representa mediante un polinomio que contiene un único monomio.

```
(defthm |p ->F* q => m * p ->F* m * q|
  (let ((descendente
         (prueba-|p ->F* q => m * p ->F* m * q| m prueba)))
    (implies (and (->* p q prueba F)
                  (k-polinomiosp F)
                  (k-polinomiop m)
                  (nulop (resto m))
                  (not (nulop m)))
             (->* (* m p) (* m q) descendente F))))
```

Debemos, en primer lugar, definir una función que construya la secuencia de pasos de reducción que hay que realizar para obtener $m \cdot p \rightarrow_F^* m \cdot q$. Esto se hace partiendo de la prueba que contiene la secuencia de pasos de reducción de p a q.

Así, siguiendo la demostración que acabamos de describir, definimos la función `prueba-|p ->F* q => m * p ->F* m * q|`. Esta función recibirá el monomio m y la prueba que demuestra que un polinomio p se reduce a otro q y obtiene la prueba que reduce $m \cdot p$ a $m \cdot q$.

```
(defun prueba-|p ->F* q => m * p ->F* m * q| (m prueba)
  (if (endp prueba)
      nil
    (cons (prueba-|p ->fi r => m * p ->fi m * r| m (first prueba))
          (prueba-|p ->F* q => m * p ->F* m * q| m (rest prueba)))))
```

Para ello, se define la función `prueba-|p ->fi r => m * p ->fi m * r|` que recibe también el monomio m y una prueba de un sólo paso que demuestra que el polinomio p se reduce a r. Esta función calcula una prueba (que también consistirá en un paso) que demuestra que $m \cdot p$ se reduce a $m \cdot r$.

```
(defun prueba-|p ->fi r => m * p ->fi m * r| (m paso)
  (let* ((mp (o-monomio (operator paso)))
         (fi (o-polinomio (operator paso)))
         (nm (RAC-MON::* (primero m) mp))
         (nc (* m (o-factor (operator paso)))))
    (make-r-step
     :direct (direct paso)
     :elt1 (* m (elt1 paso))
     :elt2 (* m (elt2 paso))
     :operator (operador nm nc fi))))
```

Se demuestra que la prueba que se construye es descendente si la prueba original también lo era.

```
(defthm |descendentep(prueba) => descendentep(prueba-*(m, prueba))|
   (implies (descendentep prueba)
            (descendentep
             (prueba-|p ->F* q => m * p ->F* m * q| m prueba))))
```

Por último, para obtener el teorema requerido hemos de demostrar que los pasos de prueba son válidos. Esto es fácil una vez que se comprueba que el

paso que devuelve la función `prueba-|p ->fi r => m * p ->fi m * r|` lo es.

```
(defthm |p ->F+ q => paso-valido(prueba-p ->fi r=>m * p ->fi m * r)|
   (let (paso
         (prueba-|p ->fi r => m * p ->fi m * r| m (first prueba)))
     (implies (and (->+ p q prueba F)
                   (k-polinomiosp F)
                   (k-polinomiop m)
                   (nulop (resto m))
                   (not (nulop m)))
              (paso-valido paso F))))
```

Finalmente, podemos proceder a la demostración del teorema principal de este capítulo, que presentamos a continuación. Éste establece que la relación de reducción inducida por un conjunto finito F es localmente confluente si se verifica $\Phi(F)$.

Teorema 8.12. *La relación de reducción inducida por un conjunto finito F es localmente confluente si se verifica $\Phi(F)$. Esto es:*

$$\Phi(F) \implies \forall p, q, r\ (r \to_F p \wedge r \to_F q \implies p \downarrow_F^* q)$$

Demostración. Podemos distinguir dos casos a partir de:

$$p \leftarrow_{f_i} r \to_{f_j} q$$

Caso 1: Las reducciones se aplican en diferentes monomios de r.

Supongamos que $\to_{f_i}$ se aplica al monomio m_a de r, y que $\to_{f_j}$ se aplica al monomio m_b de r, donde los términos de m_a y m_b son distintos; entonces tenemos la siguiente situación:

- $r = m_a + m_b + r_r$, donde r_r es un polinomio que no tiene ni a m_a ni a m_b.
- De acuerdo con la definición de la relación de reducción, se ha de cumplir que $p = r - m_a/mp(f_i) \cdot f_i = m_b + r_r - m_a/mp(f_i) \cdot resto(f_i)$
- De igual modo, $q = r - m_b/mp(f_j) \cdot f_j = m_a + r_r - m_b/mp(f_j) \cdot resto(f_j)$.

Sin pérdida de generalidad, supongamos que $m_a > m_b$. Entonces m_a es mayor que todos los monomios de $m_b/mp(f_j) \cdot resto(f_j)$. Por esta razón, el polinomio q contiene el monomio m_a, por tanto se puede hacer la reducción $q \rightarrow_{f_i} q - m_a/mp(f_i) \cdot f_i$. Como además se tiene que $r \rightarrow_F q$ entonces

$$p = r - \frac{m_a}{mp(f_i)} \cdot f_i \downarrow_F^* q - \frac{m_a}{mp(f_i)} \cdot f_i$$

por el lema 7.16. Por lo tanto, $p \downarrow_F^* q$.

Caso 2: Las reducciones se aplican sobre el mismo monomio m de r. En este caso tenemos la siguiente situación:

- $r = m + r_r$, donde r_r es un polinomio que no tiene el monomio m.
- De acuerdo con la definición de la relación de reducción, se ha de cumplir que $p = r - m/mp(f_i) \cdot f_i$ y $q = r - m/mp(f_j) \cdot f_j$.

Ya que $mp(f_i)$ y $mp(f_j)$ dividen a m, su mínimo común múltiplo $m_{mcm} = mcm(mp(f_i), mp(f_j))$ también divide a m.

Por hipótesis se tiene que $s\text{-}polinomio(f_i, f_j) \rightarrow_F^* 0$, lo cual implica que $-m/m_{mcm} \cdot s\text{-}polinomio(f_i, f_j) \rightarrow_F^* 0$, , por el lema 8.11. Ya que

$$\begin{aligned} -\frac{m}{m_{mcm}} \cdot s\text{-}polinomio(f_i, f_j) &= -\frac{m}{m_{mcm}} \left(\frac{m_{mcm}}{mp(f_i)} \cdot f_i - \frac{m_{mcm}}{mp(f_j)} \cdot f_j \right) \\ &= -\frac{m}{mp(f_i)} \cdot f_i + \frac{m}{mp(f_j)} \cdot f_j \\ &= p - q \end{aligned}$$

tenemos que $p - q \rightarrow_F^* 0$. Aplicando el lema 8.10 obtenemos que $p \downarrow_F^* q$.

□

El teorema ACL2 se expresa de la siguiente forma.

```
(defthm |Phi(F) => confluencia-local(->F)|
  (let ((valle (transforma-pico-local-F prueba)))
    (implies (and (polinomiop p) (polinomiop q)
                  (pico-localp prueba)
                  (<->*-k p q prueba (F) (k)))
             (and (<->*-k p q valle (F) (k)) (vallep valle)))))
```

La propiedad de confluencia local se reformula mediante los conceptos de prueba con forma de pico local y prueba con forma de valle:

> Una reducción tiene la propiedad de confluencia local si, y sólo si, para toda prueba con forma de pico local existe una prueba equivalente (es decir, que justifica la equivalencia de los mismos elementos) con forma de valle.

Nótese que para suplir la ausencia de cuantificación existencial definiremos una función, que llamaremos `transforma-pico-local-F`, que recibirá una prueba en forma de pico local y devolverá otra equivalente con forma de valle.

Este teorema utiliza un conjunto de polinomios F de k variables, tanto F como k se representa en ACL2 mediante dos funciones abstractas sin argumentos, `F` y `k`, respectivamente, donde `F` está restringida por axiomas a satisfacer la propiedad Φ. Esto es necesario hacerlo así, debido a que F debe satisfacer Φ y esa propiedad no se puede poner como hipótesis de ningún teorema ACL2 debido a su cuantificador universal.

También se define una función binaria abstracta, `prueba-s-polinomio-F`, que está restringida por un axioma a proporcionar una prueba descendente de que el s-polinomio de cualquier par de polinomios de `F` se reduce a 0.

Es decir, para formalizar las hipótesis del teorema se construye un encapsulado que contiene una función binaria, `prueba-s-polinomio-F`, y dos funciones, `F` y `k`, donde `F` está restringida por axiomas a ser un conjunto de polinomios de `k` variables y a satisfacer Φ.

```
(encapsulate
 ((F () t)
  (k () t)
  (prueba-s-polinomio-F (p q) t))

 (local (defun F () nil))
 (local (defun k () 1))
 (local (defun prueba-s-polinomio-F (p q)
          (declare (ignore p q))
          nil))

 (defthm |k-polinomiosp(F())|-k
   (k-polinomiosp-k (F) (k)))

 (defthm |Phi(F)|-k
   (let ((prueba (prueba-s-polinomio-F p q)))
     (implies (and (en p (F)) (en q (F)))
              (->*-k (s-polinomio p q) (nulo) prueba (F) (k))))))
```

Nótese que empleamos versiones de `k-polinomiosp`, `->*` y `<->*` con un parámetro adicional, la k. Esto es así debido a detalles técnicos de la instanciación funcional. Aquí no podemos utilizar un número de variables arbitrario, sino que necesitamos un número concreto, el dado por la función abstracta `k`.

Nuevamente, como en todos los casos, debemos en primer lugar definir una función que construya la secuencia de pasos de reducción que justifica $p \downarrow_F^* q$ a partir de la prueba que tiene la secuencia de pasos para reducir p a r ($p \leftarrow_F q$) y r a q ($r \rightarrow_F q$).

```
(defun transforma-pico-local-F (prueba)
  (let ((m1 (o-monomio (operator (first prueba))))
        (m2 (o-monomio (operator (second prueba)))))
    (cond ((equal (termino m1) (termino m2))
           (transforma-pico-local-F-= prueba))
          ((RAC-TER::< (termino m1) (termino m2))
           (transforma-pico-local-F-< prueba))
          (t
           (transforma-pico-local-F-> prueba)))))
```

Siguiendo la demostración, dividimos el proceso de obtención de la prueba en dos casos (a partir de la situación crítica que tenemos: $p \leftarrow_{f_i} r \rightarrow_{f_j} q$). Nótese que, en realidad, aparecen tres casos en la función, ya que el primer caso de la demostración matemática se debe dividir en dos en la demostración ACL2 (aunque son simétricos).

Caso 1: Las reducciones se aplican en diferentes monomios de r.

Es decir, supongamos que $\rightarrow_{f_i}$ se aplica al monomio m_a de r, y que $\rightarrow_{f_j}$ se aplica al monomio m_b de r, donde los términos de m_a y m_b son distintos.

Caso 1.1: $m_a > m_b$.

En este caso, la función `transforma-pico-local-F->`, a partir de la prueba original en forma de pico local, calcula la prueba valle correspondiente. Para ello, emplea la función `prueba-|p ->F q => r + p ->F<- r + q|` pasándole como parámetros los polinomios r y q, la prueba que reduce r a q (que consta de un único paso) y $-m_a/mp(f_i) \cdot f_i$. Esta función devuelve una prueba valle que une $p = r - m_a/mp(f_i) \cdot f_i$ con $q - m_a/mp(f_i) \cdot f_i$.

Por tanto, ahora sólo es necesario añadir a esa prueba un paso final inverso que lleva a cabo la reducción $q \rightarrow_{f_i} q - m_a/mp(f_i) \cdot f_i$.

```
(defun transforma-pico-local-F-> (prueba)
  (let* ((paso1 (first prueba))
         (paso2 (second prueba)))
    (append (prueba-|p ->F q => r + p ->*<-F r + q|
                    (elt1 paso2)
                    (elt2 paso2)
                    (rest prueba)
                    (* (o-factor (operator paso1))
                       (o-polinomio  (operator paso1))))
            (list (make-r-step
                   :direct nil
                   :elt1 (reduccion (elt2 paso2)
                                    (operator paso1))
                   :elt2 (elt2 paso2)
                   :operator (operator paso1))))))
```

Se demuestra que realmente la función devuelve una prueba valle.

```
(defthm |valle(transforma-pico-local-F->(prueba))|
  (vallep (transforma-pico-local-F-> prueba)))
```

Y se demuestra que es una prueba.

```
(defthm |transforma-pico-local-F->(prueba) es una prueba|
  (let ((valle (transforma-pico-local-F-> prueba))
        (m1 (o-monomio (operator (first prueba))))
        (m2 (o-monomio (operator (second prueba)))))
    (implies (and (k-polinomiosp F)
                  (<->* p q prueba F) (pico-localp prueba)
                  (not (equal (termino m1) (termino m2)))
                  (not (RAC-TER::< (termino m1) (termino m2))))
             (<->* p q valle F))))
```

Caso 1.2: $m_a < m_b$.

En este caso se construye la prueba inversa de la función anterior.

```
(defun transforma-pico-local-F-< (prueba)
  (prueba-inversa
   (transforma-pico-local-F-> (prueba-inversa prueba))))
```

Para ello, se emplea la función `prueba-inversa` que, como su nombre indica, invierte una prueba, utilizando la función `inverse-proof` definida en `abstract-proos.lisp`.

```
(defmacro prueba-inversa (prueba)
  `(inverse-proof ,prueba))

(defun inverse-proof (p)
```

```
    (if (atom p)
        p
      (append (inverse-proof (cdr p))
              (list (inverse-r-step (car p))))))

(defun inverse-r-step (st)
   (make-r-step
    :direct (not (direct st))
    :elt1 (elt2 st)
    :elt2 (elt1 st)
    :operator (operator st)))
```

A continuación, se demuestra que la función devuelve una prueba valle.

```
(defthm |valle(transforma-pico-local-F-<(prueba))|
  (vallep (transforma-pico-local-F-< prueba)))
```

Con el siguiente teorema se comprueba que, realmente, la función devuelve una prueba.

```
(defthm |transforma-pico-local-F-<(prueba) es una prueba|
  (let ((valle (transforma-pico-local-F-< prueba))
        (m1 (o-monomio (operator (first prueba))))
        (m2 (o-monomio (operator (second prueba)))))
    (implies (and (k-polinomiosp F)
                  (<->* p q prueba F) (pico-localp prueba)
                  (RAC-TER::< (termino m1) (termino m2)))
             (<->* p q valle F))))
```

Para lo cual se emplea, entre otros, el siguiente teorema sobre las pruebas inversas.

```
(defthm |p <->F* q => q <->F* p|
  (implies (<->* p q prueba F)
           (<->* q p (prueba-inversa prueba) F)))
```

Caso 2: Las reducciones se aplican sobre el mismo monomio de r.

En este caso, la función `transforma-pico-local-F-=` calcula la prueba valle correspondiente.

```
(defun transforma-pico-local-F-= (prueba)
  (let* ((fi (o-polinomio (operator (first prueba))))
         (fj (o-polinomio (operator (second prueba))))
         (m (o-monomio (operator (first prueba)))))
    (prueba-|p - q ->*F 0 => p ->*F<- q|
     (elt1 (first prueba))
```

```
      (elt2 (second prueba))
      (prueba-|p ->F* q => m * p ->F* m * q|
              (coeficiente-mcm m fj fi)
              (prueba-s-polinomio-F fj fi)))))
```

Siguiendo la demostración presentada previamente, para obtener una prueba valle en este caso, primero hay que calcular la prueba que demuestra que el $s\text{-}polinomio(f_i, f_j)$ se reduce a 0. Esta prueba, por hipótesis, se obtiene mediante la función `prueba-s-polinomio-F`. A continuación se obtiene la prueba de que $-m/m_{mcm} \cdot s\text{-}polinomio(f_i, f_j)$ se reduce a 0 utilizando la función `prueba-|p ->F* q => m * p ->F* m * q|`, y, por último, aplicando la función `prueba-|p - q ->*F 0 => p ->*F<- q|` se compone la prueba valle entre p y q.

Como se ha comentado anteriormente, ya que $mp(f_i)$ y $mp(f_j)$ dividen a m, su mínimo común múltiplo $m_{mcm} = mcm(mp(f_i), mp(f_j))$ también divide a m. Así que podemos calcular m/m_{mcm}. La función `coeficiente-mcm` calcula ese monomio.

```
(defun coeficiente-mcm (m fi fj)
  (polinomio (RAC-MON::/
         m
         (monomio (RAC::identidad)
                  (RAC-TER::mcm (termino (primero fi))
                                (termino (primero fj)))))))
```

A continuación se demuestra que la función devuelve una prueba valle.

```
(defthm |transforma-pico-local-F-=(prueba) es un valle|
  (vallep (transforma-pico-local-F-= prueba)))
```

Finalmente, se demuestra que la función realmente devuelve una prueba.

```
(defthm |transforma-pico-local-F-=(prueba) es una prueba|
  (let* ((valle (transforma-pico-local-F-= prueba))
         (m1 (o-monomio (operator (first prueba))))
         (m2 (o-monomio (operator (second prueba)))))
    (implies (and (polinomiop p) (polinomiop q)
                  (pico-localp prueba)
                  (<->*-k p q prueba (F) (k))
                  (equal (termino m1) (termino m2)))
             (<->*-k p q valle (F) (k)))))
```

La demostración de este teorema es complicada y técnica. Para su demostración se utiliza los teoremas 8.10 y 8.11. Además son necesarias múltiples propiedades de polinomios, entre ellas una que ayuda a determinar que $-m/m_{mcm} \cdot s\text{-}polinomio(f_i, f_j)$ es igual a $p - q$.

```
(local
 (defthm |p - q = coeficiente-mcm(m, fj, fi) * s-polinomio(fj, fi)|
   (let* ((fi (o-polinomio (operator (first prueba))))
          (fj (o-polinomio (operator (second prueba))))
          (m1 (o-monomio (operator (first prueba))))
          (m2 (o-monomio (operator (first prueba)))))
     (implies (and (polinomiop p) (polinomiop q) (k-polinomiosp F)
                   (<->* p q prueba F) (pico-localp prueba)
                   (equal (termino m1) (termino m2)))
              (equal (+ p (- q))
                     (* (coeficiente-mcm m1 fj fi)
                        (s-polinomio fj fi)))))))
```

Por último, se obtiene el teorema que determina que la prueba obtenida con la función `transforma-pico-local-F` es valle con el que se demuestra el teorema original.

```
(defthm |valle(transforma-pico-local-F(prueba))|
  (vallep (transforma-pico-local-F prueba)))
```

8.3.2. Clausura de equivalencia y formas normales

Como $\rightarrow_F$ es noetheriana y localmente confluente cuando se satisface $\Phi(F)$ (es decir, cuando los s-polinomios formados a partir de los polinomios de la base F se reducen a 0), podemos utilizar resultados generales de las reducciones abstractas para concluir que su clausura de equivalencia se puede decidir comprobando la igualdad de formas normales. Su formalización se encuentra en `bases-groebner.lisp`.

Teorema 8.13. *Si F es un conjunto de polinomios tal que $\Phi(F)$, entonces*

$$p \leftrightarrow_F^* q \iff fn_F(p) = fn_F(q)$$

Los teoremas ACL2 correspondientes son los siguientes.

```
(defthm |Phi(F) & p <->*F q => fnF(p) = fnF(q)|
  (implies (and (k-polinomiop-k p (k))
                (k-polinomiop-k q (k))
                (<->*-k p q prueba (F) (k)))
           (fn-equivalentes-k p q (F) (k)))

(defthm |Phi(F) & fnF(p) = fnF(q) => p <->*F q|
  (implies (and (k-polinomiop-k p (k))
                (k-polinomiop-k q (k))
                (fn-equivalentes-k p q (F) (k)))
    (<->*-k p q (prueba-comun-k p q (F) (k)) (F) (k))))
```

Estos dos teoremas se pueden obtener por instanciación funcional de los teoremas ACL2 `r-equivalent-complete` y `r-equivalent-sound`, respectivamente, presentados en [86]. En los cuales se asume la existencia de una reducción abstracta convergente (noetheriana y localmente confluente) y de un test de reducibilidad, y utilizando el lema de Newman y la decidibilidad de la relación de equivalencia descrita por una reducción normalizadora y Church-Rosser obtienen el resultado deseado.

Nótese que empleamos versiones de `fn-equivalentes`, `prueba-comun` y `<->*` con un parámetro adicional, la k. Esto es así debido a detalles técnicos de la instanciación funcional, como se ha comentado anteriormente.

Pasamos a continuación a describir en detalle cómo se lleva a cabo la instanciación funcional y demostración de este resultado.

Instanciación funcional y demostración

En [86], como ya se comentó en el capítulo anterior, se formalizan propiedades generales sobre reducciones abstractas. Estas propiedades son expresadas de manera general, de forma que después se puedan utilizar las correspondientes instancias para cualquier reducción concreta. Por tanto, trabaja con las reducciones sin definirlas completamente, simplemente asume como ciertas las propiedades que se necesitan en la hipótesis del teorema que se quiere demostrar. Atendiendo a estas consideraciones, usa un encapsulado (`encapsulate`) para introducir las funciones necesarias, en particular las tres correspondientes a una reducción abstracta:

- `q`, el dominio de definición.
- `reduce-one-step`, la función que aplica un paso de reducción.

- `legal`, el test de aplicabilidad.

A continuación se va a explicar brevemente como en [86] se demuestra el teorema 8.6, ya que se va a utilizar instanciándolo funcionalmente.

Además de las tres funciones expuestas anteriormente, hay que introducir en el encapsulado funciones adicionales y una serie de propiedades a partir de las cuales se deduce la decidibilidad fuera del ámbito del encapsulado.

Veamos en primer lugar, las funciones adicionales que son necesarias dentro del encapsulado:

- `proof-step-p`, la función que comprueba la corrección de un paso de reducción utilizando el test de aplicabilidad.
- `equiv-p`, función que define la clausura de equivalencia de la relación de reducción.
- `rel`, relación de orden parcial que está bien fundamentada en el dominio definido por `q`. Se utiliza para justifica la noetherianidad de la relación de reducción.
- `fn`, función que hace una inmersión de los elementos del dominio a los ϵ_0-ordinales. Se utiliza en la demostración de la buena fundamentación de `rel`.
- `q-w`, función que devuelve un elemento del dominio.
- `reducible`, test de reducibilidad de los elementos del dominio.
- `transform-local-peak`, función que devuelve una prueba valle equivalente a una prueba en forma de pico local que recibe como parámetro.

A continuación se muestran las propiedades asumidas en el encapsulado:

- `rel-well-founded-relation-on-q`, buena fundamentación de la relación de orden, `rel`.
- `rel-transitive`, transitividad de `rel`.
- `one-element-of-q`, `q-w` es un elemento del conjunto `q`.

- `legal-reduce-one-step-closure`, todo operador legal al aplicarse sobre un elemento del dominio de definición de la reducción obtiene un elemento en el mismo dominio.
- `legal-reducible-1`, la aplicación del test de reducibilidad a un elemento reducible del dominio devuelve un operador aplicable al elemento.
- `legal-reducible-2`, la aplicación del test de reducibilidad a un elemento no reducible o en forma normal del dominio devuelve `nil`.
- `local-confluence`, para toda prueba en forma de pico local existe una prueba valle equivalente. Esta prueba valle viene dada por la función `transform-local-peak`.
- `noetherian`, la relación de reducción es noetheriana.

El encapsulado completo en ACL2 aparece a continuación:

```
(encapsulate
 ((rel (x y) boolean)
  (q (x) boolean)
  (q-w () elemement)
  (fn (x) e0-ordinalp)
  (legal (x u) boolean)
  (reducible (x) boolean)
  (reduce-one-step (x u) element)
  (transform-local-peak (x) proof))

 (local (defun rel (x y) (declare (ignore x y)) nil))
 (local (defun q (x) (declare (ignore x)) t))
 (local (defun fn (x) (declare (ignore x)) 1))

 (defthm rel-well-founded-relation-on-q
   (and
    (implies (q x) (e0-ordinalp (fn x)))
    (implies (and (q x) (q y) (rel x y))
             (e0-ord-< (fn x) (fn y))))
   :rule-classes (:well-founded-relation
                  :rewrite))

 (defthm rel-transitive
   (implies (and (q x) (q y) (q z) (rel x y) (rel y z))
```

```
           (rel x z)))

(in-theory (disable rel-transitive))

(local (defun q-w () 0))

(defthm one-element-of-q (q (q-w)))

(local (defun legal (x u) (declare (ignore x u)) nil))
(local (defun reduce-one-step (x u) (+ x u)))

(defun proof-step-p (s)
  (let ((elt1 (elt1 s)) (elt2 (elt2 s))
        (operator (operator s)) (direct (direct s)))
    (and (r-step-p s)
         (implies direct
                  (and (legal elt1  operator)
                       (equal (reduce-one-step elt1 operator)
                              elt2)))
         (implies (not direct)
                  (and (legal elt2 operator)
                       (equal (reduce-one-step elt2 operator)
                              elt1))))))

(defun equiv-p (x y p)
  (if (endp p)
      (and (equal x y) (q x))
      (and
       (q x)
       (proof-step-p (car p))
       (equal x (elt1 (car p)))
       (equiv-p (elt2 (car p)) y (cdr p)))))

(defthm legal-reduce-one-step-closure
  (implies (and (q x) (legal x op))
           (q (reduce-one-step x op))))

(local (defun reducible (x) (declare (ignore x)) nil))

(defthm legal-reducible-1
  (implies (and (q x) (reducible x))
           (legal x (reducible x))))
```

```
(defthm legal-reducible-2
  (implies (and (q x) (not (reducible x)))
           (not (legal x u))))

(local (defun transform-local-peak (x) (declare (ignore x)) nil))

(defthm local-confluence
  (let ((valley (transform-local-peak p)))
    (implies (and (equiv-p x y p) (local-peak-p p))
             (and (steps-valley valley)
                  (equiv-p x y valley)))))

(defthm noetherian
  (implies (and (q x) (legal x u))
           (rel (reduce-one-step x u) x))))
```

Además, se define externamente al encapsulado el procedimiento de decisión, `r-equivalent`, y que utiliza el teorema `r-equivalent-complete` como veremos a continuación. Este procedimiento se limita a comprobar la igualdad de las formas normales de los dos elementos que recibe como entrada.

```
(defun r-equivalent (x y)
  (equal (normal-form x) (normal-form y)))
```

donde la función `normal-form` es la siguiente:

```
(defun normal-form (x)
  (declare (xargs :measure (if (q x) x (q-w))
                  :well-founded-relation rel))
  (if (q x)
      (let ((red (reducible x)))
        (if red
            (normal-form (reduce-one-step x red))
            x))
    x))
```

Por último, se muestra la función `make-proof-common-n-f` que se utiliza en la prueba de validez.

```
(defun make-proof-common-n-f (x y)
   (append (proof-irreducible x)
           (inverse-proof (proof-irreducible y))))
```

donde `proof-irreducible` es como sigue:

```
(defun proof-irreducible (x)
  (declare (xargs :measure (if (q x) x (q-w))
                  :well-founded-relation rel))
  (if (q x)
      (let ((red (reducible x)))
        (if red
            (cons (make-r-step
                   :direct t :elt1 x :elt2 (reduce-one-step x red)
                   :operator red)
                  (proof-irreducible (reduce-one-step x red)))
          nil))
    nil))
```

Finalmente, la decidibilidad de la relación de equivalencia asociada se demuestra con los teoremas `r-equivalent-complete` y `r-equivalent-sound` que se muestran a continuación.

```
(defthm r-equivalent-complete
  (implies (equiv-p x y p)
           (r-equivalent x y)))

(defthm r-equivalent-sound
  (implies (and (q x) (q y) (r-equivalent x y))
           (equiv-p x y (make-proof-common-n-f x y))))
```

En nuestro caso, ya tenemos definida la reducción convergente necesaria y demostradas todas las propiedades que se asumen en el encapsulado.

Por tanto, ahora sólo debemos definir un algoritmo de decisión para la misma, demostrando que es correcto y completo. Este algoritmo se limita, igual que su correspondiente en la versión abstracta, a comprobar la igualdad de las formas irreducibles de los dos elementos que se reciben como entrada.

```
(defun<k> fn-equivalentes (p q F)
  (equal (fn-F p F) (fn-F q F)))
```

Por último hay que definir la función `prueba-comun` que es utilizada en la prueba de validez. Esta función se implementa, siguiendo el mismo esquema que su correspondiente abstracta, utilizando `prueba-forma-normal` presentada en la sección 7.6 del capítulo anterior.

```
(defun<k> prueba-comun (p q F)
   (append (prueba-forma-normal p F)
           (prueba-inversa (prueba-forma-normal q F))))
```

En este momento, como ya se ha comentado, los resultados probados para reducciones abstractas se pueden trasladar a nuestra reducción concreta de polinomios. Hemos representado nuestra reducción polinómica mediante operadores, hemos definido específicamente las correspondientes funciones que definen el dominio (`k-polinomiop`), un paso de reducción (`reduccion`) y la aplicabilidad (`valido`), y hemos probado que tiene las propiedades asumidas en el teorema de decidibilidad de la relación de equivalencia asociada a reducciones abstractas. Por tanto, el mecanismo de instanciación funcional permitirá hacer uso del teorema para deducir un resultado análogo para la reducción concreta de polinomios.

Veamos cómo quedan enunciados finalmente los dos teoremas ACL2 que se quieren demostrar mediante este mecanismo. Nótese como hay que indicar al sistema que las demostraciones se harán por instanciación funcional de los teoremas, `r-equivalent-complete` y `r-equivalent-sound`, junto con la correspondencia entre las funciones abstractas y las concretas de polinomios:

```
(defthm |Phi(F) & p <->*F q => fnF(p) = fnF(q)|
  (implies (and (k-polinomiop-k p (k))  (k-polinomiop-k q (k))
                (<->*-k p q prueba (F) (k)))
           (fn-equivalentes-k p q (F) (k)))
  :hints (("Goal"
           :use (:functional-instance
                 (:instance CNV::r-equivalent-complete
                            (CNV::x p) (CNV::y q) (CNV::p prueba))
                 (CNV::q (lambda (p) (k-polinomiop-k p (k))))
                 (CNV::q-w (lambda () (nulo)))
                 (CNV::reduce-one-step reduccion)
                 (CNV::legal (lambda (p o) (valido p o (F))))
                 (CNV::proof-step-p
                  (lambda (prueba) (paso-valido prueba (F))))
                 (CNV::equiv-p
                  (lambda (p q prueba) (<->*-k p q prueba (F) (k))))
                 (CNV::transform-local-peak transforma-pico-local-F)
                 (CNV::fn RAC-NPOL::polinomio->e0-ordinal)
                 (CNV::rel <)
                 (CNV::reducible (lambda (p) (reducible-F p (F))))
                 (CNV::normal-form (lambda (p) (fn-F-k p (F) (k))))
```

```
                (CNV::r-equivalent
                 (lambda (p q)
                  (fn-equivalentes-k p q (F) (k))))))))


(defthm |Phi(F) & fnF(p) = fnF(q) => p <->*F q|
  (implies (and (k-polinomiop-k p (k))
                (k-polinomiop-k q (k))
                (fn-equivalentes-k p q (F) (k)))
    (<->*-k p q (prueba-comun-k p q (F) (k)) (F) (k)))
  :hints (("Goal"
           :use (:functional-instance
                 (:instance CNV::r-equivalent-sound
                            (CNV::x p) (CNV::y q))
                 (CNV::q (lambda (p) (k-polinomiop-k p (k))))
                 (CNV::q-w (lambda () (nulo)))
                 (CNV::reduce-one-step reduccion)
                 (CNV::legal (lambda (p o) (valido p o (F))))
                 (CNV::proof-step-p
                  (lambda (prueba) (paso-valido prueba (F))))
                 (CNV::equiv-p
                  (lambda (p q prueba) (<->*-k p q prueba (F) (k))))
                 (CNV::transform-local-peak transforma-pico-local-F)
                 (CNV::fn RAC-NPOL::polinomio->e0-ordinal)
                 (CNV::rel <)
                 (CNV::reducible (lambda (p) (reducible-F p (F))))
                 (CNV::normal-form (lambda (p) (fn-F-k p (F) (k))))
                 (CNV::r-equivalent
                  (lambda (p q) (fn-equivalentes-k p q (F) (k))))
                 (CNV::proof-irreducible
                  (lambda (p) (prueba-forma-normal-k p (F) (k))))
                 (CNV::make-proof-common-n-f
                  (lambda (p q) (prueba-comun-k p q (F) (k))))))))
```

La demostración de estos dos teoremas es inmediata, ya que en las secciones anteriores se han demostrado las obligaciones de prueba generadas.

8.3.3. La propiedad Φ y las bases de Gröbner

Una vez que se tiene definida la relación de reducción polinómica y se han demostrado las propiedades anteriores, estamos en condiciones de poder demostrar el siguiente teorema.

Teorema 8.14. *Si I es un ideal y F una base de I tal que $\Phi(F)$, entonces F es una base de Gröbner de I. Es decir,*

$$p \in \langle F \rangle \iff p \to_F^* 0$$

Demostración. Demostremos primero que $p \in \langle F \rangle \implies p \to_F^* 0$.

$$\begin{array}{lll} p \in \langle F \rangle & \implies p \equiv_F 0 & (\text{def. de } \equiv_F) \\ & \implies p \leftrightarrow_F^* 0 & (\text{ya que } p \equiv_F q \implies p \leftrightarrow_F^* q) \end{array}$$

Por el teorema 8.12

$$\begin{array}{lll} p \leftrightarrow_F^* 0 & \implies fn_F(p) = fn_F(0) & (\text{ya que } p \leftrightarrow_F^* q \iff fn_F(p) = fn_F(q)) \\ & \implies red_F^*(p) = red_F^*(0) & (\text{ya que } fn_F(p) = red_F^*(p)) \\ & \implies red_F^*(p) = 0 & (\text{def. } red_F^*) \\ & \implies p \to_F^* 0 & (\text{ya que } p \to_F^* red_F^*(p)) \end{array}$$

A continuación, demostremos que $p \to_F^* 0 \implies p \in \langle F \rangle$.

$$\begin{array}{lll} p \to_F^* 0 & \implies p \leftrightarrow_F^* 0 & (\text{ya que } \to^* \subseteq \leftrightarrow^*) \\ & \implies fn_F(p) = fn_F(0) & (\text{ya que } p \leftrightarrow_F^* q \iff fn_F(p) = fn_F(q)) \\ & \implies red_F^*(p) = red_F^*(0) & (\text{ya que } fn_F(p) = red_F^*(p)) \\ & \implies red_F^*(p) = 0 & (\text{def. de } red_F^*) \\ & \implies red_F^*(p) \in \langle F \rangle & (\text{ya que } 0 \in \langle F \rangle) \\ & \implies p \in \langle F \rangle & (\text{ya que } p \in \langle F \rangle \iff red_F^*(p) \in \langle F \rangle) \end{array}$$

□

Los teoremas ACL2 correspondientes se presentan a continuación, donde `prueba-forma-normal` es la función que construye una prueba que conduce desde un polinomio hasta su forma normal y que se expuso en el capítulo anterior.

Nótese que seguimos considerando `(F)` y `(k)` de la subsección anterior.

La primera parte queda descrita de la siguiente forma.

```
(defthm |Phi(F) & p en <F> => p ->*F 0|
  (let* ((prueba (prueba-forma-normal-k p (F) (k))))
    (implies (and (k-polinomiop-k p (k))
                  (en-ideal-k p (F) (k)))
             (->*-k p (nulo) prueba (F) (k)))))
```

Este teorema se demuestra aplicando el teorema 7.41 y, el siguiente teorema que demuestra que si p pertenece a $\langle F \rangle$, p se reduce a 0 respecto a F.

```
(defthm |Phi(F) & p en <F> => red*(p, F) = 0|
    (implies (and (k-polinomiop-k p (k))
                  (en-ideal-k p (F) (k)))
             (equal (red-F*-k p (F) (k)) (nulo))))
```

Este teorema se demuestra utilizando los teoremas 7.40, 7.18 y 8.13.

La segunda parte es bastante más sencilla y se puede puede expresar en ACL2 de la siguiente forma.

```
(defthm |Phi(F) & p ->*F 0 => p en <F>|
    (implies (and (k-polinomiop-k p (k))
                  (<->*-k p (nulo) prueba (F) (k)))
             (en-ideal-k p (F) (k))))
```

Este teorema se demuestra aplicando los teoremas 7.40 y 8.13 y, el siguiente teorema que demuestra que si p se reduce a 0 respecto a F, p pertenece a $\langle F \rangle$,

```
(defthm |red*(p, F) = 0 => p en <F>|
  (implies (and (k-polinomiop p)
                (k-polinomiosp F)
                (equal (red-F* p F) (nulo)))
           (en-ideal p F)))
```

Finalmente, este teorema se demuestra aplicando estabilidad del ideal respecto de las funciones de reducción.

8.4. Resumen

En este capítulo:

- Hemos presentado el concepto de base de Gröbner.
- Hemos formalizado el concepto de s-polinomio en ACL2 y demostrado sus propiedades fundamentales.

- Hemos demostrado la confluencia local de la relación de reducción inducida por un conjunto de polinomios que verifica la propiedad Φ.
- Hemos demostrado que la clausura de equivalencia inducida por un conjunto de polinomios que verifica la propiedad Φ se puede decidir comprobando la igualdad de formas normales.
- Finalmente, hemos demostrado que si I es un ideal y F una base de I tal que $\Phi(F)$ entonces F es una base de Gröbner de I.

Capítulo 9

Algoritmo de Buchberger

9.1. Introducción

Ya se ha visto la importancia de las bases de Gröbner para decidir la pertenencia al ideal. En este capítulo veremos cómo obtener una base de Gröbner de un ideal dado.

Para ver si una base de un ideal es de Gröbner, lo que se hace es comprobar si los s-polinomios se reducen a cero. Si no es el caso, existe una solución trivial para que el s-polinomio se reduzca a cero, y es añadirlo a la base. Este nuevo polinomio generará nuevos s-polinomios, cuya reducción a cero debe ser comprobada, y así sucesivamente. Es posible demostrar que este proceso termina, obteniendo como resultado una base de Gröbner del ideal de partida. Este algoritmo se conoce como algoritmo de Buchberger.

El algoritmo tiene aplicaciones en el álgebra, el razonamiento automático, la robótica y otras áreas. Se emplea fundamentalmente para resolver el problema de la pertenencia al ideal en un anillo de polinomios.

Nuestro objetivo en este capítulo es la implementación y verificación de este algoritmo que Buchberger diseñó para calcular bases de Gröbner en ACL2.

9.2. Algoritmo de Buchberger

Seguidamente se define la función principal que lleva a cabo todo el proceso de cálculo de una base de Gröbner a partir de una base inicial $F = \{f_1, \ldots, f_n\}$

compuesta por una secuencia finita de polinomios.

El algoritmo de Buchberger puede expresarse esquemáticamente de la forma descrita en la figura 9.1, donde :: representa la construcción de secuencias y ++ la concatenación.

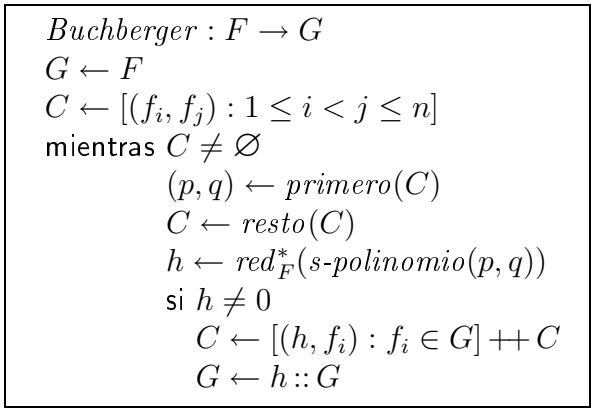

Figura 9.1: Algoritmo de Buchberger

La implementación en ACL2 presentada en `buchberger.lisp` usa dos funciones. La primera función actúa como punto de entrada, calculando el conjunto inicial de parejas con el que comenzará a trabajar la segunda y llamando a esta segunda función sobre tales parejas. La segunda función, que es la que realiza todo el trabajo, procede recursivamente implementando el cuerpo del bucle que aparece en la figura.

```
(defmacro Buchberger (F)
  `(Buchberger-aux ,F (parejas-iniciales ,F)))

(defun<k> Buchberger-aux (F C)
  (declare (xargs :mode :program))
  (if (and (naturalp k)
           (k-polinomiosp F)
           (pares-k-polinomiosp C))
      (if (endp C)
          F
        (let* ((p (first (first C)))
               (q (second (first C)))
               (h (red-F* (s-polinomio p q) F)))
          (if (equal h (nulo))
              (Buchberger-aux F (rest C))
            (Buchberger-aux (cons h F)
```

```
                              (append (parejas h F) (rest C))))))
    F))
```

Para completar en ACL2 el algoritmo precedente es necesario previamente definir varias funciones. La función `parejas-iniciales` devuelve todas las parejas que se pueden formar con los elementos de un conjunto.

```
(defun parejas-iniciales (F)
  (if (endp F)
      nil
    (append (parejas (first F) (rest F))
            (parejas-iniciales (rest F)))))
```

La función `parejas` calcula las parejas formadas por un elemento y todos los de otro conjunto.

```
(defun parejas (p F)
  (if (endp F)
      nil
    (cons (list p (first F))
          (parejas p (rest F)))))
```

La función `Buchberger-aux` se ha definido en modo programa, con lo que aún no ha sido admitida bajo el principio de definición. Obsérvese que se ha de suministrar una medida y una relación bien fundamentada, para poder demostrar la parada del algoritmo, probando que dicha medida decrece (respecto de la relación bien fundamentada) en cada llamada recursiva. Esta cuestión será estudiada a continuación.

9.3. Terminación

Los argumentos sobre la terminación de las funciones en ACL2 se basan en la noción de estructuras bien fundamentadas. Una estructura bien fundamentada es un par $\langle A, < \rangle$ consistente en un conjunto A y una relación $<$, tal que no hay una sucesión infinita $\{a_i\}$ de elementos de A tal que $a_{i+1} < a_i$. Es decir, no se pueden formar secuencias estrictamente decrecientes de longitud infinita con los elementos de A.

Los ordinales de ACL2, reconocidos por el predicado `e0-ordinalp`, con su relación de orden, `e0-ord-<`, constituyen una estructura bien fundamentada [1] y son utilizados para demostrar la terminación de las funciones definidas en el sistema.

Para abordar la terminación del algoritmo de Buchberger emplearemos un producto lexicográfico de dos relaciones bien fundamentadas. La razón es que en el algoritmo, en la primera rama recursiva, el primer parámetro permanece inalterado mientras el segundo decrece estructuralmente —se elimina un elemento de la secuencia— y, por otro lado, en la segunda rama, el primer parámetro decrece en un cierto sentido pese a que se le añade un nuevo polinomio. Este decrecimiento es consecuencia del lema de Dickson.

Lema 9.1 (de Dickson). *Dada una sucesión* $\{s_n\}$ *en* $\mathbb{N}^k$*, existen* i *y* j *tales que* $i < j$ *y* $s_i \leq_k s_j$ *donde* $[a_1, \ldots, a_k] \leq_k [b_1, \ldots, b_k]$ *si* $a_i \leq b_i$ *para todo* $1 \leq i \leq k$*.*

Las demostraciones clásicas de este teorema [79] no son constructivas y por tanto no podemos utilizarlas tal cual en ACL2.

Los términos pueden verse como elementos de $\mathbb{N}^k$ donde la relación de divisibilidad juega el papel de $\leq_k$. Por tanto, el lema anterior establece que a una secuencia de términos no se le puede añadir indefinidamente términos que no sean divisibles por ninguno de los términos que ya estuvieran en ella.

Consideremos, ahora, la secuencia de términos principales de los polinomios del primer parámetro de la función `Buchberger-aux`. En este caso, el lema anterior establece que a una secuencia de polinomios no se le puede añadir sin fin polinomios cuyos términos principales no sean divisibles por ninguno de los términos principales de los polinomios que ya estuvieran en la secuencia. Esto es justamente lo que ocurre en la segunda rama del algoritmo: el polinomio, h, que se añade a la base no es nulo y es irreducible por F. En consecuencia, su término principal no es divisible por ninguno de los términos principales de los polinomios que ya estuvieran en la base.

Ejemplo 9.2. Si los términos se representan mediante k-uplas de sus exponentes, (esto es, el término $x^3y^2z^5$ se representaría por $[3, 2, 5]$) entonces se puede considerar un orden $<$ sobre secuencias de términos donde aparezcan situaciones como la siguiente, en la que cada k-upla añadida no es divisible

[1] Esto se asume en la meta-teoría.

por las anteriores:

$$\begin{aligned}
&[[3,2,5]] > \\
&[[2,6,8],[3,2,5]] > \\
&[[20,50,4],[2,6,8],[3,2,5]] > \\
&[[15,1,80],[20,50,4],[2,6,8],[3,2,5]]
\end{aligned}$$

Por el lema de Dickson sabemos que no podemos mantenernos continuamente insertando términos que no sean divisibles por los demás términos de la secuencia. Por tanto, $<$ es bien fundamentado.

El problema está en definir ese orden para el primer parámetro y demostrar en ACL2, mediante inmersión en los ϵ_0-ordinales, que está bien fundamentado. Una vez demostrado, es fácil combinarlo con el orden estructural del otro parámetro mediante un producto lexicográfico.

Esto es, dada una secuencia de términos que tiene la propiedad que ningún término divide a otro más allá en la secuencia, hay que construir una inmersión de la secuencia de términos en los ordinales, de manera que la secuencia de ordinales sea estrictamente decreciente.

Ésta es una reformulación «finita» del lema de Dickson, pero es justo lo que se necesita si un determinado argumento de terminación confía en esta propiedad de las secuencias de términos. Y éste es precisamente el caso del algoritmo de Buchberger.

Siguiendo estas líneas se ha definido una medida para demostrar la terminación del algoritmo de Buchberger a partir de la formalización que se hace del lema de Dickson en [64] y contenida en el directorio `dickson`. En [78] también se desarrolla prácticamente la misma idea matemática.

Otra formalización del lema de Dickson en ACL2 pueden encontrarse en [97]. Esta formalización se realiza mediante una inmersión directa en los ϵ_0-ordinales y hace un uso frecuente de diversas propiedades de la aritmética ordinal [59]. También existen formalizaciones en MIZAR y COQ [56, 80].

9.3.1. Formalización del lema de Dickson

En la formalización del lema de Dickson presentada en [64], la representación de términos de k variables coincide con la nuestra. Los términos son representados por k-uplas de números naturales formadas por los exponentes.

La relación de divisibilidad entre términos coincide con la siguiente relación entre tuplas: dada las k-uplas $A = [a_1, \ldots, a_k]$ y $B = [b_1, \ldots, b_k]$, $A \leq_k B$ si, y sólo, si $a_i \leq b_i$ para todo $1 \leq i \leq k$. La función `tuple-<` implementa esta relación.

Los siguientes teoremas ACL2 permiten formalizar el lema de Dickson.

```
(defthm get-dickson-indices-1
  (implies (naturalp n)
           (let ((n1 (first (get-dickson-indices n)))
                 (n2 (second (get-dickson-indices n))))
             (< n1 n2))))

(defthm get-dickson-indices-2
   (implies (naturalp n)
            (let ((n1 (first (get-dickson-indices n)))
                  (n2 (second (get-dickson-indices n))))
              (tuple-< (f n1) (f n2)))))
```

donde la función `get-dickson-indices` y, las funciones `get-tuple-<-f` y `tuple-<` necesarias para su definición son como sigue.

```
(defun get-dickson-indices (n)
  (declare (xargs :measure (pattern-list-measure
                            (pattern-list-tuple-list
                             (initial-segment-f n)
                             (list (initial-pattern (K)))))
                  :well-founded-relation mul-e0-ord-<))
  (if (naturalp n)
      (let ((m (get-tuple-<-f (+ n 1) (f (+ n 1)))))
        (if m (list m (+ n 1))
            (get-dickson-indices (+ n 1))))
      nil))

(defun get-tuple-<-f (m TT)
  (if (naturalp m)
      (cond ((= m 0)
             nil)
            ((tuple-< (f (- m 1)) TT)
             (- m 1))
            (t
             (get-tuple-<-f (- m 1) TT)))
```

```
      nil))

(defun tuple-< (Tn Tm)
  (cond ((endp Tn) (endp Tm))
        ((endp Tm) (endp Tn))
        ((and (naturalp (car Tn))
              (naturalp (car Tm)))
         (and (<= (car Tn) (car Tm))
              (tuple-< (cdr Tn) (cdr Tm))))
        (t nil)))
```

Los teoremas anteriormente presentados hacen uso, además de la función `get-dickson-indices`, de un encapsulado en el cual se define una función constante k, que representa el número de variables [2], y una función unaria f que suministra una secuencia infinita de k-uplas de números naturales. De esta forma, las propiedades que se demuestren sobre esa secuencia infinita serán válidas para cualquier otra secuencia infinita de k-uplas de números naturales.

Nótese que estos teoremas aseguran que para cualquier secuencia infinita de k-uplas $\{f_k\}$, existen $i < j$ tales que f_i divide a f_j, y la función `get-dickson-indices` suministra estos valores.

La principal dificultad de la demostración se encuentra en la prueba de terminación de la función `get-dickson-indices`. Para ello se define una medida sobre los parámetros de la función que decrece en cada llamada recursiva con respecto a una relación bien fundamentada. La definición de esa medida, y la prueba de que decrece, constituyen la clave principal de la formalización del lema de Dickson en [64]. Esa medida será la que reutilizaremos para construir una medida que permita demostrar la terminación del algoritmo de Buchberger.

La idea es la siguiente: para cada número natural n, se construye un conjunto de *patrones* asociados con las n primeras k-uplas, $f_1, \ldots, f_n$, suministradas por f. Estos patrones representan los conjuntos de tuplas que no verifican la condición de terminación de `get-dickson-indices` (es decir, que no son divisibles por ninguna de las tuplas anteriores). Cada patrón tiene un número natural asociado (su dimensión). Un conjunto finito de patrones tiene asociado, por tanto, un multiconjunto de números naturales, y se demuestra que estos multiconjuntos decrecen con respecto a la extensión a multiconjuntos de `e0-ord-<`, presentada en [88, 63]. La medida viene dada por la siguiente

[2] O lo que es lo mismo, la longitud de las tuplas.

función.

```
(defun dickson-indices-measure (n)
  (pattern-list-measure
   (pattern-list-tuple-list (initial-segment-f n)
                            (list (initial-pattern (K))))))
```

donde

1. `pattern-list-measure` recibe un multiconjunto de patrones y devuelve el multiconjunto de las dimensiones de los patrones.
2. `pattern-list-tuple-list` devuelve el multiconjunto de patrones asociado con una secuencia de tuplas, representando el conjunto de tuplas que no son divisibles por ninguna de las tuplas de la secuencia.
3. `initial-segment-f` recibe un número natural n y devuelve las n primeras tuplas suministradas por f.
4. `initial-pattern` recibe un número natural k y devuelve el patrón asociado a una lista vacía de k-uplas (es decir, el patrón que representa a todo $\mathbb{N}^*$).

9.3.2. Uso del lema de Dickson para probar terminación

La adaptación del lema de Dickson para la prueba del algoritmo de Buchberger se encuentra en `buchberger.lisp`.

En nuestro caso la medida correspondiente recibiría dos parámetros que corresponderían a lista inicial de términos, LT, y al número de variables de esos términos, k. De esta forma, en lugar de la llamada a la función `(initial-segment-f n)` se emplearía LT, y en lugar de la constante `(k)`, la variable k.

```
(defun medida-terminos (LT k)
  (map-e0-ord-<-fn-e0-ord
   (pattern-list-measure
    (pattern-list-tuple-list LT (list (initial-pattern k))))))
```

Además se añade la llamada a la función `map-e0-ord-<-fn-e0-ord`. Esta función devuelve un ϵ_0-ordinal asociado a un multiconjunto de ordinales, tal y como se describe en [63]. Por tanto, el resultado de la función `medida-terminos` es un ordinal. Esto nos permite trabajar directamente con la relación de orden `e0-ord-<` en vez de con su extensión a multiconjuntos.

Con todo esto, estamos en disposición de definir una nueva medida a la que llamaremos `medida-Buchbeger` y demostrar la terminación del algoritmo de Buchberger.

```
(verify-termination Buchberger-aux-k
  (declare (xargs :measure (medida-Buchberger F C))))
```

Si nos fijamos en la segunda llamada recursiva del algoritmo, se ve que el polinomio que se añade a la base, h, no es cero y es irreducible por F. En particular esto significa que su término principal no es divisible por ninguno de los términos principales de los polinomios de F. Con lo cual, el primer parámetro decrece utilizando la medida proporciona en la demostración del lema de Dickson, como se acaba de ver.

Como en la primera llamada recursiva el tamaño del segundo parámetro decrece en su longitud, un producto lexicográfico nos proporciona la terminación de la función. El ϵ_0-ordinal resultante, que permite comparar lexicográficamente ambas medidas, viene dado por la siguiente expresión:

$$w^{\alpha} + len(C)$$

donde

- α es

  `incrementa-entero(medida-k-terminos(terminos-lideres(F)))`

- `len` devuelve la longitud de la lista C.

- `incrementa-entero` incrementa su parámetro en uno si el parámetro es un entero. Esto es necesario, por cuestiones técnicas, para poder hacer el producto lexicográfico.

- `terminos-lideres` recibe una lista de polinomios y devuelve una lista con los términos principales de cada uno de los polinomios de la lista.

- `medida-k-terminos` recibe una lista de términos de k variables y devuelve el ϵ_0-ordinal asociado. Este ordinal se obtiene mediante la aplicación de la función `medida-terminos` fijando el número de variables a k, como se puede ver a continuación.

La implementación en ACL2 de la medida es la siguiente.

```
(defun<k> medida-Buchberger (F C)
  (if (and (k-polinomiosp F)
           (pares-k-polinomiosp C))
      (cons (incrementa-entero
             (medida-k-terminos (terminos-lideres F)))
            (len C))
    0))
```

A continuación, se presentan las definiciones del resto de las funciones que emplea.

```
(defun incrementa-entero (x)
  (if (integerp x)
      (1+ x)
    x))

(defun terminos-lideres (F)
  (cond ((endp F)
         nil)
        ((equal (first F) (nulo))
         (terminos-lideres (rest F)))
        (t
         (cons (termino (primero (first F)))
               (terminos-lideres (rest F))))))

(defmacro medida-k-terminos (LT)
  `(medida-terminos ,LT k))
```

Ahora es necesario comprobar que se satisfacen las condiciones necesarias para la aplicación de la medida.

En primer lugar hemos de demostrar una propiedad sobre los polinomios que se están añadiendo a la base en cada llamada recursiva. Hay que demostrar que el término principal del polinomio que se está añadiendo a la base no

es divisible por ninguno de los términos principales de los polinomios que ya están en la base.

Claramente, esto es cierto, ya que el polinomio que se está añadiendo es un polinomio irreducible con respecto a la base. Para esto se ha tenido que definir este concepto de divisibilidad de una lista de términos respecto de otro término dado.

```
(defun divisibilidad (LT te)
  (cond ((endp LT)
         nil)
        (t
         (or (dividep (first LT) te)
             (divisibilidad (rest LT) te)))))

(defthm |~div(terminos-lideres(F), tp(red*(p, F)))|
  (implies (and (k-polinomiosp F)
                (k-polinomiop (red-F* p F))
                (not (equal (red-F* p F) (nulo))))
           (not (divisibilidad (terminos-lideres F)
                               (termino (primero (red-F* p F)))))))
```

El siguiente teorema establece la conexión con la función `get-tuple-<`, que se emplea en la demostración del lema de Dickson, estableciendo una de las hipótesis exigidas para utilizar la medida propuesta.

```
(defthm |~div(LT, te) => ~get-tuple-<(LT, te)|
  (implies (and (k-terminop te)
                (k-terminosp LT)
                (not (divisibilidad LT te)))
           (not (get-tuple-< LT te))))
```

En concreto, la función `get-tuple-<` es equivalente en este contexto a la función `divisibilidad`.

```
(defun get-tuple-< (T-lst TT)
  (cond ((endp T-lst)
         nil)
        (t
         (or (tuple-< (car T-lst) TT)
             (get-tuple-< (cdr T-lst) TT)))))
```

Nótese que las funciones `get-tuple-<` y `divisibilidad` no son exactamente iguales ya que `tuple-<` y `dividep` son sutilmente diferentes.

Por otro lado, la función `k-terminosp` reconoce listas de términos uniformes de k variables.

```
(defun<k> k-terminosp (L)
  (cond ((atom L)
         (equal L nil))
        ((not (k-terminop (first L)))
         nil)
        (t
         (k-terminosp (rest L)))))
```

Nótese que la formalización en la que nos apoyamos para definir nuestra medida trabaja con secuencias de términos *uniformes*, esto es, secuencias de términos con el mismo número de variables. Esto ha ocasionado que sea necesario introducir el concepto de polinomio uniforme y demostrar que cada una de las operaciones sobre polinomios preservan la uniformidad (véase el capítulo 4).

9.4. Corrección parcial

Una vez demostrado que nuestra implementación en ACL2 del algoritmo de Buchberger siempre termina su ejecución, nos disponemos a razonar sobre su corrección parcial. El resultado principal que debemos alcanzar es la garantía de que siempre se obtiene una base de Gröbner del ideal generado por la base inicial.

Por el camino irán apareciendo muchos resultados interesantes que organizaremos en torno a un plan de prueba detallado. Esto nos conducirá hasta el resultado final.

9.4.1. Esquema general

Varias de las ideas necesarias para llevar a cabo la demostración de corrección del algoritmo de Buchberger aparecen en [1, 26, 102], pero dado que una demostración completamente formal requiere un nivel de detalle mucho mayor del que aparece en un texto estándar, hemos establecido una serie de etapas

que nos conducirán a la prueba final. Estas etapas se presentan en el siguiente esquema. En primer lugar, se expondrán las propiedades fundamentales ya demostradas en capítulos anteriores.

1. Se ha demostrado que la función fn_F calcula una forma normal, es decir, $p \to_F^* fn_F(p)$ y $fn_F(p)$ es irreducible (teorema 7.29).

2. Se ha demostrado que $fn_F(p) = red_F^*(p)$ (teorema 7.40), y por lo tanto que $p \to_F^* red_F^*(p)$ (teorema 7.41), donde red_F^* es la función de reducción que se utiliza en el algoritmo de Buchberger.

3. Se ha demostrado que $\Phi(F)$ implica que la relación de reducción es localmente confluente (teorema 8.12).

$$\Phi(F) \implies \forall p, q, r\ (r \to_F p \wedge r \to_F q \implies p \downarrow_F^* q)$$

4. Se ha demostrado que la clausura de equivalencia de la reducción polinómica inducida por un conjunto de polinomios que verifica la propiedad Φ se puede decidir comprobando la igualdad de formas normales (teorema 8.13).

$$\Phi(F) \implies (p \leftrightarrow_F^* q \iff fn_F(p) = fn_F(q))$$

5. Se ha demostrado que cualquier base (formada por un conjunto de polinomios) que satisfaga la propiedad Φ es una base de Gröbner (teorema 8.14).

$$\Phi(F) \implies (p \in \langle F \rangle \iff p \to_F^* 0)$$

A continuación se exponen las etapas que aún restan para obtener finalmente la demostración de corrección parcial del algoritmo:

1. Se demostrará que se cumple $\Phi(Buchberger(F))$.

2. Se demostrará la estabilidad del ideal respecto al algoritmo de Buchberger.

$$p \in \langle F \rangle \iff p \in \langle Buchberger(F) \rangle$$

3. Se demostrará que la base devuelta por el algoritmo de Buchberger es una base de Gröbner. Sea $G = Buchberger(F)$, entonces

$$p \in \langle G \rangle \iff p \to_G^* 0$$

   Esto se llevará a cabo utilizando el resultado anterior y mediante instanciación funcional del teorema del punto 5 de la enumeración anterior.

4. Se demostrará, utilizando los puntos anteriores 2 y 3, que la base devuelta por el algoritmo de Buchberger es una base de Gröbner de la base original. Es decir, si $G = Buchberger(F)$, entonces

$$p \in \langle F \rangle \iff p \to_G^* 0$$

Los teoremas correspondientes en ACL2, que demuestran la corrección parcial del algoritmo de Buchberger, son los siguientes. Estos dos teoremas son los resultados principales de la memoria.

```
(defthm |G = Buchberger(F) & p en <F> => p ->*G 0|
  (let* ((G (Buchberger F))
         (prueba (prueba-forma-normal p G)))
    (implies (and (naturalp k)
                  (k-polinomiop p)
                  (k-polinomiosp F)
                  (en-ideal p F))
             (->* p (nulo) prueba G))))

(defthm |G = Buchberger(F) & p ->*G 0 => p en <F>|
  (let ((G (Buchberger F)))
    (implies (and (naturalp k)
                  (k-polinomiop p)
                  (k-polinomiosp F)
                  (->* p (nulo) prueba G))
             (en-ideal p F))))
```

5. Se verificará y construirá un procedimiento de decisión para el problema de la pertenencia al ideal, ya que, si $G = Buchberger(F)$ se tiene que

$$p \in \langle F \rangle \iff red_G^*(p) = 0$$

Es decir, no nos limitamos a demostrar que el algoritmo de Buchberger devuelve una base de Gröbner sino que proporcionamos un procedimiento de decisión verificado para el problema de la pertenencia al ideal.

El procedimiento en ACL2 se expresaría de la siguiente forma.

```
(defun<k> decide (p F)
  (equal (red-F* p (buchberger F))
         (nulo)))
```

El teorema ACL2 que demuestra la corrección de ese procedimiento es el siguiente.

```
(defthm |p en <F> <=> decide(p, F)|
    (implies (and (naturalp k)
                  (k-polinomiop p)
                  (k-polinomiosp F))
             (iff (en-ideal p F) (decide p F))))
```

Nótese la relevancia de este resultado: `en-ideal` no es ejecutable y está definida como una función de Skolem. Por el contrario, `decide` es ejecutable.

La formalización de todas las distintas etapas se encuentra en el fichero `bases-groebner.lisp`.

9.4.2. Satisfacción de la propiedad Φ

Teorema 9.3. *Sea F una secuencia finita de polinomios. Entonces, todos los s-polinomios formados a partir de los polinomios de la base construida por el algoritmo de Buchberger a partir de F se reducen a 0. Es decir,*

$$\Phi(Buchberger(F))$$

donde:

$$\Phi(G) \equiv \forall p, q \in G \; s\text{-}polinomio(p, q) \rightarrow_G^* 0$$

Demostración. Sea $G = Buchberger(F)$.

Si $p, q \in G$ entonces en algún paso j del algoritmo uno de los s-polinomios $s\text{-}polinomio(p, q)$ o $s\text{-}polinomio(q, p)$ fue procesado. Supongamos, sin pérdida de generalidad, que fue $s\text{-}polinomio(p, q)$.

Si $s\text{-}polinomio(p, q) \rightarrow_{G_j}^* 0$, entonces $s\text{-}polinomio(p, q) \rightarrow_G^* 0$ ya que $G_j \subseteq G$.

En caso contrario, $s\text{-}polinomio(p, q) \rightarrow_{G_j}^* h$ y $h \in G_{j+1}$. Entonces, como es evidente que h se puede utilizar para reducir h a 0 en un paso, esto implica $s\text{-}polinomio(p, q) \rightarrow_{G_{j+1}}^* 0$ y, por tanto, $s\text{-}polinomio(p, q) \rightarrow_G^* 0$. □

El teorema ACL2 correspondiente se expresa a continuación. El principal problema está en expresar en ACL2 el hecho de que en algún paso j del algoritmo uno de los s-polinomios, $s\text{-}polinomio(p, q)$ o $s\text{-}polinomio(q, p)$, fue procesado.

```
(defthm |Phi(Buchberger(F))|
  (let* ((G (Buchberger F))
         (prueba (prueba-s-polinomio p q F)))
    (implies (and (k-polinomiosp F) (naturalp k)
                  (en p G) (en q G))
             (->* (s-polinomio p q) (nulo) prueba G))))
```

Nótese que la demostración ACL2 es diferente a la prueba matemática presentada. Por tanto, lo que viene a continuación (y en particular todos los lemas) están encaminados a mostrar con detalle la prueba ACL2.

Nuevamente como en todos los casos (aunque aquí es diferente en cierto sentido, ya que la prueba se obtiene a partir de los pasos seguidos por el algoritmo), debemos en primer lugar definir una función que construya la secuencia de pasos de reducción que hay que hacer para que $s\text{-}polinomio(p,q) \rightarrow_F^* 0$. Esta función se denomina `prueba-s-polinomio` y viene dada por la siguiente definición.

```
(defun<k> prueba-s-polinomio (p q F)
  (let* ((prueba
          (prueba-Buchberger F (parejas-iniciales F) nil nil)))
    (prueba-par p q (pares prueba) (pruebas prueba))))
```

La función `prueba-Buchberger` construye una prueba que demuestra la reducción a 0 de los *s-polinomios*. La demostración de la parada de esta función es igual que la del algoritmo de Buchberger, por tanto se utiliza la misma medida (omitida aquí) que se construyó para demostrar la parada de Buchberger, así como los mismos lemas necesarios.

```
(defun<k> prueba-Buchberger (F C Cp pruebas)
  (if (and (naturalp k)
           (k-polinomiosp F)
           (pares-k-polinomiosp C))
      (if (endp C)
          (list F Cp pruebas)
        (let* ((p (first (first C)))
               (q (second (first C)))
               (h (red-F* (s-polinomio p q) F))
               (prueba (prueba-forma-normal (s-polinomio p q) F)))
          (if (equal h (nulo))
              (prueba-Buchberger F
```

```
                                  (rest C)
                                  (cons (first C) Cp)
                                  (cons prueba pruebas))
            (prueba-Buchberger (cons h F)
                               (append (parejas h F) (rest C))
                               (cons (first C) Cp)
                               (cons (prueba-h prueba h) pruebas)))))
    (list F Cp pruebas)))
```

donde `prueba-h` se define de la siguiente forma,

```
(defun prueba-h (prueba h)
  (let* ((c (polinomio
             (RAC-MON::- (RAC-MON::/ (first h) (first h)))))
         (paso (list (make-r-step
                      :direct t :elt1 h :elt2 (nulo)
                      :operator (operador (first h) c h)))))
    (if (endp prueba)
        paso
      (append prueba paso))))
```

La función `prueba-Buchberger` sigue el mismo esquema que `Buchberger-aux`. Recibe la base inicial, el conjunto de las parejas iniciales de esa base, el conjunto de las parejas procesadas (inicialmente, vacío) y las pruebas que demuestran que los s-polinomios de las parejas procesadas se reducen a 0 respecto de la base que se va calculando (inicialmente, también vacío). Su resultado consiste en una tupla de tres componentes (al primer componente lo identificaremos como *base*, al segundo como *pares* y al tercero *pruebas*):

- La base de Gröbner correspondiente a la base inicial (se demostrará que la base devuelta coincide con la que devuelve la función `Buchberger`).
- El conjunto de parejas procesadas. Al final, tendrá que tener todas las parejas posibles que se puedan formar con la base devuelta (excepto las de un polinomio consigo mismo, y salvo el orden).
- El conjunto de pruebas correspondientes al conjunto de parejas procesadas y que demuestran que los s-polinomios de esas parejas se reducen a 0 respecto de la base devuelta.

A continuación, se presenta la función `prueba-par`. Esta función recibe un par de polinomios, p y q, una lista de pares, C_p y una lista de pruebas, *prueba*,

correspondiente a la lista de pares. Devuelve la prueba que le corresponde al par, (p, q).

```
(defun prueba-par (p q Cp prueba)
  (cond ((endp Cp)
         nil)
        ((equal (list p q) (first Cp))
         (first prueba))
        ((equal (list q p) (first Cp))
         (prueba-negada (first prueba)))
        (t
         (prueba-par p q (rest Cp) (rest prueba)))))
```

Puede ocurrir que el par que se encuentre en la lista de pares sea el inverso y que, por tanto, el s-polinomio esté negado. En ese caso, debe obtener la prueba opuesta.

La función `prueba-negada` es la encargada de construir la opuesta de una prueba. Para ello se define la función `paso-negado` que devuelve el opuesto del paso de prueba que recibe como entrada.

```
(defun prueba-negada (prueba)
  (cond ((endp prueba)
         nil)
        (t (cons (paso-negado (first prueba))
                 (prueba-negada (rest prueba))))))

(defun paso-negado (paso)
  (let* ((nm (RAC-MON::- (o-monomio (operator paso))))
         (np (o-polinomio (operator paso)))
         (nc (polinomio (RAC-MON::- (RAC-MON::/ nm (first np))))))
    (make-r-step :direct (direct paso)
                 :elt1 (- (elt1 paso))
                 :elt2 (- (elt2 paso))
                 :operator (operador nm nc np))))
```

Antes de llevar a cabo la demostración del teorema 9.3 en ACL2 es necesario probar los siguientes lemas.

Lema 9.4 (monotonía de la clausura de equivalencia). *Sean p un polinomio y, F y G secuencias finitas de polinomios. Si $F \subseteq G$ entonces*

$$p \leftrightarrow_F^* q \implies p \leftrightarrow_G^* q$$

En ACL2,

```
(defthm |F suf G & p <->F* q => p <->G* q|
  (implies (and (subsetp F G)
                (k-polinomiosp G)
                (k-polinomiop p)
                (k-polinomiop q)
                (<->* p q prueba F))
           (<->* p q prueba G)))
```

Demostración. Por inducción sobre la longitud de la prueba.

Caso base:

Si la longitud de la prueba es 0, es porque $p = q$, resultando $p \leftrightarrow_G^* q$ trivialmente.

Paso de inducción:

Si la longitud de la prueba es mayor que 0 podemos suponer que existe $f_i \in F$ tal que:

$$p \leftrightarrow_{f_i} r \leftrightarrow_F^* q$$

Por hipótesis de inducción:

$$r \leftrightarrow_F^* q \implies r \leftrightarrow_G^* q$$

De esta forma, sólo resta demostrar que:

$$p \leftrightarrow_{f_i} r \implies p \leftrightarrow_G r$$

Como $p \leftrightarrow_{f_i} r$, $f_i \in F$ y $F \subseteq G$ entonces $f_i \in G$ y, por tanto, $p \leftrightarrow_G r$.

□

Lema 9.5. *Sean p, h polinomios y F una secuencia finita de polinomios.*

$$p \leftrightarrow_F^* h \implies p \leftrightarrow_{h::F}^* 0$$

En ACL2,

```
(defthm |h != 0 & p <->F* h => p <->{h,F}* 0|
  (implies (and (not (equal h (nulo)))
                (k-polinomiosp F)
                (k-polinomiop p)
                (k-polinomiop h)
                (<->* p h prueba F))
           (<->* p (nulo) (prueba-h prueba h) (cons h F))))
```

Demostración. Por inducción sobre la longitud de la prueba, $p \leftrightarrow_F^* h$.

Caso base:

Si la longitud de la prueba es 0, es porque $p = h$, resultando $p \leftrightarrow_{h::F}^* 0$ ya que $h \leftrightarrow_{h::F} 0$, por definición.

Paso de inducción:

Si la longitud de la prueba es mayor que 0 podemos suponer que existe $f_i \in F$ tal que:

$$p \leftrightarrow_{f_i} r \leftrightarrow_F^* h$$

Por hipótesis de inducción:

$$r \leftrightarrow_F^* h \implies r \leftrightarrow_{h::F}^* 0$$

De esta forma, sólo resta demostrar que:

$$p \leftrightarrow_F r \implies p \leftrightarrow_{h::F}^* r$$

Como $p \leftrightarrow_{f_i} r$, $f_i \in F$ y $f_i \in h :: F$ y, por tanto, $p \leftrightarrow_{h::F}^* r$.

□

Lema 9.6. *Sea p un polinomio y F una secuencia finita de polinomios.*

$$p \rightarrow_F^* 0 \implies -p \rightarrow_F^* 0$$

En ACL2,

```
(defthm |p ->F* 0 => -p ->F* 0|
  (implies (->* p (nulo) prueba F)
           (->* (- p) (nulo) (prueba-negada prueba) F)))
```

Demostración. Por inducción sobre la longitud de la prueba.

Caso base:

Si la longitud de la prueba es 0, es porque $p = 0$, resultando $-p \rightarrow_F^* 0$.

Paso de inducción:

Si la longitud de la prueba es mayor que 0 podemos suponer que existe $f_i \in F$ tal que:

$$p \rightarrow_{f_i} r \rightarrow_F^* 0$$

Por hipótesis de inducción:

$$r \rightarrow_F^* 0 \implies -r \rightarrow_F^* 0$$

De esta forma, sólo resta demostrar que:

$$p \rightarrow_F r \implies -p \rightarrow_F -r$$

Sea m el monomio de p por el que se produce la reducción. Como $p \rightarrow_{f_i} r$ entonces $r = p - m/mp(f_i) \cdot f_i$.

Como $m \in p$ entonces $-m \in -p$ y, por tanto, el polinomio resultante de reducir $-p$ por f_i en el monomio $-m$ será $-p + m/mp(f_i) \cdot f_i$ que es justamente $-r$. Por tanto, $-p \rightarrow_F -r$.

□

Para la demostración en ACL2 se comprueba, primero, que si una prueba es descendente (es decir, está compuesta solamente por pasos directos de prueba) entonces su negación también.

```
(defthm |descendentep(prueba-negada(prueba))|
  (implies (descendentep prueba)
           (descendentep (prueba-negada prueba))))
```

Y a continuación se establece que si un paso es válido su negado también, para obtener la demostración del lema original.

```
(defthm |paso-valido(paso) => paso-valido(paso-negado(paso))|
   (implies (paso-valido paso F)
            (paso-valido (paso-negado paso) F)))
```

Además, nótese que la función *Buchberger* no genera todos los posibles s-polinomios sino sólo un conjunto reducido. Los dos lemas siguientes aseguran que la reducción a cero del conjunto reducido implican la reducción del conjunto completo.

Lema 9.7. *Sea p un polinomio.*

$$s\text{-}polinomio(p, p) = 0$$

En ACL2,

```
(defthm |s-polinomio(p, p) = 0|
  (implies (polinomiop p)
           (equal (s-polinomio p p) (nulo))))
```

Demostración. Inmediata por propiedades de polinomios. □

Lema 9.8. *Sean p y q polinomios.*

$$s\text{-}polinomio(p, q) = -s\text{-}polinomio(q, p)$$

En ACL2,

```
(defthm |s-polinomio(p, q) = - s-polinomio(q, p)|
  (implies (and (polinomiop p) (polinomiop q))
           (equal (- (s-polinomio q p)) (s-polinomio p q))))
```

Demostración. Inmediata por propiedades de polinomios. □

Falta ahora, probar que la base devuelta por la función `prueba-Buchberger` coincide con la que devuelve la función `Buchberger` (lema 9.10).

Ya que la función *Buchbeger* no es recursiva, no sugiere esquema de inductivo alguno, y, por tanto, no es posible realizar una prueba inductiva del lema 9.10 sobre la función *Buchberger*. Esto es muy común en este tipo de demostraciones donde se tiene una función auxiliar, en nuestro caso *Buchberger-aux*, y una función principal, *Buchberger*, que delega en ella. La función auxiliar corresponde en realidad a una generalización de la función principal, con parámetros adicionales. Cuando los parámetros toman los valores apropiados, la función auxiliar realiza el trabajo encomendado a la principal. Por tanto,

las propiedades de la principal son casos particulares de propiedades más generales de la auxiliar.

Por esta razón, demostramos primero el siguiente teorema, más general, obteniendo el resultado deseado a partir de él.

Lema 9.9. *Sean F una secuencia finita de polinomios y C, C_p secuencias finitas de pares de polinomios. Entonces,*

$$\mathit{Buchberger\text{-}aux}(F, C) = \mathit{base}(\mathit{prueba\text{-}Buchberger}(F, C, C_p, \mathit{pruebas}))$$

En ACL2,

```
(defthm |Buchberger-aux(F,C) = prueba-Buchberger(F,C,Cp,pruebas)|
  (equal (Buchberger-aux F C)
         (base (prueba-Buchberger F C Cp pruebas)))))
```

Demostración. Por inducción, siguiendo el esquema sugerido por la función *Buchberger-aux*.

Caso base: $C = []$.

Inmediato, ya que $F = F$.

Paso de inducción: $C \neq []$.

Sean $(p, q) = \mathit{primero}(C)$ y $h = \mathit{red}^*_F(\mathit{s\text{-}polinomio}(p, q))$.

- Si $h = 0$, por hipótesis de inducción se tiene que

  $$\mathit{Buchberger\text{-}aux}(F, \mathit{resto}(C)) = \mathit{base}(\mathit{prueba\text{-}Buchberger}(F, \mathit{resto}(C), (p, q) :: C_p, p_s :: \mathit{pruebas}))$$

  donde $p_s = \mathit{prueba\text{-}forma\text{-}normal}(\mathit{s\text{-}polinomio}(p, q), F)$.
  Por tanto, como en este caso, por definición $\mathit{Buchberger\text{-}aux}(F, C) = \mathit{Buchberger\text{-}aux}(F, \mathit{resto}(C))$, el resultado se sigue directamente.
- Si $h \neq 0$, por hipótesis de inducción se tiene que

  $$\mathit{Buchberger\text{-}aux}(h :: F, \mathit{parejas}(h, F) +\!\!+ \mathit{resto}(C))) = \mathit{base}(\mathit{prueba\text{-}Buchberger}(h :: F, \mathit{parejas}(h, F) +\!\!+ \mathit{resto}(C)), (p, q) :: C_p, p_h :: \mathit{pruebas}))$$

donde $p_h = \mathit{prueba\text{-}h}(\mathit{prueba\text{-}forma\text{-}normal}(\mathit{s\text{-}polinomio}(p, q), F), h)$.
El resultado se sigue de la definición de la función.

□

Lema 9.10. *Sea F una secuencia finita de polinomios. Entonces,*

$$\mathit{Buchberger}(F) = \mathit{base}(\mathit{prueba\text{-}Buchberger}(F, \mathit{parejas\text{-}iniciales}(F), \mathit{nil}, \mathit{nil}))$$

En ACL2,

```
(defthm |Buchberger(F) = prueba-Buchberger(F, C, nil, nil)|
  (let ((C (parejas-iniciales F)))
    (equal (Buchberger F)
           (base (prueba-Buchberger F C nil nil)))))
```

Demostración. Inmediata utilizando el lema 9.9. □

Además, también hay que demostrar que el conjunto de parejas procesadas que devuelve `prueba-Buchberger` tiene todas las parejas posibles que se puedan formar con la base devuelta (lema 9.12). Este lema se demostrará utilizando el siguiente.

Lema 9.11. *Sean c un par de polinomios, F una secuencia finita de polinomios y, C, C_p secuencias de pares de polinomios.*

Si $c \in \mathit{parejas\text{-}iniciales}(\mathit{base}(\mathit{prueba\text{-}Buchberger}(F, C, C_p, \mathit{pruebas}))$ y, además se cumple que $c \in \mathit{parejas\text{-}iniciales}(F) \implies c \in C$, entonces

$$c \in \mathit{pares}(\mathit{prueba\text{-}Buchberger}(F, C, C_p, \mathit{pruebas}))$$

```
(defthm
|par en parejas-iniciales(base(prueba)) => par en pares(prueba)|
  (let* ((prueba (prueba-Buchberger F C Cp pruebas)))
    (implies (and (naturalp k)
                  (k-polinomiosp F)
                  (pares-k-polinomiosp C)
                  (implies (en par (parejas-iniciales F))
                           (en par C))
                  (en par (parejas-iniciales (base prueba))))
             (en par (pares prueba)))))
```

Demostración. Por inducción, siguiendo el esquema sugerido por la función *prueba-Buchberger*.

Caso base: $C = []$.

Inmediato, ya que $c \in \textit{parejas-iniciales}(F) \implies c \in C$.

Paso de inducción: $C \neq []$.

Sean $(p, q) = \textit{primero}(C)$ y $h = \textit{red}^*_F(\textit{s-polinomio}(p, q))$.

- Si $h = 0$, por hipótesis de inducción se tiene que

  $$c \in \textit{pares}(\textit{prueba-Buchberger}(F, \textit{resto}(C), (p, q) :: C_p, p_s :: \textit{pruebas}))$$

  donde $p_s = \textit{prueba-forma-normal}(\textit{s-polinomio}(p, q), F)$.
  El resultado se sigue por definición y propiedades de polinomios.
- Si $h \neq 0$, por hipótesis de inducción se tiene que
  $c \in \textit{pares}(\textit{prueba-Buchberger}(h :: F, \textit{parejas}(h, F) +\!\!+ \textit{resto}(C)),$
  $(p, q) :: C_p, p_h :: \textit{pruebas}))$

  donde $p_h = \textit{prueba-h}(\textit{prueba-forma-normal}(\textit{s-polinomio}(p, q), F), h)$.
  El resultado se sigue de la definición de la función y propiedades de polinomios.

□

Lema 9.12. *Sean c un par de polinomios y F una secuencia finita de polinomios. Si $c \in \textit{parejas-iniciales}(\textit{Buchberger}(F))$, entonces*

$$c \in \textit{pares}(\textit{prueba-Buchberger}(F, \textit{parejas-iniciales}(F), \textit{nil}, \textit{nil}))$$

```
(defthm
|(par en parejas-iniciales(Buchberger(F)) => par en pares(prueba)|
  (let* ((C (parejas-iniciales F))
         (prueba (prueba-Buchberger F C nil nil)))
    (implies (and (naturalp k)
                  (k-polinomiosp F)
                  (en par (parejas-iniciales (Buchberger F))))
             (en par (pares prueba)))))
```

Demostración. Inmediata por los lemas 9.10 y 9.11. □

A continuación, se tiene que probar que realmente los s-polinomios obtenidos de la lista de parejas procesadas que devuelve la función `prueba-Buchberger` como segundo parámetro se reducen a 0, utilizando las pruebas correspondientes del conjunto de pruebas devuelto en tercer lugar, respecto de la base que devuelve la función como primer parámetro (lema 9.14). Todo esto se hace respecto a un conjunto inicial de parejas procesadas y pruebas (ambos vacíos), y con el conjunto de parejas iniciales de la base (que es lo que justamente se tiene en la función `Buchberger`).

Su demostración se obtiene a partir del lema siguiente que es una generalización que en cierta manera expresa un invariante en todo el proceso. En este caso, el conjunto inicial de parejas procesadas y pruebas no tiene que ser vacío pero tiene que satisfacer la propiedad Φ. Es decir, como hipótesis se tiene que cumplir Φ para lo que ya hubiera en el conjunto de parejas procesadas, `Cp`.

Lema 9.13. *Sean F una secuencia finita de polinomios, C y C_p secuencias finitas de pares de polinomios, $G = base(prueba\text{-}Buchberger(F, C, C_p, pruebas))$ y donde los pares de C_p satisfacen la propiedad Φ. Entonces para todo $(p, q) \in pares(prueba\text{-}Buchberger(F, C, C_p, pruebas))$*

$$s\text{-}polinomio(p, q) \rightarrow_G^* 0$$

Demostración. Por inducción siguiendo el esquema sugerido por la función prueba-Buchberger y usando los lemas 9.4 y 9.5, que $p \rightarrow_F^* red_F^*(p)$ y que $red\text{-}F^*(p, F) = 0 \implies p \rightarrow_F^* 0$. □

En ACL2,

```
(defthm |Phi(prueba-Buchberger(F, C))|
  (let* ((prueba (prueba-Buchberger F C Cp pruebas)))
    (implies (and (Phi Cp pruebas F)
                  (k-polinomiosp F))
             (Phi (pares prueba) (pruebas prueba) (base prueba)))))
```

donde la función `Phi` nos va a servir para expresar en ACL2 la propiedad Φ. Esta función recibe tres parámetros, todas las parejas de polinomios cuyos s-polinomios se tienen que reducir a 0, las pruebas correspondientes y la base, y comprueba si realmente los s-polinomios se reducen a 0 con las pruebas correspondientes respecto de la base.

```
(defun<k> Phi (parejas pruebas F)
  (cond ((or (endp parejas) (endp pruebas))
         t)
        ((let* ((par (first parejas))
                (p (first par))
                (q (second par)))
           (<->* (s-polinomio p q) (nulo) (first pruebas) F))
         (Phi (rest parejas) (rest pruebas) F))
        (t
         nil)))
```

Lema 9.14. *Sea F una secuencia de polinomios, $C = parejas\text{-}iniciales(F)$ y $G = base(prueba\text{-}Buchberger(F, C, nil, nil))$. Entonces para todo $(p, q) \in pares(prueba\text{-}Buchberger(F, C, nil, nil))$*

$$s\text{-}polinomio(p, q) \rightarrow_G^* 0$$

Demostración. Inmediata por el lema 9.13. □

En ACL2,

```
(defthm |Phi(prueba-Buchberger(F, parejas-iniciales(F)))|
  (let* ((prueba
          (prueba-Buchberger F (parejas-iniciales F) nil nil)))
    (implies (k-polinomiosp F)
             (Phi (pares prueba) (pruebas prueba) (base prueba)))))
```

Por último, se demuestra el lema que relaciona `prueba-par` con la función `Phi`.

Lema 9.15. *Sean p, q polinomios, F una secuencia finita de polinomios y C_p una secuencia finita de pares de polinomios. Si $Phi(C_p, pruebas, F)$ y, $(p, q) \in C_p$ o $(q, p) \in C_p$, entonces $s\text{-}polinomio(p, q) \rightarrow_F^* 0$.*

```
(defthm |(p, q) en Cp & Phi(Cp, pruebas, F) => s-pol(p, q) ->F* 0|
  (implies (and (polinomiop p) (polinomiop q)
                (equal (len Cp) (len pruebas))
                (or (en (list p q) Cp)
                    (en (list q p) Cp))
                (Phi Cp pruebas F)
                (descendentesp pruebas))
           (->* (s-polinomio p q)
```

```
(nulo)
(prueba-par p q Cp pruebas)
F)))
```

Demostración. Por inducción, siguiendo el esquema sugerido por la función *prueba-par*.

Caso base 1: $C_p = []$

Inmediato, por definición de $\in$.

Caso base 2: $C_p \neq [] \wedge (p,q) = primero(C_p)$.

Inmediato, por definición de las funciones.

Caso base 3: $C_p \neq [] \wedge (p,q) \neq primero(C_p) \wedge (q,p) = primero(C_p)$.

Por definición y por los teoremas 9.6 y 9.8.

Paso de inducción: $C_p \neq []$.

Por hipótesis de inducción:

$Phi(resto(C_p), resto(pruebas), F) \wedge ((p,q) \vee (q,p)) \in resto(C_p) \implies$

$s\text{-}polinomio(p,q) \rightarrow_F^* 0$

El resultado se sigue por definición de las funciones.

□

Lema 9.16. *Sean F una secuencia finita de polinomios y $G = Buchberger(F)$. Entonces para todo par (p,q) tal que $(p = q \vee (p,q) \in parejas\text{-}iniciales(G) \vee (q,p) \in parejas\text{-}iniciales(G))$ se tiene que*

$$s\text{-}polinomio(p,q) \rightarrow_G^* 0$$

Demostración. Por los lemas 9.7, 9.10, 9.12, 9.14 y 9.15. □

En ACL2,

```
(defthm |Phi(prueba-Buchberger(F))|
  (let ((G (Buchberger F)))
    (implies (and (or (en (list p q) (parejas-iniciales G))
                      (en (list q p) (parejas-iniciales G))
                      (equal p q))
```

```
            (k-polinomiosp F) (naturalp k)
            (polinomiop p) (polinomiop q))
      (<->* (s-polinomio p q) (nulo)
            (prueba-s-polinomio p q F) G))))
```

Finalmente, la demostración ACL2 del teorema 9.3 se lleva a cabo aplicando el lema anterior y que la función `prueba-s-polinomio` devuelve una prueba descendente.

9.4.3. Estabilidad del ideal

Presentamos la formalización realizada en `estabilidad-buchberger.lisp` de que el ideal permanece estable respecto al algoritmo de Buchberger, esto es, que los ideales generados por la base inicial y la final coinciden.

$$p \in \langle F \rangle \iff p \in \langle Buchberger(F) \rangle$$

En ACL2,

```
(defthm |p en <Buchberger(F)> <=> p en <F>|
  (implies (k-polinomiosp F)
           (iff (en-ideal p (Buchberger F)) (en-ideal p F))))
```

Recuérdese que ya se demostró previamente que los ideales son cerrados bajo las operaciones elementales, así como el hecho de que las funciones de reducción y de cálculo de s-polinomios son operaciones internas al ideal. Esto quiere decir que un polinomio pertenece a un ideal si, y sólo si, su reducción también pertenece al ideal, y por otro lado, que el s-polinomio obtenido a partir de dos polinomios de la base, también pertenece al ideal.

Observemos también que el algoritmo manipula la base inicial tratándola como una secuencia finita de polinomios que siempre se extiende por el principio. Esto provoca que la base inicial sea un sufijo de la base final, vistas ambas como secuencias. Resulta necesario establecer este hecho, así como las propiedades elementales de la relación «ser sufijo de». Notaremos esta relación por $\sqsubseteq$.

A continuación procederemos a describir las propiedades necesarias para obtener la estabilidad del ideal bajo el algoritmo de Buchberger. Nótese que la función `Buchberger` lo único que hace es una llamada, con los parámetros

adecuados, a la función `Buchberger-aux`, que es la que en realidad hace todo el trabajo. Por tanto, son necesarias propiedades sobre la función auxiliar como es el caso de la siguiente.

Proposición 9.17. *Sea F una secuencia finita de polinomios. Entonces*

$$F \sqsubseteq \textit{Buchberger-aux}(F, C)$$

Demostración. El algoritmo completa la base inicial tratándola como una secuencia finita de polinomios que siempre se extiende insertando polinomios por el principio. Esto provoca que la base inicial sea un sufijo de la final.

Por inducción sobre la estructura de la función *Buchberger-aux*.

Caso base: $C = []$.

Inmediato, ya que $F \sqsubseteq F$.

Paso de inducción: $C \neq []$.

Sean $(p, q) = \textit{primero}(C)$ y $h = \textit{red}^*_F(\textit{s-polinomio}(p, q))$.

- Si $h = 0$, por hipótesis de inducción se tiene que

  $$F \sqsubseteq \textit{Buchberger-aux}(F, \textit{resto}(C))$$

  Por tanto, como en este caso, por definición $\textit{Buchberger-aux}(F, C) = \textit{Buchberger-aux}(F, \textit{resto}(C))$, el resultado se sigue directamente.
- Si $h \neq 0$, por hipótesis de inducción se tiene que

  $$h :: F \sqsubseteq \textit{Buchberger-aux}(h :: F, \textit{parejas}(h, F) +\!\!+ \textit{resto}(C))$$

  El resultado se sigue de la definición de la función, ya que $F \sqsubseteq h :: F$ y $\textit{Buchberger}(F, C) = \textit{Buchberger-aux}(h :: F, \textit{parejas}(h, F) +\!\!+ \textit{resto}(C))$.

□

El teorema en ACL2 se expresa de la siguiente forma.

```
(defthm |F suf Buchberger-aux(F, C)|
  (sufijop F (Buchberger-aux F C)))
```

Donde la función `sufijop` representa $\sqsubseteq$. Su definición en ACL2 es como sigue.

```
(defun sufijop (F G)
  (if (endp G)
      (endp F)
    (or (equal F G) (sufijop F (rest G)))))
```

Algunas de las propiedades elementales de la función anterior se resumen en la figura 9.2. Estas propiedades las emplea automáticamente ACL2 en las demostraciones de varios de los teoremas que presentaremos más adelante. Las citamos sin demostración, ya que se intuyen sin una dificultad especial.

Proposición 9.18. *Sean p un polinomio y, F y G dos secuencias finitas de polinomios.*

$$p \in \langle F \rangle \wedge F \sqsubseteq G \implies p \in \langle G \rangle$$

En ACL2,

```
(defthm |p en <F> & F suf G => p en <G>|
  (implies (and (en-ideal p F) (sufijop F G))
           (en-ideal p G)))
```

Demostración. G es una extensión sufija de F: tiene los mismos elementos de F y, quizás, algunos nuevos. Para los elementos comunes a F y G, basta tomar los mismos testigos de la pertenencia de p a $\langle F \rangle$. Para cada nuevo elemento basta tomar como testigo el polinomio nulo. Con estos testigos se asegura la pertenencia de p a $\langle G \rangle$. □

Lema 9.19. *Sean p y q polinomios, y F una secuencia finita de polinomios.*

$$p \in \langle q :: F \rangle \wedge q \in \langle F \rangle \implies p \in \langle F \rangle$$

En ACL2,

```
(defthm |p en <cons(q, F)> & q en <F> => p en <F>|
  (implies (and (en-ideal p (cons q F)) (en-ideal q F))
           (en-ideal p F)))
```

```
(defthm |F suf F|
  (sufijop F F))

(defthm |F suf G & G suf H => F suf H|
  (implies (and (sufijop F G) (sufijop G H))
           (sufijop F H)))

(defthm |F suf cons(p, F)|
  (sufijop F (cons p F)))

(defthm |F suf G => F suf cons(p, G)|
  (implies (sufijop F G)
           (sufijop F (cons p G))))

(defthm |cons(p, F) suf G => F suf G|
  (implies (sufijop (cons p F) G)
           (sufijop F G)))

(defthm |resto(F) suf F|
  (sufijop (rest F) F))

(defthm |F suf resto(G) => F suf G|
  (implies (sufijop F (rest G))
           (sufijop F G)))

(defthm |F suf G => resto(F) suf G|
  (implies (sufijop F G)
           (sufijop (rest F) G)))
```

Figura 9.2: Otras propiedades elementales de los sufijos

Demostración. Sea $F = \langle f_1, \ldots, f_n \rangle$.

$$\begin{aligned} p \in \langle q :: F \rangle \wedge q \in \langle F \rangle &\implies p = c \cdot q + \sum_{i=1}^{n} c_i f_i \wedge q = \sum_{i=1}^{n} d_i f_i \\ &\implies p = \sum_{i=1}^{n} (c \cdot d_i) f_i + \sum_{i=1}^{n} c_i f_i \\ &\implies p = \sum_{i=1}^{n} (c \cdot d_i + c_i) f_i \\ &\implies p \in \langle F \rangle. \end{aligned}$$

□

A continuación, y abusando del lenguaje, ampliaremos el concepto de pertenencia de un polinomio a un ideal a objetos más complejos como las secuencias de polinomios y de pares de polinomios.

Definición 9.20. Sea P una secuencia finita de polinomios y F una base. Decimos que P pertenece a $\langle F \rangle$, que notamos por $P \in \langle F \rangle$ (extendiéndose la relación de pertenencia), si todos los polinomios de P pertenecen a $\langle F \rangle$.

Definición 9.21. Sea C una secuencia finita de pares de polinomios y F una base. Decimos que C está en $\langle F \rangle$, que notamos por $C \subseteq \langle F \rangle \times \langle F \rangle$, si todos los pares de C pertenecen a $\langle F \rangle \times \langle F \rangle$.

Teorema 9.22. *Sean p un polinomio, F una secuencia finita de polinomios y C una secuencia finita de pares de polinomios tal que $C \subseteq \langle F \rangle \times \langle F \rangle$. Entonces,*

$$p \in \langle F \rangle \iff p \in \langle \textit{Buchberger-aux}(F, C) \rangle$$

En ACL2,

```
(defthm |C en <<F>> => (p en <Buchberger-aux(F, C)> <=> p en <F>)|
  (implies (and (k-polinomiosp F)
                (pares-k-polinomiosp C)
                (pares-en-ideal C F))
           (iff (en-ideal p (Buchberger-aux F C)) (en-ideal p F))))
```

Demostración. Sea $G = \textit{Buchberger-aux}(F, C)$. Dividimos la demostración en las dos implicaciones.

$\Rightarrow$:

$$\begin{aligned} & p \in \langle F \rangle \\ & \quad \Longrightarrow\ p \in \langle F \rangle \wedge F \sqsubseteq G && \text{(prop. 9.17)} \\ & \quad \Longrightarrow\ p \in \langle G \rangle && \text{(prop. 9.18)} \end{aligned}$$

$\Leftarrow$: Supongamos que $p \in \langle G \rangle$ y $C \subseteq \langle F \rangle \times \langle F \rangle$ y probemos que $p \in \langle F \rangle$ por inducción sobre la estructura de la función *Buchberger-aux*.

Caso base: $C = []$.

Inmediato, por la definición de la función.

Paso de inducción: $C \neq []$.

Sean $(p, q) = \mathit{primero}(C)$ y $h = \mathit{red}^*_F(\mathit{s\text{-}polinomio}(p, q))$.

- Si $h = 0$, por hipótesis de inducción se tiene que

  $$\mathit{resto}(C) \subseteq \langle F \rangle \times \langle F \rangle \wedge p \in \mathit{Buchberger\text{-}aux}(F, \mathit{resto}(C)) \implies p \in \langle F \rangle$$

  Como $C \subseteq \langle F \rangle \times \langle F \rangle$, entonces $\mathit{resto}(C) \subseteq \langle F \rangle \times \langle F \rangle$. Además $G = \mathit{Buchberger\text{-}aux}(F, \mathit{resto}(C))$, por definición de la función, luego podemos aplicar la hipótesis de inducción para concluir que $p \in \langle F \rangle$.
- Si $h \neq 0$, por hipótesis de inducción se tiene que

  $$\mathit{parejas}(h, F) +\!\!+ \mathit{resto}(C) \subseteq \langle h :: F \rangle \times \langle h :: F \rangle \wedge$$
  $$p \in \mathit{Buchberger\text{-}aux}(h :: F, \mathit{parejas}(h, F) +\!\!+ \mathit{resto}(C)) \implies p \in \langle h :: F \rangle$$

  Nótese que:

  1. $\mathit{parejas}(h, F) +\!\!+ \mathit{resto}(C) \subseteq \langle h :: F \rangle \times \langle h :: F \rangle$, ya que se verifican $\mathit{parejas}(h, F) \subseteq \langle h :: F \rangle \times \langle h :: F \rangle$ y $\mathit{resto}(C) \subseteq \langle h :: F \rangle \times \langle h :: F \rangle$ (por hipótesis, $C \subseteq \langle F \rangle \times \langle F \rangle$).
  2. $h \in \langle F \rangle$, porque h es el s-polinomio reducido de dos polinomios de la base. Los polinomios de la base pertenecen al ideal, por lo tanto, el s-polinomio de dos polinomios de la base también (por el teorema 8.9) y reducirlo no altera este hecho (véase el teorema 7.44).
  3. $p \in \langle h :: F \rangle$ por hipótesis de inducción.
  4. Por el lema 9.19, como $h \in \langle F \rangle$ y $p \in \langle h :: F \rangle$, se obtiene que $p \in \langle F \rangle$.

□

Teorema 9.23. *El algoritmo de Buchberger no altera el ideal generado.*

Sean p un polinomio y F una secuencia finita de polinomios.

$$p \in \langle F \rangle \iff p \in \langle \mathit{Buchberger}(F) \rangle$$

En ACL2,

```
(defthm |p en <Buchberger(F)> <=> p en <F>|
  (implies (k-polinomiosp F)
           (iff (en-ideal p (Buchberger F)) (en-ideal p F))))
```

Demostración. Sea $G = \mathit{Buchberger}(F)$ y $C = \mathit{parejas\text{-}iniciales}(F)$.

$$\begin{array}{lll}
p \in \langle F \rangle & & \\
\iff p \in \langle F \rangle \wedge C \subseteq \langle F \rangle \times \langle F \rangle & & (C \subseteq \langle F \rangle \times \langle F \rangle) \\
\iff p \in \langle \mathit{Buchberger\text{-}aux}(F, C) \rangle & & (\text{teorema } 9.22) \\
\iff p \in \langle G \rangle & & (\text{def. de } \mathit{Buchberger})
\end{array}$$

□

9.4.4. Relación con las bases de Gröbner

Una vez que se han demostrado las propiedades anteriores, estamos en condiciones de poder demostrar que el algoritmo de Buchberger devuelve una base de Gröbner.

Teorema 9.24. *El algoritmo de Buchberger devuelve una base de Gröbner. Es decir, sea p un polinomio, $F = \{f_1, \ldots, f_k\}$ una secuencia finita de polinomios y $G = \mathit{Buchberger}(F)$ entonces:*

$$p \in \langle G \rangle \iff p \to_G^* 0$$

Demostración. Por el teorema 9.3 se tiene que $\Phi(G)$, entonces aplicando el teorema 8.14 se tiene el resultado. □

El teorema anterior en ACL2 se expresa de la siguiente forma. Recuérdese que *prueba-forma-normal* es la función que construye una prueba que conduce desde un polinomio hasta su forma normal.

```
(defthm |G = Buchberger(F) & p en <G> => p ->*G 0|
  (let* ((G (Buchberger F))
         (prueba (prueba-forma-normal p G)))
    (implies (and (naturalp k)
                  (k-polinomiop p)
                  (k-polinomiosp F)
                  (en-ideal p G))
             (->* p (nulo) prueba G))))

(defthm |G = Buchberger(F) & p ->*G 0 => p en <G>|
  (let ((G (Buchberger F)))
    (implies (and (naturalp k)
                  (k-polinomiop p)
                  (k-polinomiosp F)
                  (<->* p (nulo) prueba G))
             (en-ideal p G))))
```

Estos dos teoremas se demuestran en ACL2 utilizando instanciación funcional de los teoremas ya presentados `|Phi(F) & p en <F> => p ->*F 0|` y `|Phi(F) & p ->*F 0 => p en <F>|` y, porque se ha probado que el algoritmo satisface la propiedad Φ, `|Phi(Buchberger(F))|`.

A partir del resultado anterior y el del teorema 9.23 se puede demostrar que la base devuelta por el algoritmo de Buchberger es una base de Gröbner del ideal generado por la base original.

Teorema 9.25. *El algoritmo de Buchberger devuelve una base de Gröbner de la base original. Es decir, sea p un polinomio, $F = \{f_1, \ldots, f_k\}$ una secuencia finita de polinomios y $G = Buchberger(F)$ entonces:*

$$p \in \langle F \rangle \iff p \to_G^* 0$$

Demostración.

$$\begin{aligned} p \in \langle F \rangle &\iff p \in \langle G \rangle && \text{(teorema 9.23)} \\ &\iff p \to_G^* 0 && \text{(teorema 9.24)} \end{aligned}$$

□

Los teoremas ACL2 correspondientes se presentan a continuación.

```
(defthm |G = Buchberger(F) & p en <F> => p ->*G 0|
  (let* ((G (Buchberger F))
```

```
          (prueba (prueba-forma-normal p G)))
     (implies (and (naturalp k)
                   (k-polinomiop p)
                   (k-polinomiosp F)
                   (en-ideal p F))
              (->* p (nulo) prueba G))))

(defthm |G = Buchberger(F) & p ->*G 0 => p en <F>|
  (let ((G (Buchberger F)))
    (implies (and (naturalp k)
                  (k-polinomiop p)
                  (k-polinomiosp F)
                  (<->* p (nulo) prueba G))
             (en-ideal p F))))
```

9.5. Procedimiento de decisión de la pertenencia a ideales

En esta sección se construye y verifica un procedimiento de decisión para el problema de la pertenencia al ideal. Dicho procedimiento consiste en reducir el polinomio utilizando la función red^* sobre la base devuelta por el algoritmo de Buchberger y comprobar si es 0.

Definición 9.26. Dado p un polinomio y F una secuencia finita de polinomios definimos el predicado de pertenencia de p al ideal generado por F, $decide(p, F)$, de la siguiente forma:

$$decide(p, F) \equiv red^*_G(p) = 0, \text{ donde } G = Buchberger(F).$$

En ACL2,

```
(defun<k> decide (p F)
  (equal (red-F* p (Buchberger F)) (nulo)))
```

La corrección de este procedimiento de decisión para el problema de la pertenencia al ideal viene dado por el siguiente teorema.

Teorema 9.27. *Sea p un polinomio y F una secuencia finita de polinomios, entonces*

$$p \in \langle F \rangle \iff decide(p, F)$$

En ACL2,

```
(defthm |p en <F> <=> decide(p, F)|
    (implies (and (naturalp k)
                  (k-polinomiop p)
                  (k-polinomiosp F))
             (iff (en-ideal p F) (decide p F))))
```

Demostración.

$$\begin{aligned}
p \in \langle F\rangle &\iff p \to_G^* 0 && \text{(teorema 9.25)}\\
&\iff fn_G(p) = fn_G(0) && \text{(teoremas 8.13 y 9.3)}\\
&\iff red_G^*(p) = red_G^*(0) && \text{(teorema 7.40)}\\
&\iff red_G^*(p) = 0 && \text{(def. de } red^*\text{)}\\
&\iff decide(p, F) && \text{(def. de } decide\text{)}
\end{aligned}$$

□

Nótese que todas las funciones que intervienen en el procedimiento son computables y, por tanto, también lo es el procedimiento. Además, obsérvese como ni en su definición ni en el teorema que demuestra su corrección intervienen en ningún sentido los operadores ni las pruebas que intervenían en las relaciones de reducción. Esto es, los operadores y todos los conceptos asociados se han utilizado como mecanismo de demostración. Una vez todas las propiedades están demostradas, los operadores desaparecen del procedimiento de decisión y de su teorema de corrección.

Con estos teoremas quedan conseguidos los objetivos propuestos en esta memoria.

9.6. Resumen

En este capítulo:

- Hemos presentado una implementación ACL2 del algoritmo de Buchberger.
- Hemos demostrado su terminación mediante inmersión ordinal.

- Hemos demostrado su corrección parcial probando que las bases inicial y final generan el mismo ideal, y que la base final es de Gröbner.
- Hemos construido y verificado un procedimiento de decisión para el problema de la pertenencia a un ideal.

Los s-polinomios no son más que una forma concisa de representar los pares críticos que se producen al poder reducir un mismo elemento de dos formas distintas. En este sentido, la idea es similar a la que aparece en otros algoritmos de completación para obtener sistemas de reescritura de términos que ayuden a decidir una teoría ecuacional dada, notablemente, el de Knuth-Bendix.

Las bases de Gröbner, que son precisamente las que devuelve el algoritmo, permiten decidir efectivamente el problema de la pertenencia al ideal y, por lo tanto, la congruencia del ideal.

Dada la ejecutabilidad del algoritmo desarrollado hemos obtenido también un *procedimiento de decisión verificado* para el problema de la pertenencia al ideal.

Parte V

Conclusiones

Capítulo 10

Ejecutabilidad: algunos ejemplos

Una posibilidad que presenta ACL2 y que no aparece en otros sistemas es la de ejecutar directamente las funciones definidas en la lógica. De hecho, el propio sistema puede emplear sus capacidades de ejecución durante el curso de una demostración, a modo de ejecución simbólica. Por ejemplo, si definimos la función factorial y pedimos a ACL2 que demuestre que el factorial de 5 es 120, el sistema nos responderá afirmativamente y justificará la «demostración» diciéndonos que basta con ejecutar la función.

Sin embargo, el aspecto más interesante de la ejecución es que es un proceso independiente del de demostración que puede ser empleado por una persona para validar el modelo que está construyendo frente a ejemplos concretos. Evidentemente, estos ejemplos se eligen de manera que el comportamiento del sistema que se está modelando se conoce de antemano o bien porque son lo suficientemente sencillos o porque presentan alguna estructura común, o han sido estudiados con anterioridad por otras personas.

Explicaremos a continuación, muy brevemente, cómo funciona este mecanismo en ACL2 y resolveremos algunos problemas con nuestros algoritmos verificados. Los problemas que propondremos estarán principalmente basados en ejemplos extraídos de la bibliografía.

10.1. Protecciones, compilación y ejecución

Todas las funciones ACL2 susceptibles de ejecución que aparecen en este trabajo han sido etiquetadas con un aserto, denominado *protección* o *guarda*, que permite especificar el conjunto de entradas que la función es capaz

de procesar correctamente. De hecho, se ha demostrado para cada una de estas funciones que siempre que al evaluarla se satisface su protección, también se satisfacen las correspondientes a todas las evaluaciones de función que produce subsidiariamente. A este proceso se le denomina *verificación de protecciones*.

Aunque una protección bien puede entenderse como una precondición, evitamos referirnos a ellas como tales puesto que curiosamente no pertenecen a la lógica: no forman parte de la definición de la función, ni tienen influencia alguna sobre las propiedades que puedan demostrarse sobre ellas. Dado que las protecciones son un mecanismo extralógico y su verificación no ha supuesto en nuestro caso un esfuerzo especial, no hemos considerado necesario incluir en la memoria los detalles de dichas demostraciones, en general irrelevantes. No obstante, sí creemos importante explicar qué papel juegan en la ejecución de las funciones en ACL2.

A primera vista esto puede parecer innecesario, ya que hemos explicado que las funciones ACL2 son totales, por lo que producen un resultado para cualquier entrada (y siempre el mismo para la misma entrada) y, en este sentido, pueden procesar «correctamente» lo que se desee. Pero, como explicaremos a continuación, esto es sólo cierto desde un punto de vista estrictamente lógico.

Como lenguaje de programación, ACL2 está basado en LISP, del que intenta modelar un subconjunto. Como la mayoría de los lenguajes de programación reales, LISP posee primitivas que se corresponden con funciones parciales. La especificación de COMMON LISP está llena de expresiones del tipo: «y para cualquier otra entrada, el resultado de la ejecución depende del sistema». Este «depende del sistema» no significa necesariamente que cada implementador fijará a su criterio un resultado razonable a devolver en dichas circunstancias: el sistema podría producir un error de ejecución, devolver cada vez algo distinto o entrar en ciclo.

Ahora bien, existe una importante relación entre una misma evaluación de función en ACL2 y en COMMON LISP: *ambas deben producir exactamente el mismo resultado, en tanto en cuanto no se viole ninguna protección.* Por lo tanto, cuando una función ACL2 tiene su protección verificada se dice que «cumple» con COMMON LISP, haciendo referencia a este hecho.

ACL2 mantiene en memoria dos versiones de cada definición de función. Por un lado se encuentra la versión lógica que se corresponde con la *regla de definición* que se emplea habitualmente para razonar en las demostraciones. Esta versión coincide básicamente con lo que el usuario introduce en el sistema en

modo lógico.[1] Por otro lado, tenemos una versión de ejecución o *contrapartida ejecutable* que se emplea cuando se requiere ejecutar la función.

Así, por cada función definida se introducen en el sistema LISP subyacente dos versiones que poseen el mismo nombre, pero que se encuentran en paquetes distintos. Ninguna de ellas se compila inicialmente. La primera de ellas es la versión «cruda» que se obtiene al introducir directamente la definición proporcionada por el usuario en LISP. Ésta sólo se emplea para evaluar una llamada a función si su protección ha sido verificada y sus parámetros reales satisfacen la protección. En caso contrario, se emplea un «evaluador especial» implementado por la segunda versión, denominada versión «`*1*`», que se encarga de comprobar la protección en tiempo de ejecución y, si ésta ha sido violada, presenta un mensaje de error indicando la protección y la llamada que la infringe.

Por lo tanto, al evaluar una función que tiene su protección verificada sólo se comprueba la protección en la llamada más externa, con el consiguiente ahorro. Esto es válido, ya que precisamente, el proceso de verificación de guardas nos asegura que en tal caso es imposible violar la protección de ninguna de las funciones que pueda ser evaluada internamente como resultado de la llamada inicial.

Con la certificación de los libros que componen el trabajo se producen no sólo las demostraciones correspondientes, sino también los ficheros de código objeto que resultan del proceso de compilación de las funciones definidas en ACL2. Sólo se compila la versión cruda de cada función y la compilación se realiza a través del sistema LISP subyacente que normalmente produce código C y posteriormente emplea el compilador estándar disponible en el sistema operativo para producir el código objeto.

Esto permite utilizar durante la ejecución instrucciones de la máquina directamente en lugar de interpretar el código LISP original, que es bastante más ineficiente. De hecho, salvo indicación en contra, al incluir un libro en ACL2 se carga en memoria el código objeto correspondiente. Si éste no está disponible, se carga el código LISP original.

No obstante, para que ACL2 emplee la versión cruda, compilada, de una función su protección debe haberse verificado previamente.[2]

[1] En realidad, ACL2 emplea internamente una versión «normalizada» de la función, lógicamente equivalente pero más adecuada a los procesos internos que habitualmente realiza con ella.

[2] Existe la posibilidad de forzar la compilación de *ambas* versiones de una función con `comp` aunque su protección no haya sido verificada. Así siempre se ejecutará una versión

10.2. Ejemplos

Hemos diseñado una serie de utilidades basadas en macros y funciones de ACL2 que nos facilitan la tarea de crear problemas de prueba para el código ACL2 que hemos desarrollado en la memoria y de imprimir los resultados que se obtienen de su ejecución.

Los siguientes datos de entrada, que explicaremos en breve, corresponden a un problema que hemos incluido en el fichero `ejemplos-winkler.lisp`. Este problema en concreto se basa en un ejemplo que aparece en [102].

```
(problema
 (imprime "~%*** Winkler: Ejemplo 8.4.1 (pp. 190)~%")

 ;;; Establecemos las variables y su orden:

 (vars z y x)

 ;;; Establecemos la base inicial:

 (<- f1 '((4 1 0 0) (-4 0 2 1) (-16 0 0 2) (-1 0 0 0)))
 (<- f2 '((2 1 2 0) (4 0 0 1) (1 0 0 0)))
 (<- f3 '((2 1 0 2) (2 0 2 0) (1 0 0 1)))

 (<- F (base f1 f2 f3))

 (imprime "~%Base inicial:~%")
 (imprime-base F :nombre "f")

 ;;; Aplicamos el algoritmo de Buchberger:

 (<- F (base f1 f2 f3))

 (imprime "~%Base de Gröbner:~%")
 (imprime-tiempo (<- G (buchberger F)))
 (imprime-base G :nombre "g")

 (imprime "~%Escalado:~%")
```

compilada, pero se pierde toda garantía sobre el resultado de una ejecución. Si aparece durante ella un valor imprevisto como entrada de alguna función, al no tener ésta la protección verificada, la función aceptaría dicho valor extraño produciendo un resultado dependiente del sistema.

```
 (imprime-base G :nombre "g" :fmt escalada)

 (imprime "~%Escalado e interreducción:~%")
 (imprime-base G :nombre "g" :fmt escalada-reducida)

 ;;; Mostramos la reducción a 0 de un polinomio respecto de la base:

 (<- h '((4 0 4 1) (16 0 2 2) (1 0 2 0) (8 0 0 1) (2 0 0 0)))
 (<- r (red* h G))

 (imprime "~%")
 (imprime-polinomio h :pre "h = " :nl t)
 (imprime-polinomio r :pre "\\red^*_G(h) = " :nl t)
)
```

La macro `problema` crea un entorno de evaluación ACL2 en el que sólo se pueden mostrar los efectos colaterales de las instrucciones que lo componen. Dentro del entorno podemos manejar objetos que se asignan mediante la macro `<-`. Estos objetos se comportan internamente como variables globales, por lo que pueden ser compartidos por distintos problemas. También se pueden ejecutar funciones como `red`, `red*` y `buchberger` que pueden recibir dichos variables y que devuelven valores asignables a ellos.

El conjunto de variables con las que se forman los términos y su orden relativo se fijan mediante la macro `vars`, en la que las variables aparecen en orden decreciente de importancia. Esto se hace habitualmente al principio del problema, aunque puede cambiarse posteriormente tantas veces como se desee. Así, `(vars z y x)` indica que se considerará $X = \{x, y, z\}$ con $x < y < z$.

Los polinomios se representan por listas de $k+1$-uplas, cada una de las cuales tiene como primer elemento un coeficiente al que siguen los exponentes de cada una de las k variables sobre las que se define el polinomio. Por ejemplo, al ejecutar `(<- p '((4 1 0 0) (-4 0 2 1) (-16 0 0 2) (-1 0 0 0)))` se obtiene $4z - 4y^2x - 16x^2 - 1$, si se ha empleado previamente `(vars z y x)`, mientras que si se emplea `(vars x y z)` se obtiene $4x - 4y^2z - 16z^2 - 1$. Las bases se pueden forman con la macro `base`, a partir de los objetos que contienen sus polinomios, y asignarse a otro objeto para su empleo posterior.

Se puede imprimir un resultado a través de alguna de las macros `imprime` disponibles. Estas macros efectúan la salida a través de una «ventana de comentarios» de ACL2, es decir, emplean internamente `cw` para mostrar texto de manera no aplicativa. Esto es cómodo, ya que puede hacerse sin necesidad

de manipular directamente el estado ni los flujos de salida.[3]

Como se observa, las macros admiten parámetros opcionales que afectan a la forma de imprimir los resultados. Por ejemplo, `imprime-base` contiene una clave `:fmt` que permite indicar el formato en el que se desea imprimir:

1. Si es `nil` o no aparece, se imprime tal cual.
2. Si es `escalada`, se escala cada uno de los elementos de la base de manera que no aparezcan coeficientes fraccionarios.
3. Si es `escalada-reducida` se interreducen los polinomios entre sí, eliminando los nulos y escalando los restantes.

Nótese que los procesos de impresión, escalado e interreducción no están verificados y se incluyen aquí como una mera herramienta que facilita la interpretación de los resultados.

La salida de las funciones `imprime-polinomio` y `imprime-base` contiene código LaTeX que puede ser incluido en un documento con poca o nula edición. Por ejemplo, utilizamos fragmentos de la salida en la descripción de los ejemplos que expondremos más adelante. Lo que sigue es un fragmento de la salida producida al resolver el problema anterior:

```
*** Winkler: Ejemplo 8.4.1 (pp. 190)

Base inicial:
\begin{align*}
f_1 &= 4z-4y^2x-16x^2-1 \\
f_2 &= 2zy^2+4x+1 \\
f_3 &= 2zx^2+2y^2+x
\end{align*}

Base de Gröbner:
Tiempo: 205.32 s

...

Escalado e interreducción:
```

[3] Desde un punto de vista lógico, `cw` siempre devuelve `nil`, con lo que estas macros carecen de interés en la lógica. Su único propósito es ayudarnos a visualizar la representación interna de los polinomios y facilitar la inclusión de los resultados en esta memoria.

```
\begin{align*}
g_1 &= 32x^7-216x^6+34x^4-12x^3-x^2+30x+8 \\
g_2 &= 2745y^2-112x^6-812x^5+10592x^4-61x^3-812x^2+988x+2 \\
g_3 &= 10980z-6272x^6+42368x^5+232x^4-3416x^3-39982x^2+428x-2633
\end{align*}

h = 4y^4x+16y^2x^2+y^2+8x+2
\red^*_G(h) = 0
```

Hay que tener en cuenta que todas las operaciones de impresión sobre polinomios son relativas al conjunto actual de variables y al orden establecido entre ellas.

En adelante, omitiremos el código interno y mostraremos las porciones relevantes de los resultados generados en LaTeX para mayor legibilidad.

Comenzamos con una prueba intensiva de las funciones sobre polinomios. Calculamos $(x+1)^n$ para diversos valores de n. Los tiempos en segundos se muestran en la siguiente tabla:

n	25	50	75	100	125	150	175	200	225	250
TIEMPO	0.0	0.2	1.5	4.1	10.3	19.1	36.6	61.9	98.1	150.4

En la última potencia generada se emplea bastante tiempo. El polinomio resultante tiene 251 monomios y el coeficiente máximo es el siguiente número de 74 cifras:

$$\binom{250}{125} = 91208366928185711600087718663295946582847985411225264672245111235434562752$$

Continuamos con unos ejemplos extraídos del popular libro de Geddes, Czapor y Labahn [26].

- En el ejemplo 10.4, pág. 434, se presentan los siguientes polinomios. Tienen una única variable y ambos son de distinto grado. El de mayor grado puede reducirse por el de menor grado hasta anularse, de hecho es divisible por él.

$$\begin{aligned} p &= 6x^4 + 13x^3 - 6x + 1 \\ q &= 3x^2 + 5x - 1 \end{aligned}$$

En efecto, podemos observar que al realizar dos pasos de reducción de p por q se obtiene el polinomio nulo.

$$\begin{aligned} s_1 &= red(p,q) = 3x^3 + 2x^2 - 6x + 1 \\ s_2 &= red(s_1,q) = -3x^2 - 5x + 1 \\ s_3 &= red(s_2,q) = 0 \end{aligned}$$

Esto se puede comprobar directamente calculando la clausura de la reducción o, lo que es lo mismo, su forma normal o irreducible respecto a $\{q\}$.

$$r = red^*_{\{q\}}(p) = 0$$

Consideremos ahora la base formada por los polinomios p y q. Estamos ante una base que ya es de Gröbner y, de hecho, podemos comprobar que el algoritmo de Buchberger no añade ningún polinomio a la base. El algoritmo devuelve $G = \{g_1, g_2\}$ como base de Gröbner:

$$\begin{aligned} g_1 &= 6x^4 + 13x^3 - 6x + 1 \\ g_2 &= 3x^2 + 5x - 1 \end{aligned}$$

- En el ejemplo 10.5, pág. 434–435, se presenta la siguiente base.

$$\begin{aligned} f_1 &= -xz^2 + 2y^2z \\ f_2 &= 7y^2 + yz - 4 \\ f_3 &= -3x + 2yz + 1 \end{aligned}$$

Podemos calcular una base de Gröbner con orden lexicográfico $x > y > z$ (tarda 2 s).

$$\begin{aligned} g_1 &= -\frac{7}{72}z^5 - \frac{1}{24}z^4 + \frac{49}{1296}z^3 - \frac{7}{27}z^2 + \frac{4}{9}z \\ g_2 &= -\frac{648}{49}yz + \frac{9}{7}z^4 + \frac{243}{49}z^3 - \frac{1}{2}z^2 + \frac{12}{7}z \\ g_3 &= \frac{1}{2}yz^2 - \frac{12}{7}yz + \frac{9}{14}z^3 \\ g_4 &= \frac{2}{3}yz^3 + \frac{2}{7}yz^2 + \frac{1}{3}z^2 - \frac{8}{7}z \\ g_5 &= -xz^2 + 2y^2z \\ g_6 &= 7y^2 + yz - 4 \\ g_7 &= -3x + 2yz + 1 \end{aligned}$$

Escalando la base adecuadamente podemos eliminar los coeficientes racionales. En este caso, obtenemos:

$$\begin{aligned}
g_1 &= 126z^5 + 54z^4 - 49z^3 + 336z^2 - 576z \\
g_2 &= 1296yz - 126z^4 - 486z^3 + 49z^2 - 168z \\
g_3 &= 7yz^2 - 24yz + 9z^3 \\
g_4 &= 14yz^3 + 6yz^2 + 7z^2 - 24z \\
g_5 &= xz^2 - 2y^2z \\
g_6 &= 7y^2 + yz - 4 \\
g_7 &= 3x - 2yz - 1
\end{aligned}$$

Si además se interreducen los polinomios que la componen y se eliminan los que se reducen a 0 resulta la siguiente base.

$$\begin{aligned}
g_1 &= 126z^5 + 54z^4 - 49z^3 + 336z^2 - 576z \\
g_2 &= 1296yz - 126z^4 - 486z^3 + 49z^2 - 168z \\
g_3 &= 9072y^2 + 126z^4 + 486z^3 - 49z^2 + 168z - 5184 \\
g_4 &= 1944x - 126z^4 - 486z^3 + 49z^2 - 168z - 648
\end{aligned}$$

- En el ejemplo 10.6, pág. 437, se consideran los siguientes polinomios en orden lexicográfico $x > y$.

$$\begin{aligned}
p_1 &= x^2y + 5x^2 + y^2 \\
p_2 &= 7xy^2 - 2y^3 + 1 \\
q &= 3x^3y + 2x^2y^2 - 3xy + 5x
\end{aligned}$$

Podemos llevar a cabo la siguiente secuencia de reducciones, que representa 3 pasos de reducción de q por p_1:

$$\begin{aligned}
s_1 &= red(q, p_1) = -15x^3 + 2x^2y^2 - 3xy^2 - 3xy + 5x \\
s_2 &= red(s_1, p_1) = -15x^3 - 10x^2y - 3xy^2 - 3xy + 5x - 2y^3 \\
s_3 &= red(s_2, p_1) = -15x^3 + 50x^2 - 3xy^2 - 3xy + 5x - 2y^3 + 10y^2
\end{aligned}$$

y, si reducimos el resultado por p_2, obtenemos el irreducible de q respecto de $\{p_1, p_2\}$.

$$s_4 = red(s_3, p_2) = -15x^3 + 50x^2 - 3xy + 5x - \frac{20}{7}y^3 + 10y^2 + \frac{3}{7}$$

Calculemos una base de Gröbner de $F = \{p_1, p_2, q\}$. La base contiene 16 polinomios (tarda 19 s). Se observa un gran crecimiento en los coeficientes fraccionarios; mostramos algunos de los polinomios resultantes:

$$
\begin{aligned}
g_1 &= -\frac{1}{7} \\
g_2 &= \frac{13094326044856861224890276}{490165562235497422092752025} y^2 \\
g_3 &= -\frac{155519332574413599605832419 25}{104787785624281146117884 7376} y^3 + \cdots \\
&\vdots \\
g_{14} &= x^2 y + 5x^2 + y^2 \\
g_{15} &= 7xy^2 - 2y^3 + 1 \\
g_{16} &= 3x^3 y + 2x^2 y^2 - 3xy + 5x
\end{aligned}
$$

Si escalamos la base se obtienen los siguientes resultados.

$$
\begin{aligned}
g_1 &= 1\\
g_2 &= y^2\\
g_3 &= 22139682975045y^3 - 562023154678y^2\\
g_4 &= 32370941540876y^4 - 121823193288521y^3 + 2206901437070y^2\\
g_5 &= 140489213643853y^5 + 15845831250081y^4 + 45274160744736y^3\\
&\quad - 15804397095055y^2\\
g_6 &= 2012574073853y^6 - 628607499626y^5 + 226555829773y^4\\
&\quad + 722888054490y^3 + 74059817115y^2\\
g_7 &= 11594545037y^7 - 15276539911y^6 + 282755738y^5 + 2326660652y^4\\
&\quad - 5262320751y^3 + 95652820y^2\\
g_8 &= 111788263y^8 + 268128y^7 - 4604236y^6 - 32403020y^5 - 66444y^4\\
&\quad + 1150079y^3 + 8101735y^2\\
g_9 &= 4y^9 + 69y^8 - 4y^6 - 20y^5 + y^3 + 5y^2\\
g_{10} &= 16y^{10} + 552y^9 + 4761y^8 - 16y^7 - 356y^6 - 1380y^5 + 4y^4 + 89y^3\\
&\quad + 345y^2\\
g_{11} &= 214375x + 16y^9 + 472y^8 + 2401y^7 - 12021y^6 + 59749y^5 - 422625y^4\\
&\quad + 4y^3 + 69y^2 + 61250y\\
g_{12} &= 7xy + 35x - 4y^5 - 69y^4 + 2y^2 + 10y\\
g_{13} &= 1225x^2 + 7x - 4y^4 - 49y^3 + 245y^2 + 2y\\
g_{14} &= x^2y + 5x^2 + y^2\\
g_{15} &= 7xy^2 - 2y^3 + 1\\
g_{16} &= 3x^3y + 2x^2y^2 - 3xy + 5x
\end{aligned}
$$

Interreduciendo los polinomios se queda sólo $\{1\}$.

- En el ejemplo 10.8, pág. 439–440, aparece la siguiente base:

$$
\begin{aligned}
f_1 &= x^3yz - xz^2\\
f_2 &= xy^2z - xyz\\
f_3 &= x^2y^2 - z^2\\
f_4 &= x^2y^2z - z^3\\
f_5 &= -x^2y^2z + x^2yz
\end{aligned}
$$

La base de Gröbner escalada que devuelve el algoritmo con orden lexicográfico $x > y > z$ se compone de una veintena de polinomios que se calculan instantáneamente.

$$
\begin{aligned}
g_1 &= yz^3 - z^3 & g_{11} &= z^7 - z^6 \\
g_2 &= x^2yz - z^3 & g_{12} &= x^2z^3 - z^6 \\
g_3 &= z^5 - z^4 & g_{13} &= xz^4 - xz^3 \\
g_4 &= xz^3 - xz^2 & g_{14} &= x^3z^2 - xz^3 \\
g_5 &= yz^4 - z^4 & g_{15} &= xyz^2 - xz^2 \\
g_6 &= x^2z^2 - z^4 & g_{16} &= x^3yz - xz^2 \\
g_7 &= yz^5 - z^5 & g_{17} &= xy^2z - xyz \\
g_8 &= z^6 - z^5 & g_{18} &= x^2y^2 - z^2 \\
g_9 &= yz^6 - z^6 & g_{19} &= x^2y^2z - z^3 \\
g_{10} &= y^2z^6 - z^6 & g_{20} &= x^2y^2z - x^2yz
\end{aligned}
$$

Al reducirla desaparecen 12 de ellos.

$$
\begin{aligned}
g_1 &= yz^3 - z^3 \\
g_2 &= x^2yz - z^3 \\
g_3 &= z^5 - z^4 \\
g_4 &= xz^3 - xz^2 \\
g_5 &= x^2z^2 - z^4 \\
g_6 &= xyz^2 - xz^2 \\
g_7 &= xy^2z - xyz \\
g_8 &= x^2y^2 - z^2
\end{aligned}
$$

- El ejemplo 10.10, pág. 445–446, presenta el siguiente sistema.

$$
\begin{aligned}
f_1 &= x^2 + yz - 2 \\
f_2 &= xz + y^2 - 3 \\
f_3 &= xy + z^2 - 5
\end{aligned}
$$

Los autores del libro indican que éste es un ejemplo complejo a pesar de su aspecto. Al calcular una base de Gröbner con $x > y > z$ observamos que se emplean 388 s y que se consume una gran cantidad de memoria.

Se observa también un gran crecimiento de los coeficientes fraccionarios implicados. Mostramos únicamente una porción de la base resultante, que contiene 22 polinomios.

$$
\begin{aligned}
g_1 &= -\frac{68283}{242}z^8 + \frac{1707075}{484}z^6 - \frac{14953977}{968}z^4 + \frac{6486885}{242}z^2 - \frac{24650163}{1936} \\
g_2 &= -\frac{1830125}{12791682}z^{10} + \frac{127942375}{25583364}z^8 - \frac{2455528625}{51166728}z^6 + \frac{9695170375}{51166728}z^4 + \cdots \\
g_3 &= \frac{13488}{166375}z^{12} - \frac{193352}{99825}z^{10} + \frac{9343204}{499125}z^8 - \frac{9230828}{99825}z^6 + \cdots \\
&\vdots \\
g_{20} &= xy + z^2 - 5 \\
g_{21} &= xz + y^2 - 3 \\
g_{22} &= x^2 + yz - 2
\end{aligned}
$$

Mostramos a continuación el resultado escalado en el que desaparecen los coeficientes fraccionarios.

$$
\begin{aligned}
g_1 &= 8z^8 - 100z^6 + 438z^4 - 760z^2 + 361\\
g_2 &= 88z^{10} - 3076z^8 + 29518z^6 - 116546z^4 + 191691z^2 - 89167\\
g_3 &= 40464z^{12} - 966760z^{10} + 9343204z^8 - 46154140z^6 + 120394688z^4\\
&\quad - 150551820z^2 + 61631725\\
g_4 &= 540552375x + 6655504z^{13} - 175635360z^{11} + 1879326144z^9\\
&\quad - 10317939790z^7 + 30066113368z^5 - 42344055145z^3 + 19774993850z\\
g_5 &= 8z^{14} - 220z^{12} + 2538z^{10} - 15830z^8 + 57111z^6 - 117165z^4 + 122075z^2\\
&\quad - 45125\\
g_6 &= 8z^{15} - 220z^{13} + 2538z^{11} - 15830z^9 + 57111z^7 - 117165z^5 + 122075z^3\\
&\quad - 45125z\\
g_7 &= 8z^{16} - 220z^{14} + 2538z^{12} - 15830z^{10} + 57111z^8\\
&\quad - 117165z^6 + 122075z^4 - 45125z^2\\
g_8 &= 27387987y - 32552z^{15} + 848668z^{13} - 8559970z^{11}\\
&\quad + 40898478z^9 - 85622161z^7 + 29485715z^5 + 84451504z^3 + 14414730z\\
g_9 &= 1441473yz - 2448z^{14} + 93008z^{12} - 1237568z^{10}\\
&\quad + 7724342z^8 - 23539930z^6 + 30588141z^4 - 8905205z^2\\
g_{10} &= 231933yz^2 - 331398y + 4696z^{13} - 95548z^{11} + 747970z^9 - 2751536z^7\\
&\quad + 4564851z^5 - 2454721z^3 - 174420z\\
g_{11} &= 918yz^3 + 9633yz + 328z^{12} - 6436z^{10} + 47758z^8 - 160664z^6 + 222531z^4\\
&\quad - 68305z^2\\
g_{12} &= 587yz^4 - 4199yz^2 + 14801y + 152z^{11} - 2868z^9 + 20156z^7 - 61583z^5\\
&\quad + 65957z^3 + 7790z\\
g_{13} &= 41yz^5 - 323yz^3 + 722yz + 8z^{10} - 138z^8 + 899z^6 - 2596z^4 + 2755z^2\\
g_{14} &= 19yz^6 - 164yz^4 + 437yz^2 - 361y + 6z^9 - 75z^7 + 299z^5 - 332z^3 - 190z\\
g_{15} &= 4yz^7 - 41yz^5 + 143yz^3 - 190yz + 6z^8 - 79z^6 + 340z^4 - 475z^2\\
g_{16} &= 4yz^8 - 60yz^6 + 307yz^4 - 627yz^2 + 361y - 4z^7 + 41z^5 - 143z^3 + 190z\\
g_{17} &= 57y^2 - 4yz^7 + 60yz^5 - 269yz^3 + 342yz + 4z^6 - 41z^4 + 105z^2\\
g_{18} &= 3y^2z + 2yz^4 - 15yz^2 + 19y - 2z^3 + 10z\\
g_{19} &= y^3 - 3y - z^3 + 5z\\
g_{20} &= xy + z^2 - 5\\
g_{21} &= xz + y^2 - 3\\
g_{22} &= x^2 + yz - 2
\end{aligned}
$$

En realidad, la base se puede simplificar para obtener:

$$\begin{aligned} g_1 &= 8z^8 - 100z^6 + 438z^4 - 760z^2 + 361 \\ g_2 &= 361x - 88z^7 + 872z^5 - 2690z^3 + 2375z \\ g_3 &= 361y + 8z^7 + 52z^5 - 740z^3 + 1425z \end{aligned}$$

- El ejemplo 10.16, pág. 455, contiene el siguiente sistema:

$$\begin{aligned} f_1 &= xy + xz - x + z^2 - 2 \\ f_2 &= xy^2 + 2xz - 3x + y + z - 1 \\ f_3 &= y^3 + y^2z + 2yz - 3y + 2z^2 - 3z \end{aligned}$$

Podemos calcular una base de Gröbner empleando el orden lexicográfico con $x > y > z$ en 77 s. Presentamos la base convenientemente escalada:

$$\begin{aligned} g_1 &= z^6 + 2z^5 - 7z^4 - 8z^3 + 15z^2 + 8z - 10 \\ g_2 &= 3z^7 + 11z^6 - 11z^5 - 59z^4 + 5z^3 + 99z^2 + 10z - 50 \\ g_3 &= z^8 + 2z^7 - 10z^6 - 14z^5 + 36z^4 + 32z^3 - 55z^2 - 24z + 30 \\ g_4 &= 2z^9 + 7z^8 - 14z^7 - 58z^6 + 30z^5 + 172z^4 - 14z^3 - 213z^2 - 12z + 90 \\ g_5 &= z^{10} + 2z^9 - 12z^8 - 18z^7 + 56z^6 + 60z^5 - 127z^4 - 88z^3 + 140z^2 + 48z \\ &\quad - 60 \\ g_6 &= 4y - 11z^9 - 25z^8 + 103z^7 + 177z^6 - 357z^5 - 413z^4 + 542z^3 + 326z^2 \\ &\quad - 300z - 20 \\ g_7 &= 22yz - 6y - 122z^8 - 270z^7 + 1159z^6 + 1894z^5 - 4083z^4 - 4344z^3 \\ &\quad + 6331z^2 + 3244z - 3600 \\ g_8 &= 122yz^2 - 26yz - 248y + 3z^7 - 59z^5 - 38z^4 + 337z^3 + 32z^2 - 440z + 40 \\ g_9 &= 4yz^3 + 8yz^2 - 10yz - 18y - z^6 - 4z^5 + 5z^4 + 22z^3 - 7z^2 - 28z \\ g_{10} &= 3yz^4 - 6yz^3 - yz^2 + 10yz - 8y + z^5 - 8z^4 + 13z^3 + 8z^2 - 26z + 10 \\ g_{11} &= 4y^2 + 3yz^3 - 3yz^2 + 2yz + z^4 - 7z^3 + 8z^2 + 8z - 10 \\ g_{12} &= 2y^2z - 2y^2 + 3yz^2 - 6yz + 4y + z^3 - 4z^2 + 4z \\ g_{13} &= y^2z^2 - 3y^2 - 2yz + 2y + 2z^3 - 4z^2 - 2z + 5 \\ g_{14} &= xz^2 - 2x - yz^2 + 3y + z^3 - z^2 - z + 1 \\ g_{15} &= xy + xz - x + z^2 - 2 \\ g_{16} &= xy^2 + 2xz - 3x + y + z - 1 \\ g_{17} &= y^3 + y^2z + 2yz - 3y + 2z^2 - 3z \end{aligned}$$

La interreducción produce la siguiente base:

$$\begin{aligned} g_1 &= z^6 + 2z^5 - 7z^4 - 8z^3 + 15z^2 + 8z - 10 \\ g_2 &= y + z^4 + 2z^3 - 5z^2 - 3z + 5 \\ g_3 &= xz^2 - 2x - z^4 + 4z^2 - 4 \end{aligned}$$

- Finalmente, el ejercicio 10.20, pág. 464, plantea un problema de geometría en el plano que puede resolverse a través del cálculo de una base de Gröbner del ideal generado por el siguiente sistema:

$$\begin{aligned} f_1 &= y_1 - 2y_3 \\ f_2 &= y_2 - 2y_4 \\ f_3 &= y_1 y_3 - y_2 y_4 \\ f_4 &= y_1^2 z - y_2^2 z - 1 \end{aligned}$$

Calculamos una base con orden lexicográfico $y_1 > y_2 > y_3 > y_4 > z$. El problema se resuelve instantáneamente obteniéndose los siguientes polinomios:

$$\begin{aligned} g_1 &= 1 \\ g_2 &= y_4^2 \\ g_3 &= -2y_3^2 + 2y_4^2 \\ g_4 &= y_1 - 2y_3 \\ g_5 &= y_2 - 2y_4 \\ g_6 &= y_1 y_3 - y_2 y_4 \\ g_7 &= y_1^2 z - y_2^2 z - 1 \end{aligned}$$

De hecho, simplificando la base se obtiene que el ideal está generado únicamente por $\{1\}$. Esto indica que el sistema es inconsistente y, como el método que se emplea está basado en la refutación, se concluye que el enunciado geométrico es válido.

Seguidamente presentamos la solución de algunos problemas inspirados en conjuntos de polinomios que aparecen en los ejemplos del libro de Winkler [102].

- En el ejemplo 8.3.1, pág. 186-87, aparece el siguiente conjunto de polinomios:

$$f_1 = yx^2 + x$$
$$f_2 = y^2x^2 + y - 1$$

  Se desea obtener una base de Gröbner del ideal generado por f_1 y f_2 Este problema es sencillo y una solución, con orden lexicográfico $y > x$, se obtiene instantáneamente:

$$g_1 = -x$$
$$g_2 = -y + 1$$
$$g_3 = yx - y + 1$$
$$g_4 = yx^2 + x$$
$$g_5 = y^2x^2 + y - 1$$

  Podemos simplificar la base, en cuyo caso se obtiene:

$$g_1 = x$$
$$g_2 = y - 1$$

- El ejemplo 8.3.2, pág. 188–89, trata del ideal generado por los siguientes polinomios:

$$f_1 = x^3y - 2y^2 - 1$$
$$f_2 = x^2y^2 + x + y$$

  Podemos calcular una base de Gröbner con orden lexicográfico $x > y$ (tarda 25 s) que en forma escalada queda:

$$
\begin{aligned}
g_1 &= 4y^9 + 4y^7 - 5y^5 + y^4 - 3y^3 + 2y^2 + 1\\
g_2 &= 8y^{11} + 12y^9 - 6y^7 + 2y^6 - 11y^5 + 5y^4 - 3y^3 + 4y^2 + 1\\
g_3 &= 14x - 216y^{10} - 40y^9 - 236y^8 - 52y^7 + 254y^6 - 36y^5 + 169y^4 - 84y^3\\
&\quad - 17y^2 - 37y - 11\\
g_4 &= 27xy - 5x + 184y^9 - 16y^8 + 196y^7 - 56y^6 - 234y^5 + 38y^4 - 159y^3\\
&\quad + 143y^2 - 8y + 56\\
g_5 &= 23xy^2 + 2xy + 9x - 8y^8 - 40y^7 - 28y^6 - 48y^5 - 4y^4 + 24y^3 + 14y^2\\
&\quad + 19y + 5\\
g_6 &= xy^3 - 5xy^2 - 2x + 8y^7 + 4y^6 + 10y^5 + 2y^4 - 5y^3 - 2y^2 - 4y - 1\\
g_7 &= 2xy^4 - xy^3 + xy^2 - 2y^5 - y^3 + 2y^2 + 1\\
g_8 &= x^2 + xy + 2y^3 + y\\
g_9 &= x^3y - 2y^2 - 1\\
g_{10} &= x^2y^2 + x + y
\end{aligned}
$$

A continuación mostramos el resultado en forma reducida:

$$
\begin{aligned}
g_1 &= 4y^9 + 4y^7 - 5y^5 + y^4 - 3y^3 + 2y^2 + 1\\
g_2 &= 14x - 20y^8 - 12y^7 - 16y^6 - 32y^5 + 17y^4 - 6y^3 + 3y^2 + 17y - 1
\end{aligned}
$$

- En el ejemplo 8.4.1, pág. 190, se presenta el siguiente sistema y se quiere comprobar si un cierto polinomio $h = 4y^4x + 16y^2x^2 + y^2 + 8x + 2$ se encuentra en el ideal generado por él.

$$
\begin{aligned}
f_1 &= 4z - 4y^2x - 16x^2 - 1\\
f_2 &= 2zy^2 + 4x + 1\\
f_3 &= 2zx^2 + 2y^2 + x
\end{aligned}
$$

Para ello, calculamos primero una base de Gröbner (tarda 205 s). El resultado escalado es:

$$
\begin{aligned}
g_1 &= 32x^7 - 216x^6 + 34x^4 - 12x^3 - x^2 + 30x + 8\\
g_2 &= 32x^8 - 216x^7 + 34x^5 - 12x^4 - x^3 + 30x^2 + 8x\\
g_3 &= 10496x^9 - 68704x^8 - 14472x^7 + 11152x^6 - 1658x^5 - 1132x^4\\
&\quad + 9773x^3 + 4634x^2 + 536x\\
g_4 &= 512x^{10} - 3200x^9 - 1696x^8 + 328x^7 + 80x^6 - 78x^5 + 460x^4\\
&\quad + 367x^3 + 94x^2 + 8x\\
g_5 &= 512x^{11} - 3200x^{10} - 1696x^9 + 328x^8 + 80x^7 - 78x^6 + 460x^5\\
&\quad + 367x^4 + 94x^3 + 8x^2\\
g_6 &= 1976400y^2 - 3085904384x^{10} + 20179639936x^9 + 4386125088x^8\\
&\quad - 3259718248x^7 + 466319312x^6 + 336680038x^5 - 2864037092x^4\\
&\quad - 1381005851x^3 - 165785474x^2\\
g_7 &= 30135785y^2x + 8718140y^2 + 29598464x^9 - 212333760x^8\\
&\quad + 83417872x^7 + 22349676x^6 + 89277032x^5 + 36722685x^4 + 19041872x^3\\
&\quad + 3908059x^2\\
g_8 &= 231238y^2x^2 + 213620y^2x + 42592y^2 - 70912x^8 + 337728x^7\\
&\quad + 828176x^6 + 741228x^5 + 114120x^4 + 21059x^3 + 6826x^2\\
g_9 &= 554y^2x^3 - 824y^2x^2 - 959y^2x - 196y^2 + 512x^7 - 2880x^6 - 3496x^5\\
&\quad - 398x^4 + 4x^3 - 17x^2\\
g_{10} &= 16y^2x^4 - 10y^2x^3 + 15y^2x + 4y^2 + 56x^5 + 14x^4 + 4x^3 + x^2\\
g_{11} &= 2y^4 + y^2x - 4x^3 - x^2\\
g_{12} &= 2zx^2 + 2y^2 + x\\
g_{13} &= 2zy^2 + 4x + 1\\
g_{14} &= 4z - 4y^2x - 16x^2 - 1
\end{aligned}
$$

Efectivamente, al reducir h respecto de G se obtiene que $red^*_G(h) = 0$. En realidad, los elementos de G pueden simplificarse, produciendo la siguiente base de Gröbner:

$$
\begin{aligned}
g_1 &= 32x^7 - 216x^6 + 34x^4 - 12x^3 - x^2 + 30x + 8\\
g_2 &= 2745y^2 - 112x^6 - 812x^5 + 10592x^4 - 61x^3 - 812x^2 + 988x + 2\\
g_3 &= 10980z - 6272x^6 + 42368x^5 + 232x^4 - 3416x^3 - 39982x^2 + 428x\\
&\quad - 2633
\end{aligned}
$$

- En el ejemplo 8.4.2, pág. 193, se calcula la base de Gröbner del siguiente sistema, con orden lexicográfico $x < y < z$.

$$\begin{aligned} f_1 &= 2zx^2 + 2y^2 + x \\ f_2 &= 2zy^2 + 4x + 1 \\ f_3 &= 4zx - 4y^2x - 16x^2 - 1 \end{aligned}$$

El resultado escalado (tarda 60 s) que proporciona el algoritmo es el siguiente:

$$\begin{aligned} g_1 &= 65z + 64x^4 - 432x^3 + 168x^2 - 354x + 104 \\ g_2 &= 32x^5 - 216x^4 + 64x^3 - 42x^2 + 32x + 5 \\ g_3 &= 47360x^6 - 306336x^5 + 4648x^4 - 35472x^3 + 29846x^2 + 20744x + 2085 \\ g_4 &= 799232x^7 - 5816448x^6 + 4267360x^5 - 696696x^4 + 998384x^3 \\ &\quad - 64282x^2 - 243000x - 27675 \\ g_5 &= 217088x^8 - 1419776x^7 + 125056x^6 - 181984x^5 + 144488x^4 \\ &\quad + 84384x^3 + 3694x^2 + 1200x + 225 \\ g_6 &= 1950y^2 - 591872x^8 + 3803648x^7 + 106880x^6 + 406816x^5 \\ &\quad - 345584x^4 - 273348x^3 - 33040x^2 + 975x \\ g_7 &= 512x^9 - 3200x^8 - 672x^7 - 376x^6 + 240x^5 + 294x^4 + 72x^3 + 5x^2 \\ g_8 &= 512x^{10} - 3200x^9 - 672x^8 - 376x^7 + 240x^6 + 294x^5 + 72x^4 + 5x^3 \\ g_9 &= 975y^2x - 312576x^9 + 2005824x^8 + 75280x^7 + 215628x^6 - 180592x^5 \\ &\quad - 146229x^4 - 18860x^3 + 325x^2 \\ g_{10} &= 31746y^2x^2 + 5304y^2x + 4864x^8 - 43712x^7 + 63632x^6 + 100460x^5 \\ &\quad + 21640x^4 + 7373x^3 + 1768x^2 \\ g_{11} &= 342y^2x^3 + 936y^2x^2 + 147y^2x - 512x^7 + 2880x^6 + 2984x^5 \\ &\quad + 654x^4 + 220x^3 + 49x^2 \\ g_{12} &= 16y^2x^4 - 10y^2x^3 + 16y^2x^2 + 3y^2x + 56x^5 + 14x^4 + 4x^3 + x^2 \\ g_{13} &= 2y^4 + y^2x - 4x^3 - x^2 \\ g_{14} &= 2zx^2 + 2y^2 + x \\ g_{15} &= 2zy^2 + 4x + 1 \\ g_{16} &= 4zx - 4y^2x - 16x^2 - 1 \end{aligned}$$

Y una vez reducido el resultado, queda así:

$$
\begin{aligned}
g_1 &= 65z + 64x^4 - 432x^3 + 168x^2 - 354x + 104 \\
g_2 &= 32x^5 - 216x^4 + 64x^3 - 42x^2 + 32x + 5 \\
g_3 &= 26y^2 - 16x^4 + 108x^3 - 16x^2 + 17x
\end{aligned}
$$

Capítulo 11

Conclusiones

En esta memoria se ha presentado una teoría computacional acerca del algoritmo de Buchberger para el cálculo de bases de Gröbner de ideales polinómicos.

- Se han formalizado los anillos de polinomios de múltiples variables en ACL2. Esta formalización es abstracta, en el sentido de que encapsula un anillo de coeficientes que sirve de base para la construcción de los polinomios y para la verificación de sus propiedades fundamentales. Aparte de las propiedades de anillo se han incluido muchas otras propiedades que han resultado útiles a la hora de razonar sobre los polinomios.
- Se ha incorporado a estos anillos polinómicos una relación de orden inducida a partir de sus propios términos, a los que se ha dotado de un orden lexicográfico. Se ha demostrado que esta relación está bien fundamentada y como subproducto de ello se ha obtenido una inmersión de los polinomios en los ϵ_0-ordinales.
- Se ha obtenido, como caso particular, una implementación ejecutable y verificada de los polinomios de coeficientes racionales que se ha empleado como base para la construcción del algoritmo de Buchberger.
- Se ha formalizado el concepto de ideal polinómico en ACL2 de manera que se pueda plantear el problema de la pertenencia a ideales. Se ha definido la congruencia inducida por un ideal y demostrado sus propiedades más importantes.

- Se han formalizado las relaciones de reducción sobre polinomios en el marco suministrado por las relaciones abstractas. Se ha demostrado que la relación de equivalencia coincide con la congruencia inducida por el ideal y que la reducción es noetheriana respecto al orden de polinomios subyacente. También se han presentado algoritmos para el cálculo de las formas normales y se ha demostrado que los ideales son cerrados bajo ellos.

- Se han formalizado los conceptos necesarios relacionados con las bases de Gröbner. Hemos formalizado el concepto de s-polinomio en ACL2 y demostrado que, bajo ciertas condiciones relacionadas con la reducibilidad de los s-polinomios, la relación de reducción inducida por un conjunto de polinomios es localmente confluente y su clausura de equivalencia se puede decidir comprobando la igualdad de formas normales.

- Se ha construido una implementación ACL2 del algoritmo de Buchberger que cumple con COMMON LISP y para la que se ha demostrado su terminación y corrección parcial. Finalmente, esto ha dado lugar a la obtención de un procedimiento de decisión verificado para el problema de la pertenencia a un ideal.

- Se ha obtenido una implementación completamente evaluable o ejecutable, no sólo del procedimiento de decisión, sino de todas las funciones que han intervenido en el algoritmo de Buchberger: operaciones polinómicas, cálculo de s-polinomios, reducciones, etc.

- A lo largo de la memoria se ha demostrado cómo es posible reutilizar formalizaciones ACL2 realizadas por otros autores para evitar tener que partir de cero en nuestro esfuerzo. Del mismo modo, el resultado de este trabajo es una biblioteca ACL2, algunos de cuyos libros son potencialmente reutilizables para desarrollos posteriores. Creemos que esta forma de trabajar es muy beneficiosa para la comunidad del razonamiento automático y esperamos que se introduzcan mejoras en los mecanismos de reutilización disponibles actualmente en el sistema, así como nuevas bibliotecas que faciliten la labor en un conjunto extenso de dominios de conocimiento.

Es difícil estimar el esfuerzo de desarrollo invertido en este trabajo, máxime cuando se considera que se combina en sus etapas iniciales con el aprendizaje del sistema. Hoy por hoy, los sistemas de razonamiento presentan una curva de aprendizaje pronunciada y ACL2 no es una excepción. No es descabellado concluir que si hubiera que volver a rehacer este trabajo ahora, los

conocimientos adquiridos permitirían cumplir la tarea en un tiempo menor. En la siguiente sección proporcionaremos algunos datos cuantitativos sobre el esfuerzo realizado.

Creemos, que a la vista de estos datos, puede deducirse que una gran parte del esfuerzo ha sido dedicada a la formalización de los polinomios y de sus propiedades. En un informe técnico realizado por Théry a propósito de la formalización del algoritmo de Buchberger en Coq, éste indica:

> When we started, we thought the proof could be carried out in three months. Our first mistake was to understimate the amount of work needed to formalize polynomials and the usual operations.

Sin embargo, el autor no aclara del todo las posibles causas, ni en dicho informe, ni en sus trabajos posteriores [100, 101]. Creemos que el problema radica en que en la mayoría de los textos no se presta la debida atención a la demostración de las propiedades de los polinomios. De hecho, los polinomios son tratados en muchos textos de manera axiomática, como objetos primitivos con una serie de propiedades que se asumen sin más.

Este enfoque es inapropiado para un desarrollo completamente formal en un sistema de razonamiento automático (al menos, si se pretende poder realizar cálculos con ellos) ya que las propiedades y la forma de demostrarlas dependen de la representación interna; dicho de otro modo, hay que poner especial cuidado en que las operaciones definidas sobre la representación interna cumplan las propiedades deseadas.

En este sentido, el comentario que realizan Rudnicky, Schwarzweller y Trybulec en [84], pág. 151, resulta de lo más esclarecedor. Los autores se sorprenden de las dificultades encontradas al demostrar la asociatividad del producto de polinomios de múltiples variables en Mizar, un sistema diseñado específicamente para la formalización del conocimiento matemático:

> We would like to mention that proving the associativity of this convolution product presented a technical challenge as it turned out to be extremely tedious... It is a bit surprinsing that even in a thorough algebra text (Becker and Weispfenning, 1993) the proof is left as an excercise. In MacLane and Birkhoff (1967) the corresponding proof of the univariate case occupies a quarter of a page with half of it relegated to reasoning by analogy.

11.1. Datos cuantitativos

En las tablas 11.1 y 11.2 se presentan algunas cifras sobre la teoría desarrollada. La primera columna contiene el nombre de cada uno de los libros ACL2, tal y como se describe en el capítulo 1. Las tres columnas siguientes pretenden proporcionar una idea de las dimensiones de cada libro. Por cada uno de ellos aparece, en primer lugar, el número de líneas[1] que lo componen, seguido del número de definiciones y de teoremas. La última columna merece especial atención, ya que sus valores representan el número de definiciones y teoremas en los que hemos de suministrar alguna pista o consejo para que el sistema complete su demostración[2] con éxito.

Resulta notable que la mayoría de los consejos suministrados sean para desactivar o activar reglas. Esto ocurre en determinadas demostraciones con las reglas de definición. La desactivación de una regla de definición impide al sistema expandir la definición de la función asociada, con lo que sugerimos al demostrador que no emplee dicha definición, bien porque no es relevante, bien porque no es el momento adecuado en el curso de la demostración o porque existen propiedades importantes que podrían aplicarse de no ser por dicha expansión.

Después de la activación y desactivación de reglas, el consejo más común ha sido sugerir al demostrador introducir como hipótesis en una demostración una instanciación de variables de un teorema previamente demostrado. Cada uno de estos consejos suele ir acompañado de otro que desactiva la regla en cuestión, para evitar que la hipótesis sea eliminada prematuramente por su aplicación.

Otra contribución significativa la representan los consejos de instanciación funcional. Con más de 80 sugerencias de este tipo a lo largo de todo el trabajo, esto no debe entenderse como algo negativo, sino todo lo contrario: cada instanciación funcional representa la reutilización de propiedades más generales demostradas con anterioridad.

En no más de una veintena de ocasiones, se ha sugerido explícitamente al demostrador emplear un esquema de inducción determinado. No en todas ellas ha sido por la imposibilidad de encontrar una demostración siguiendo el esquema sugerido por el demostrador, sino que algunas veces la razón ha sido

[1] Se excluyen aquí los comentarios y las líneas en blanco, que no aportan nada desde el punto de vista lógico pero que tan necesarios son para dotar de legibilidad al código.

[2] En el caso de las funciones, nos referimos por «demostración» a la de su terminación y, cuando proceda, a la verificación de su protección.

simplificar una demostración excesivamente enrevesada. Comparado con el, aproximadamente, medio millar de demostraciones realizadas por inducción, esto da una idea del éxito de las heurísticas de inducción que emplea ACL2.

Menos de una decena de veces hemos tenido que suministrar una medida para la terminación de una función. La demostración de terminación más compleja ha sido, sin duda, la del propio algoritmo de Buchberger. En el cálculo de la clausura de la reducción polinómica y de la forma normal respecto de un conjunto, la terminación depende de la buena fundamentación del orden de polinomios, algo que tampoco es trivial y que hemos tenido que indicar al sistema.

11.2. Experiencia con el sistema

A continuación haremos algunos comentarios generales sobre la experiencia que ha supuesto desarrollar el trabajo descrito en esta memoria y la impresión que hemos obtenido a lo largo de él sobre cuáles son actualmente los puntos fuertes y débiles del razonamiento automático y, en particular, del propio sistema de razonamiento ACL2.

- Existen diferencias muy importantes entre las demostraciones informales o semiformales que aparecen en la mayoría de la literatura científica y una demostración descrita a través de un lenguaje formal que la haga susceptible de ser entendida y analizada por una máquina que, a diferencia de un humano, no es subjetiva, no deja casos sin analizar, nada da por supuesto, nada le parece trivial, etc.
- Estas diferencias son aún más acusadas en el caso de las demostraciones de corrección, terminación y corrección parcial de los algoritmos. Estimamos que esto se debe a varias causas:
  - No se suele emplear un lenguaje o pseudocódigo completamente especificado y dotado de una semántica formal a la hora de describir los algoritmos. Abundan, incluso, las descripciones en lenguaje natural insertas como «instrucciones» de los algoritmos.
  - No es extraño encontrar autores para los que la sola exhibición de un algoritmo basta como demostración o justificación de un teorema de existencia. A menudo se hace referencia a lo «obvia» que resulta su corrección, probablemente por la forma en que se ha construido o presentado.

Libro	Líneas	Def.	Teoremas	Pistas
Directorio `polinomios`				
`coeficiente.lisp`	185	7	36	12
`termino.lisp`	115	7	16	2
`monomio.lisp`	135	14	24	8
`polinomio.lisp`	17	5	1	0
`forma-normal.lisp`	145	8	23	4
`suma.lisp`	46	1	11	3
`congruencias-suma.lisp`	22	0	5	2
`opuesto.lisp`	81	1	13	8
`producto.lisp`	105	5	19	9
`congruencias-producto.lisp`	120	1	18	4
Subtotal	971	49	166	52
Directorio `polinomios-normalizados`				
`polinomio-normalizado.lisp`	167	9	26	16
`orden.lisp`	35	2	6	0
Subtotal	202	11	32	16
Directorio `polinomios-racionales`				
`racional.lisp`	281	9	61	27
`termino.lisp`	137	9	20	3
`termino-division.lisp`	125	3	27	0
`monomio.lisp`	176	16	38	7
`polinomio.lisp`	46	5	1	1
`forma-normal.lisp`	147	7	10	7
`suma.lisp`	63	2	5	5
`congruencias-suma.lisp`	26	0	10	10
`opuesto.lisp`	52	2	3	2
`producto.lisp`	82	5	9	7
`congruencias-producto.lisp`	11	0	4	4
`polinomio-normalizado.lisp`	267	12	44	22
`orden.lisp`	95	3	5	5
`defun-k.lisp`	81	15	0	0
`pertenencia.lisp`	312	3	57	25
`k-polinomio.lisp`	410	18	68	47
Subtotal	2311	109	362	172
Total	3484	169	560	240

Tabla 11.1: Datos cuantitativos: polinomios

Libro	Líneas	Def.	Teoremas	Pistas
Directorio `Buchberger`				
`reduccion-polinomica.lisp`	123	17	7	3
`noetherianidad.lisp`	278	2	35	25
`congruencia-ideal.lisp`	602	18	51	31
`forma-normal.lisp`	88	4	9	4
`calculo-forma-normal.lisp`	188	8	37	18
`ideal.lisp`	152	8	26	8
`s-polinomio.lisp`	54	2	7	5
`buchberger.lisp`	183	12	19	8
`estabilidad-reduccion.lisp`	64	0	10	2
`estabilidad-s-polinomio.lisp`	18	0	1	1
`estabilidad-sufija.lisp`	67	2	14	2
`estabilidad-buchberger.lisp`	95	3	20	2
`confluencia.lisp`	687	15	57	25
`bases-groebner.lisp`	560	17	51	31
`decision.lisp`	26	1	2	2
Total	3185	109	346	167

Tabla 11.2: Datos cuantitativos: algoritmo de Buchberger

  - La programación de los algoritmos introduce detalles que, a menudo, se pasan por alto en la descripción del algoritmo. No es extraño que estos detalles sean importantes desde el punto de vista de la corrección. Programar es sencillo; programar correctamente, no.

- En todo caso, el razonamiento asistido por computador aumenta enormemente nuestra confianza en los algoritmos desarrollados y en sus implementaciones. En gran parte, pensamos que esto es así porque nos obliga a considerar detalles que normalmente no tenemos en cuenta en una demostración informal.

- Sin embargo, el razonamiento automático no garantiza en términos absolutos que no exista posibilidad de error y creemos que éste es un mito del empleo de métodos formales que hay que desterrar en beneficio de la propia disciplina. Hay varias razones para esta afirmación:

  - Los propios sistemas de razonamiento automático pueden contener errores en su lógica interna. No es del todo extraño que los usuarios de un sistema encuentren errores en él que afecten a la validez de sus resultados. Estos errores suelen ser corregidos continuamente entre versiones. La verificación formal de sistemas de razonamiento completos es aún una asignatura pendiente.
  - Los compiladores e intérpretes de los lenguajes en los que están escritos, los sistemas operativos en los que se ejecutan, etc., no están verificados.
  - Las propias máquinas pueden contener errores o sufrirlos por su condición de dispositivo físico, aunque es cierto que cada vez son más fiables.

  De estas tres causas, la primera es para nosotros la más preocupante. Afortunadamente, los otros tipos de fallos suelen provocar disfunciones que se reflejan en un ámbito más amplio que el del sistema de razonamiento que podamos estar ejecutando. Es harto improbable que un fallo aleatorio en una máquina provoque exclusivamente que nuestro sistema nos diga que una conjetura es correcta cuando no lo es. La continua revisión de los resultados obtenidos propia del proceso de desarrollo que se sigue en proyectos grandes de verificación hace esto aún más improbable.

- Aunque se suela emplear el término «razonamiento automático» en el contexto de la verificación de sistemas informáticos complejos, la realidad es que ésta dista aún mucho de ser automática. Proponer la

secuencia de propiedades adecuada para llegar al resultado deseado es la clave del éxito en estos casos, pero suministrar puntualmente sugerencias al sistema de razonamiento resulta imprescindible incluso en los más automatizados. La situación aquí es muy distinta a la que se observa en las lógicas clásicas, donde sí existen sistemas de razonamiento completamente automáticos.

- A diferencia de lo que suele ocurrir con la programación convencional, aquí es necesario preocuparse por otros aspectos de la abstracción. No sólo se trata de realizar una adecuada abstracción operacional, sino también una abstracción de propiedades y de reglas de razonamiento.

- Resulta importante disponer de bibliotecas reutilizables, donde aquí, el término «biblioteca» es de nuevo más amplio que su homónimo en programación. Hay que tener en cuenta que las interfaces de la bibliotecas no sólo están compuestas por las funciones que se desarrollan, sino por las propiedades que se demuestran sobre ellas.

- En cuanto a la curva de aprendizaje inicial de un sistema tipo, como ACL2, creemos que es pronunciada. En nuestra experiencia, consideramos que se debe dedicar al menos un año a adquirir la destreza necesaria para su manejo. Lejos de considerarse una pérdida de tiempo, el estudio detallado del sistema comenzando con ejemplos sencillos debe entenderse como una inversión que producirá sus réditos al abordar problemas más complejos. En nuestro caso particular, la simplicidad de la lógica subyacente presenta, en este sentido, ventajas e inconvenientes:

  - Ventajas a la hora de razonar, ya que no existen reglas de inferencia de especial complejidad.
  - Inconvenientes a la hora de especificar, ya que existe una falta de expresividad intrínseca.

- ACL2, a diferencia de otros demostradores, formaliza un subconjunto de un lenguaje de programación real. Valoramos esto muy positivamente, ya que nos permite ejecutar directamente en el sistema los algoritmos desarrollados. Otros sistemas permiten extraer automáticamente código escrito en un lenguaje de programación real a partir de las especificaciones desarrolladas en su propia lógica. Puesto que los sistemas de extracción no están verificados formalmente, esto añade una incertidumbre extra sobre la validez de las ejecuciones.

- ACL2 posee unas heurísticas magníficas para la automatización de la inducción. Es capaz de encontrar esquemas de inducción apropiados

en un porcentaje muy elevado de las demostraciones por inducción. Creemos que ésta es una característica intrínseca del sistema y que no sólo se aplica a nuestro caso particular.

- Sin embargo, hemos observado varias veces cómo la demostración inductiva de un teorema puede fracasar por ser éste demasiado particular. ACL2 no es responsable de este problema, que es consustancial al método de inducción. Mientras que en la verificación de programas imperativos la invención del invariante es el paso más creativo, en la verificación de programas recursivos la necesidad de generalizar una función o una propiedad para, posteriormente, obtener otra más particular es probablemente la labor más creativa a la que podemos enfrentarnos.

- ACL2 es razonablemente bueno a la hora de demostrar la terminación de funciones sencillas.

- ACL2 no es tan inteligente a la hora de decidir qué funciones expandir y cuándo. No obstante, creemos que esto es disculpable por cuanto es una actividad que también es difícil para un humano. La diferencia estriba en que un humano emplea «prueba y error» guiado por su conocimiento heurístico del problema, mientras que el sistema toma una decisión basada en heurísticas generales y no rectifica su decisión. Es tarea del usuario tener una buena «política» de activaciones y desactivaciones.

- Los mecanismos de abstracción y modularidad en ACL2 son bastante primitivos, lo que se comienza a notar cuando se acumulan unos cientos de teoremas. En particular, no existe una forma cómoda de instanciar funcionalmente propiedades generales. Hemos paliado esta deficiencia con el empleo de macros para el cálculo de los consejos necesarios durante la instanciación funcional.

- El proceso interno de inclusión de libros para su reutilización en ACL2 es bastante lento, sobre todo cuando existe una cadena larga de inclusiones: los libros ya cargados en las etapas anteriores de la cadena, se vuelven a cargar en la etapa siguiente. Nuestro trabajo se encuentra muy estructurado, lo que corresponde con una buena práctica de las técnicas de construcción de sistemas informáticos, sin embargo, esto hace que contenga cadenas muy largas de inclusiones (algunas de más de una veintena de libros) lo que hace que la recertificación de un libro que se encuentre cercano al final de la cadena sea muy lenta.

- La legibilidad de las demostraciones producidas con ayuda de ACL2 resulta normalmente aceptable. Con el sistema se obtiene un documento

en lenguaje natural por cada libro certificado. La diferencia en cuanto a la facilidad de comprensión de una demostración entre ACL2 y muchos de los sistemas que emplean tácticas es muy acentuada. Las tácticas representan un enfoque procedimental, más comprensible para la máquina que para el humano, por lo que es difícil imaginarse la evolución de una demostración sin repetirla paso a paso en el demostrador. Por contra, ACL2 es más declarativo facilitándonos el que nos concentremos en los detalles importantes de las demostraciones y podamos entender la idea general de una demostración: qué esquema de inducción se usa, en qué casos se divide, qué teoremas se aplican, etc.

- Sin embargo, los pasos de demostración pueden ser grandes pudiendo dificultar la comprensión de los detalles concretos. Esto se debe fundamentalmente a dos razones:

  - Los procedimientos de decisión pueden eliminar conjeturas complejas o simplificarlas de manera no trivial para un humano. Esto se nota en la simplificación proposicional, la aritmética lineal y el razonamiento con equivalencias.
  - El empleo reiterado de la reescritura durante la búsqueda de términos estables bajo simplificación, puede eliminar conjeturas completas o simplificarlas notablemente mediante la mera indicación de la lista de reglas empleadas, sin pista alguna de cuáles fueron sus instancias por variables.

  Las dos causas anteriores pueden incluso confluir en un mismo paso de demostración, con lo que la confusión está prácticamente asegurada. Ya que un excesivo nivel de detalle puede resultar igualmente confuso, consideramos que sería interesante que el sistema contara con más mecanismos que permitieran establecer el nivel de detalle deseado para describir una demostración.

- La ausencia de cuantificadores en la lógica de ACL2 nos obliga a ser creativos a la hora de formalizar ciertas nociones que, clásicamente, se definen en términos de cuantificadores. En este sentido, ACL2 tiene deficiencias importantes respecto de otros sistemas. Incluso con el empleo de funciones de Skolem para paliar la ausencia de cuantificadores, la falta de apoyo por parte del sistema a la hora de razonar sobre ellas nos obliga a realizar demostraciones guiadas, disminuyendo el grado de automatización. Esto se nota en nuestro trabajo, sobre todo en la formalización de conceptos relacionados con los ideales.

- La ausencia de un sistema de tipos y el carácter total de las funciones en ACL2, se combinan para producir un efecto muy desagradable: hay veces que es necesario completar las definiciones de las funciones para que comprueben a cada paso si los parámetros son del tipo esperado. Esto puede deberse a dos causas:

  - Conviene evitar la proliferación de hipótesis en el enunciado de los teoremas a demostrar, ya que esto simplifica el razonamiento, sobre todo si es inductivo. Además, ACL2 no permite que las reglas de congruencia posean hipótesis, ya que el sistema de reescritura por congruencias es incondicional; esto nos obliga a comprobar las hipótesis en las propias definiciones de las funciones implicadas.
  - A veces es necesario asegurar que los parámetros permanecen en un cierto dominio en el que podemos garantizar la terminación de la función. Esto nos ha ocurrido al definir la propia función que implementa el algoritmo de Buchberger.

  Puede pensarse que las comprobaciones adicionales de las definiciones son evitables con tal de invertir un mayor esfuerzo de demostración, pero lo cierto es que no siempre es así.

11.3. Trabajo futuro

La investigación presentada en esta memoria puede continuarse siguiendo distintas líneas. Discutimos a continuación algunas de las que consideramos de mayor interés.

- Mejoras en la eficiencia.

  El algoritmo formalizado en esta memoria no es especialmente eficiente y, en este sentido, es susceptible de diversas mejoras. Consideramos que sería interesante estudiar las siguientes posibilidades:

  - Eliminar las comprobaciones sobre los tipos de los objetos en las definiciones, ya que éstas, aunque útiles para razonar en la lógica, suponen una degradación de la eficiencia de la ejecución.[3] Tras haber estudiado el problema detenidamente no pensamos que sea posible hacerlo en nuestro caso, al menos con la versión actual del

[3] Recuérdese que estas comprobaciones se incluyen debido principalmente a las limitaciones de la lógica, que sólo permite definir primitivamente funciones totales y sin tipos.

demostrador. No obstante, está planificada una futura versión de ACL2 que promete incorporar mecanismos que permitirían refinar las contrapartidas ejecutables de las funciones a partir de información suministrada en las protecciones. Sería así posible mantener las definiciones actuales mejorando su rendimiento en ejecución.

- Mejorar computacionalmente los distintos algoritmos implicados. Aunque la verificación de los algoritmos sobre polinomios ha supuesto un esfuerzo importante, podrían emplearse algoritmos más eficientes, a costa, por supuesto, de un esfuerzo de verificación aún mayor. Incluir en el algoritmo de Buchberger criterios verificados de eliminación de s-polinomios ayudaría también a reducir sus requisitos espacio-temporales.

De todos modos, las mejoras descritas tienen sus límites computacionales. En realidad, el problema es de una gran complejidad. Aunque no parece haberse completado todavía el análisis general de la eficiencia del algoritmo de Buchberger, existen numerosos resultados parciales que indican que este algoritmo (incluso una vez aplicados los criterios de eliminación de s-polinomios) puede consumir una cantidad desorbitada de espacio y de tiempo incluso a partir de unos pocos polinomios con pocas variables. De hecho se ha demostrado que el problema de la congruencia para ideales polinómicos es exponencialmente completo en espacio, por lo que el problema de construir una base de Gröbner es intrínsecamente difícil. Véanse al respecto las notas sobre la complejidad del algoritmo que aparecen en [26].

- Generalizaciones.

  Tras los trabajos seminales de Gröbner, Buchberger e Hironaka, otros autores se han afanado en generalizar la noción de base de Gröbner y el propio algoritmo de Buchberger a dominios distintos del de los polinomios con coeficientes en un cuerpo. Existen dos líneas bien diferenciadas:

  - Los trabajos de Kandri-Rody y Kapur, que amplían la teoría para incluir distintos tipos de coeficientes. En [38] se presenta una extensión que permite calcular la base de Gröbner de un ideal polinómico sobre los enteros. En [39] estos resultados se amplían para incluir a los enteros de Gauss y a los polinomios de una sola variable sobre un cuerpo; por último, en [40] se engloban los anteriores resultados en dominios euclídeos.

- Los trabajos en Álgebra no conmutativa, que generalizan la teoría de las bases de Gröbner a álgebras no conmutativas sobre diferentes dominios. Aquí se generalizan todos los elementos de la teoría para incluir versiones por la izquierda y por la derecha. Destacan los trabajos de Kandri-Rodi, Weipsfenning y Kredel sobre anillos de polinomios resolubles [41, 53] y el de Mora sobre cuerpos [74].

Sería interesante estudiar la posibilidad de verificar en ACL2 algunas de estas generalizaciones.

- Aplicaciones.

  Existen numerosas aplicaciones del algoritmo de Buchberger. Dos de ellas nos merecen especial interés:

  - Las aplicaciones a la Lógica en diversos formalismos. Concretamente, los problemas de satisfacibilidad y deducción en lógicas proposicionales multivalentes finitas han sido resueltos en [13, 105] donde además se discuten las aplicaciones del método a distintas lógicas modales. Este método tiene aplicaciones a la verificación de bases de conocimiento, como se describe en [55].
  - Las bases de Gröbner han sido aplicadas con bastante éxito a la automatización del razonamiento geométrico. Basándose en su trabajo previo sobre la demostración de teoremas en el cálculo de predicados de primer orden [42], Kapur describe el empleo de dicho método a la Geometría [43, 44]. Partiendo de estas ideas, Cyrluk, Harris y Kapur desarrollaron el demostrador de teoremas GEOMETER [17], que permite resolver problemas de geometría algebraica. Existe también una formulación equivalente del problema debida a Wu [106].

  Estas aplicaciones se basan en disponer una teoría computacional acerca de la resolubilidad de los sistemas de ecuaciones polinómicas. Sería interesante estudiar la posibilidad de llevar a cabo este desarrollo en ACL2. Esto permitiría disponer de aplicaciones verificadas del algoritmo de Buchberger.

Bibliografía

[1] Baader, F.; Nipkow, T.: *Term rewriting and all that.* Cambridge University Press (1998)

[2] Ballarin, C.: *Computer algebra and theorem proving.* University of Cambridge Computer Laboratory. Informe Técnico 473 (1999)

[3] Barja Pérez, J. M.; Pérez Vega, G.: *Demostración en implementaciones concretas de anillos de polinomios.* Actas del Primer Congreso Anual de la Real Sociedad Matemática Española, págs. 7–25 (2000)

[4] Boyer, R. S.; Moore, J S.: *A computational logic.* ACM monograph series. Academic Press (1979)

[5] Boyer, R. S.; Moore, J S. (eds.): *The correctness problem in computer science.* Academic Press (1981)

[6] Boyer, R. S.; Moore, J S.: *A mechanical proof of the unsolvability of the halting problem.* Journal of the ACM **31**(3):441–458 (1984)

[7] Boyer, R. S.; Kaufmann, M.; Moore, J S.: *The Boyer-Moore theorem prover and its interactive enhancement.* Computers and Mathematics with Applications **29**(2):27–62 (1995)

[8] Boyer, R. S.; Moore, J S.: *NQTHM, the Boyer-Moore theorem prover.* `www.cs.utexas.edu/users/boyer/ftp/nqthm` (1997)

[9] Boyer, R. S.; Moore, J S.: *A computational logic handbook.* Academic Press. 2.ª ed. (1998)

[10] Brock, B.: `defstructure` *for ACL2.* Computational Logic, Inc. (1997)

[11] Buchberger, B.: *An algorithm for finding a basis for the residue class ring of a zero-dimensional polynomial ideal (en alemán).* Universidad de Innsbruck. Tesis doctoral (1965)

[12] Buchberger, B.; Winkler, F. (eds.): *Gröbner bases and applications.* London Mathematical Society Lecture Note Series **251**. Cambridge University Press. (1998)

[13] Chazarain, J.; Riscos, A.; Alonso, J. A.; Briales, E.: *Multi-valued logic and Gröbner bases with applications to modal logic.* Journal of Symbolic Computation **11**(3):181–194 (1991)

[14] Constable, R. L.; Allen, S.; Bromely, H.; Clevely, W. et al.: *Implementing mathematics with the NUPRL development system.* Prentice-Hall (1986)

[15] Coquand, C.: *The interactive theorem prover Agda.* `www.cs.chalmers.se/~catarina/agda` (2000)

[16] Coquand, T.; Persson, H.: *Gröbner bases in type theory.* Types for Proofs and Programs, International Workshop TYPES'98. LNCS **1657**:33–46 (1999)

[17] Cyrluk, D.; Harris, R. M.; Kapur, D.: *GEOMETER: a theorem prover for algebraic geometry.* International Conference on Automated Deduction. LNCS **310**:770–771 (1988)

[18] Davenport, J. H.; Siret, Y.; Tournier, E.: *Computer algebra. Systems and algorithms for algebraic computation.* Academic Press (1988)

[19] Davis, J.; Hart, D.; Wolfe, C.: *Getting started with Maple.* `www.indiana.edu/~statmath/math/maple/gettingstarted` (2001)

[20] Dijkstra, E. W.: *Guarded commands, non-determinacy and formal derivation of programs.* Communications of the ACM **18**(8):453–457 (1975)

[21] Dijkstra, E. W.: *A discipline of programming.* Prentice-Hall (1976)

[22] Dowek, G.; Felty, A.; Herbelin, H.; Huet, G. et al.: *The* Coq *proof assistant reference manual.* INRIA. Informe técnico 0203 (1999)

[23] Dybjer, P.: *Comparing integrated and external logics of functional programs.* Science of Computer Programming **14**(1):59–79 (1990)

[24] Floyd, R. W.: *Assigning meanings to programs.* Symposium in Applied Mathematics. AMS **19**:19–32 (1967)

[25] Gamboa, R.: *Mechanically verifying real-valued algorithms in ACL2.* University of Texas at Austin. Tesis doctoral (1999)

[26] Geddes, K. O.; Czapor, S. R.; Labahn, G.: *Algorithms for computer algebra.* Kluwer Academic Publishers (1992)

[27] Gerhard, J.; Oevel, W.; Postel, F.; Wehmeier, S.: *MuPAD tutorial.* Springer-Verlag (2000)

[28] Girard, J. Y.: *Linear logic and parallelism.* Advanced School on Mathematical Models for the Semantics of Parallelism. LNCS **280**:166–182 (1986)

[29] Gordon, M.; Milner, R.; Wadsworth, C.: *Edinburgh LCF: a mechanized logic of computation.* LNCS **78** (1979)

[30] Gordon, M.; Melham, T.: *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press (1993)

[31] Grayson, D.; Stillman, M.: *Macaulay 2: a software system for algebraic geometry and commutative algebra.* `www.math.uiuc.edu/Macaulay2` (2002)

[32] Greuel, G. M.; Pfister, G.; Schönemann, H.: *Singular: a computer algebra system for polynomial computations.* `www.singular.uni-kl.de` (2002)

[33] Hironaka, H.: *Resolution of singularities on an algebraic variety over a field of characteristic 0, I and II.* Annals of Mathematics **79**(2):109–326 (1964)

[34] Hoare, C. A. R.: *An axiomatic basis for computer programming.* Communications of the ACM **12**:576–580 (1969)

[35] Hsiang, J.; Huang, G. S.: *Some fundamental properties of Boolean ring normal forms.* DIMACS Series on Discrete Mathematics and Computer Science: The Satisfiability Problem. AMS (1996)

[36] Johnson, S. C.: *Sparse polynomial arithmetic.* SIGSAM Bulletin **8**(3):63–71 (1974)

[37] KAMAREDDINE, F. (ed.): *Thirty five years of automating mathematics.* Kluwer Applied Logic. Kluwer Academic Publisher (2003)

[38] KANDRI-RODI, A.; KAPUR, D.: *Computing the Gröbner basis of an ideal in polynomial rings over the integers.* Proceedings of the Third MACSYMA Users Conference, págs. 436–451 (1984)

[39] KANDRI-RODI, A.; KAPUR, D.: *Algorithms for computing Gröbner bases of polynomial ideals over various Euclidean rings.* EUROSAM, International Symposium on Symbolic and Algebraic Computation. LNCS **174**:195–206 (1984)

[40] KANDRI-RODY, A.; KAPUR, D.: *Computing a Gröbner basis of a polynomial ideal over a Euclidean domain.* Journal of Symbolic Computation **6**(1):37–57 (1988)

[41] KANDRI-RODI, A.; WEISPFENNING, V.: *Non-commutative Gröbner bases in algebras of solvable type.* Journal of Symbolic Computation **9**(1):1–26 (1990)

[42] KAPUR, D.; NARENDRAN, P.: *An equational approach to theorem proving in first-order predicate calculus.* Proceedings of the 9th International Joint Conference on Artificial Intelligence, págs. 1146–1153 (1985)

[43] KAPUR, D.: *Using Gröbner bases to reason about geometry problems.* Journal of Symbolic Computation **2**(4):399–408 (1986)

[44] KAPUR, D.: *A refutational approach to geometry theorem proving.* Artificial Intelligence **37**(1–3):61–93 (1988)

[45] KAUFMANN, M.; MOORE, J S.: *Design goals for ACL2.* Computational Logic, Inc. Informe técnico 101 (1994)

[46] KAUFMANN, M.; MOORE, J S.: *An industrial strength theorem prover for a logic based on* LISP. IEEE Transactions on Software Engineering **23**(4):203–213 (1997)

[47] KAUFMANN, M.; MOORE, J S.: *A precise description of the ACL2 logic.* Department of Computer Sciences, University of Texas at Austin (1998)

[48] KAUFMANN, M.; MANOLIOS, P.; MOORE, J S.: *Computer-Aided reasoning: an approach.* Kluwer Academic Publishers (2000)

[49] Kaufmann, M.; Manolios, P.; Moore, J S. (eds.): *Computer-aided reasoning: ACL2 case studies.* Kluwer Academic Publishers (2000)

[50] Kaufmann, M.: *Modular proof: the fundamental theorem of calculus.* En [49], cap. 6 (2000)

[51] Kaufmann, M.; Moore, J S.: *ACL2 version 2.6.* `www.cs.utexas.edu/users/moore/acl2/v2-6` (2001)

[52] Kaufmann, M.; Moore, J S.: *Structured theory development for a mechanized logic.* Journal of Automated Reasoning **26**(2):161-203 (2001)

[53] Kredel, H.: *Solvable polynomial rings.* Shaker (1993)

[54] Kunen, K.: *A Ramsey theorem in Boyer-Moore logic.* Journal of Automated Reasoning **15**(2):217–235 (1995)

[55] Laita, L. M.; Roanes Lozano, E.; Ledesma, L.; Alonso, J. A.: *A computer algebra approach to verification and deduction in many-valued knowledge systems.* Soft Computing **3**(1):7–19 (1999)

[56] Lee, G.; Rudnicki, P.: *Dickson's lemma.* Journal of Formalized Mathematics **14** (2002)

[57] Lombardi, H.; Perdry, H.: *The Buchberger algorithm as a tool for ideal theory of polynomial rings in constructive mathematics.* En [12], cap. 21 (1998)

[58] Luo, Z.; Pollack, R.: *The LEGO proof development system: a user's manual.* Universidad de Edimburgo. Informe técnico ECS-LFCS-92-211 (1992)

[59] Manolios, P.; Vroon, D.: *Algorithms for ordinal arithmetic.* Nineteenth International Conference on Automated Deduction. LNAI **2741**:243–257 (2003)

[60] Martin-Löf, P.: *An intuitionistic theory of types: predicative part.* Proceedings of Logic Colloquium 1973. Studies in Logic and the Foundations of Mathematics **80**:73–118. North-Holland (1975)

[61] Martin-Löf, P.: *Constructive mathematics and computer programming.* Sixth International Congress for Logic, Methodology, and Philosophy of Science **6**:153–175. North-Holland (1982)

[62] Martin, U.; Nipkow, T.: *Ordered rewriting and confluence.* 10th International Conference on Computer Aided Deduction. LNCS **449**:366–380 (1990)

[63] Martín Mateos, F. J.; Alonso, J. A.; Hidalgo, M. J.; Ruiz Reina, J. L.: *A generic instantiation tool and a case study: a generic multiset theory.* Third International Workshop on the ACL2 Theorem Prover and Its Applications (2002)

[64] Martín, F. J.; Alonso, J. A.; Hidalgo, M. J.; Ruiz, J. L.: *A formal proof of Dickson's lemma in ACL2.* Logic Programming and Automated Reasoning. LNAI **2850**:49-58 (2003)

[65] McCune, W.: *Otter: an automated deduction system.* `www-unix.mcs.anl.gov/AR/otter` (2003)

[66] Medina Bulo, I.; Alonso Jiménez, J. A.; Palomo Lozano, F.: *Automatic verification of polynomial rings fundamental properties in ACL2.* ACL2 Workshop 2000. Department of Computer Sciences, University of Texas at Austin. Informe técnico TR–00–29 (2000)

[67] Medina Bulo, I.; Palomo Lozano, F.; Alonso Jiménez, J. A.: *A certified algorithm for translating formulas into polynomials. An ACL2 approach.* International Joint Conference on Automated Reasoning. Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Siena. Informe técnico DII 11/01. (2001)

[68] Medina Bulo, I.; Palomo Lozano, F.; Alonso Jiménez, J. A.: *A certified polynomial-based decision procedure for propositional logic.* 14th International Conference on Theorem Proving in Higher Order Logics. LNCS **2152**:297–312 (2001).

[69] Medina Bulo, I.; Palomo Lozano, F.; Alonso Jiménez, J. A.: *Implementation in ACL2 of well-founded polynomial orderings.* Third International Workshop on the ACL2 Theorem Prover and Its Applications (2002)

[70] Medina Bulo, I.; Alonso Jiménez, J. A.; Palomo Lozano, F.: *Algoritmos polinómicos en ACL2 (una aproximación al algoritmo de Buchberger).* Primer Taller Iberoamericano sobre Deducción Automática e Inteligencia Artificial. Conferencia Iberoamericana de Inteligencia artificial (2002)

[71] Minker, J. (ed.): *Logic and artificial intelligence.* Kluwer (2000)

[72] Moore, J S.; Lynch, T.; Kaufmann, M.: *A mechanically checked proof of the correctness of the kernel of the $AMD5_K86$ floating-point division algorithm.* IEEE Transactions on Computers **47**(9):913–926 (1998)

[73] Moore, J S.: *Towards a mechanically checked theory of computation: a progress report.* En [71], págs. 549–575 (2000)

[74] Mora, T.: *Gröbner bases in non-commutative algebra.* Symbolic and Algebraic Computation: International Symposium. LNCS **358**:150–161 (1989)

[75] Morris, F. L.; Jones, C. B.: *An early program proof by Alan Turing.* Annals of the History of Computing **6**(2):139–143 (1984)

[76] Naur, P.: *Proof of algorithms by general snapshots.* BIT **6**(4):310–316 (1966)

[77] Owre, S.; Shankar, N.; Rushby, J.: *The PVS specification and verification system.* `pvs.csl.sri.com` (2003)

[78] Perdry, H.: *Strongly Noetherian rings and constructive ideal theory.* Journal of Symbolic Computation. Pendiente de publicación (2003)

[79] Persson, H.: *Certified computer algebra.* Lecture Notes for the Types Summer School 99 (1999)

[80] Pottier, L.: *Dickson's lemma.* `ftp://ftp-sop.inria.fr/lemme/Loic.Pottier/MON`

[81] Robbiano, L.: *CoCoA system. Computations in commutative algebra.* `cocoa.dima.unige.it` (2002)

[82] Robinson, J. A.: *A machine-oriented logic based on resolution principle.* Journal of the ACM **12**(1):23–41 (1965)

[83] Rudnicki, P.: *An overview of the Mizar project.* Proceedings of the 1992 Workshop on Types and Proofs for Programs, págs. 311–332 (1992)

[84] Rudnicki, P.; Schwarzweller, C.; Trybulec, A.: *Commutative algebra in the Mizar system.* Journal of Symbolic Computation **32**(1–2):143–169 (2001)

[85] Ruiz Reina, J. L.; Alonso, J. A.; Hidalgo, M. J.; Martín, F. J.: *Formalizing rewriting in the ACL2 theorem prover.* Artificial Intelligence and Symbolic Computation, International Conference AISC. LNCS **1930**:92–106 (2000)

[86] Ruiz Reina, J. L.: *Una teoría computacional acerca de la lógica ecuacional. Formalización en ACL2 de la lógica ecuacional y demostración automática de sus propiedades.* Universidad de Sevilla. Tesis doctoral (2001)

[87] Ruiz, J. L.; Alonso, J. A.; Hidalgo, M. J. y Martín, F. J.: *Formal proofs about rewriting using ACL2.* Annals of Mathematics and Artificial Intelligence **36**(3): 239–262 (2002)

[88] Ruiz Reina, J. L.; Alonso, J. A.; Hidalgo, M. J.; Martín, F. J.: *Termination in ACL2 using multiset relations.* En [37] (2003)

[89] Rushby, J. M.; Shankar, N.; Shrivas, M.: *PVS: combining specification, proof checking and model checking.* Computer Aided Verification, 8th International Conference. LNCS **1102**:411–414 (1996)

[90] Russinoff, D.: *A mechanical proof of quadratic reciprocity.* Journal of Automated Reasoning **8**(1): 3–21 (1992)

[91] Russinoff, D.: *A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7^{TM}processor.* London Mathematical Society Journal of Computation and Mathematics **1**:148–200 (1998)

[92] Russinoff, D.: *A mechanically checked proof of correctness of the AMD K5 floating point square root microcode.* Formal Methods in System Design **14**(1): 75–125 (1999)

[93] Shankar, N.: *A mechanical proof of the Church-Rosser theorem.* Journal of the ACM **35**(3): 475–522 (1988)

[94] Shankar, N.: *Metamathematics, machines, and Gödel's proof.* Cambridge University Presss (1994)

[95] Shoenfield, J. R.: *Mathematical logic.* Addison-Wesley (1967)

[96] Steele, G. L.: *Common Lisp. The language.* Digital Press. 2.ª ed. (1990)

[97] Sustik, M.: *Proof of Dickson's lemma using the ACL2 theorem prover via an explicit ordinal mapping.* Fourth International Workshop on the ACL2 Theorem Prover and Its Applications (2003)

[98] Talcott, C.; Kohlhase, M.: *Database of existing mechanized reasoning systems.* `www-formal.stanford.edu/clt/ARS/systems.html` (1999)

[99] Taub, E. H. (ed.): *John von Neumann: collected works.* Pergamon Press (1963)

[100] Théry, L.: *A certified version of Buchberger's algorithm.* International Conference on Automated Deduction. LNAI **1421**: 349–364 (1998)

[101] Théry, L.: *A machine-checked implementation of Buchberger's algorithm.* Journal of Automated Reasoning **26**(2): 107–137 (2001)

[102] Winkler, F.: *Polynomial algorithms in computer algebra.* Springer-Verlag (1996)

[103] Wolfram, S.: *The Mathematica book.* Wolfram Media. 5.ª ed. (2003)

[104] Wos, L.: *What is automated reasoning?* Journal of Automated Reasoning **1**(1):6–9 (1985)

[105] Wu, J.; Tan, H.: *An algebraic method to decide the deduction problem in in many-valued propositional calculus.* Proceedings of the 24th IEEE International Symposium on Multiple-Valued Logics, págs. 270–273 (1994)

[106] Wu, W.: *Basic principles of mechanical theorem proving in elementary geometries.* Journal of Automated Reasoning **2**(3):221–252 (1986)

Índice de funciones y macros

Índice de teoremas