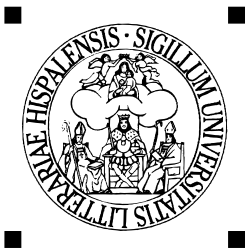

Universidad de Sevilla
Departamento de Ciencias de la Computación
e Inteligencia Artificial

Una teoría computacional acerca de la lógica ecuacional

Formalización en ACL2 de la lógica ecuacional y demostración automática de sus propiedades

Memoria presentada por
José Luis Ruiz Reina
para optar al grado de
Doctor en Matemáticas
por la Universidad de Sevilla

José Luis Ruiz Reina

V. B. Director

D. José Antonio Alonso Jiménez

Sevilla, Junio de 2001

A mis padres, Carmen y José,

y a Inma

Agradecimientos

El trabajo presentado en esta memoria está parcialmente financiado por los proyectos PB96-0098-C04-04 del Ministerio de Educación y Cultura y TIC2000-1368-C03-02 del Ministerio de Ciencia y Tecnología.

En primer lugar, quiero expresar mi agradecimiento a José Antonio Alonso Jiménez, por su dirección y apoyo durante la realización de este trabajo y por haberme hecho descubrir el mundo del razonamiento automático.

A mis colegas del Departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Sevilla, que hacen posible el trabajo diario en un entorno de amistad. En particular, me gustaría mencionar a Miguel Ángel Gutiérrez, Carmen Graciani y Mario Pérez. Y muy especialmente a Francisco Martín Mateos: su colaboración ha sido muy importante para el desarrollo de algunas partes de este trabajo.

Al Grupo de Lógica Computacional de la Universidad de Sevilla. Creo que formamos un grupo de trabajo con muchas posibilidades, tanto en el presente como en el futuro. Las fructíferas discusiones durante los seminarios han influido positivamente en este trabajo.

A J Moore, Matt Kaufmann y Robert Boyer, creadores del sistema ACL2, sin los cuales, evidentemente, este trabajo no hubiera sido posible. Agradezco enormemente su magnífica disposición para responder en todo momento cualquier cuestión relacionada con ACL2. Hago extensivo este agradecimiento a todos los miembros de la lista de correo de ACL2.

A mi familia, por su apoyo y paciencia en todo momento. Muy especialmente a mis padres, Carmen y José, a los que tanto debo por todo lo que me han dado y me siguen dando en la vida. Gracias.

Y finalmente a Inma. En los momentos bajos siempre ha estado ella con una sonrisa y unas palabras de ánimo. Tu cariño me ayuda siempre a seguir adelante.

Índice General

Capítulo 1

Introducción

El razonamiento automático (o deducción automática) constituye uno de los campos de investigación más activos dentro de la inteligencia artificial, lo que queda reflejado en el gran número de sistemas existentes [10, 71] y en las diversas aplicaciones en las que han sido empleados. El objetivo del razonamiento automático es crear programas que resuelvan o ayuden a resolver problemas en los que se requiere algún tipo de razonamiento. Usualmente, se entiende que se trata de un razonamiento en una lógica.

Durante la década de los 60 y de los 70, la mayor parte del trabajo realizado en este campo se centró en el diseño de algoritmos y sistemas basados en la regla de resolución de Robinson [62] para la lógica de primer orden. Se trata de una regla de inferencia que deriva nuevas fórmulas mediante una combinación del algoritmo de unificación y una generalización de la regla de *modus ponens*. Para probar un teorema, los axiomas de la teoría, junto con la negación de la fórmula que se desea probar, se expresan en forma de cláusulas. La regla de resolución se aplica exhaustivamente para intentar llegar a una contradicción, lo que probaría la fórmula.

El trabajo actual en los sistemas basados en resolución se centra en dos aspectos. Por un lado, el estudio de estrategias que permitan reducir el número de cláusulas generadas. Por otro lado, el diseño de implementaciones eficientes (índices, paralelismo, estructuras de datos, etc.). En la actualidad, el máximo representante de los demostradores basados en resolución es Otter [52]. Conviene resaltar que el sistema EQP (una variante de Otter), haya tenido éxito en la demostración de la conjetura de Robbins, un problema hasta entonces abierto en matemáticas [50].

Actualmente, los sistemas basados en resolución no constituyen el principal paradigma para los demostradores automáticos. Las distintas aplicaciones en las que han sido empleados (principalmente en la verificación de sistemas), han hecho emerger la necesidad de usar otras lógicas, distintas de la de primer orden: lógicas de orden superior, modales, lineales, inductivas, constructivas, teoría de tipos, etc.

Entre los sistemas de demostración automática en lógicas de orden superior, destacan HOL [24] y PVS [58]. Otro grupo importante de sistemas es el que automatiza una teoría de tipos constructiva. En este tipo de lógicas de orden superior, los mayores exponentes son Coq [17], Nuprl [14] y Lego [46]. Aunque la expresividad de estas lógicas es superior a la de la lógica de primer orden, la dificultad de automatizar la deducción es mucho mayor. Por ello, estos sistemas son más interactivos que automáticos. Precisamente, ésta es otra propiedad que permite la clasificación los sistemas de demostración automática.

En el otro extremo de los sistemas totalmente automáticos (como Otter), surgen los de-

nominados "comprobadores de pruebas"[1]. En estos sistemas (como Mizar [73]), el usuario construye la prueba y el sistema verifica si verdaderamente se compone de pasos elementales de inferencia. El "estilo LCF" [25], supone un compromiso entre la interactividad y la automatización: las pruebas se construyen con un pequeño núcleo de reglas de inferencia y el usuario puede construir programas (llamados tácticas), que aplican varios pasos básicos de una vez. Estas tácticas se usan principalmente para simplificar expresiones y aplicar procedimientos de decisión. HOL, Nuprl y Coq son demostradores que interactúan con el usuario mediante tácticas.

Otro grupo importante de sistemas de razonamiento automático es el de los demostradores de teoremas inductivos. Este tipo de sistemas está especialmente indicado para razonar sobre estructuras repetitivas: iteración en programas, procesos cíclicos o estructuras de datos definidas por recursión. La principal regla de inferencia de estos sistemas es la inducción y su automatización requiere usualmente la aplicación de heurísticas para descubrir un esquema de inducción adecuado para probar una conjetura. Ejemplos notorios de demostradores inductivos son Nqthm [7], Inka [32] y Oyster/CLaM [11].

En particular, Nqthm (o demostrador de Boyer y Moore) es uno de los demostradores automáticos que más resultados prácticos ha conseguido. La lógica de Nqthm (también conocida como lógica de Boyer y Moore) es una restricción de la lógica de primer orden, sin cuantificadores, y especialmente diseñada para razonar sobre un lenguaje de funciones recursivas, próximo a Lisp. En este sistema, la demostración de los teoremas se realiza, esencialmente, usando simplificación y una potente heurística para realizar demostraciones por inducción. A pesar de la (aparente) debilidad expresiva de su lógica, se ha usado para formalizar resultados no triviales, tanto en matemáticas como en la verificación de sistemas físicos (microprocesadores, principalmente).

Nqthm es un sistema automático, ya que una vez comienza un intento de prueba de una conjetura, el usuario no puede interactuar (excepto para cortar el intento). Sin embargo, el sistema tiene que ser guiado hacia una prueba predeterminada, mediante la demostración previa de una serie de lemas.

Una de las características deseables en cualquier aplicación de los demostradores automáticos, es la posibilidad de ejecutar los modelos formales que se están verificando. Esto aumenta la confianza del usuario en el hecho de que el modelo construido refleje la situación real que se quiere verificar. En algunos sistemas, esta posibilidad de razonar y ejecutar no está presente. En otros, como Coq o Nuprl, el hecho de que la lógica sea constructiva permite extraer, a partir de las pruebas desarrolladas, un programa ejecutable en lenguaje ML [59], lo que permite el desarrollo de programas consistentes con sus especificaciones.

El sistema ACL2

En Nqthm, la lógica está diseñada para razonar sobre funciones de un lenguaje de programación y, por tanto, la ejecución y el razonamiento se realizan en un mismo entorno. Las definiciones de las funciones se usan como axiomas de la lógica y también como reglas de cálculo. Sin embargo, a medida que aumenta la complejidad de los modelos formalizados, puede resultar ineficiente una ejecución de las funciones basada en los axiomas de la lógica.

Esta deficiencia la solventa satisfactoriamente ACL2 [37], creado a partir de 1989 por Boyer, Moore y Kaufmann, como descendiente directo de Nqthm. El sistema ACL2 im-

[1] Del inglés *proof checker*.

plementa una lógica para razonar sobre un subconjunto del lenguaje Common Lisp. Esta lógica es muy similar a la de Boyer y Moore y el demostrador automático usa esencialmente las mismas técnicas que Nqthm (aunque notablemente mejorado). La principal diferencia existente entre ACL2 y Nqthm, radica en que en ACL2 la ejecución de los modelos expresados en la lógica se pueda realizar directamente, bajo ciertas condiciones, en Common Lisp [69]. Es posible ejecutar funciones en Common Lisp con una eficiencia comparable a la de C.

ACL2 también es muy similar a Nqthm en el grado de automatismo y en la manera de guiar al demostrador hacia las pruebas. El usuario debe crear un "mundo lógico", consistente en una serie de resultados previos que el sistema utiliza en forma de reglas. Este mundo lógico debe instruir al demostrador, siguiendo una estrategia inspirada en una prueba informal preconcebida.

Métodos formales y razonamiento automático

En la actualidad, una de las principales aplicaciones de los demostradores automáticos es el uso de métodos formales en la verificación de sistemas *hardware* y *software*, que consiste en especificar el comportamiento esperado de programas o de sistemas físicos, mediante fórmulas lógicas, y usar un demostrador automático para demostrar formalmente estas especificaciones. Frente a una validación basada en la simulación y en comprobaciones, los métodos formales representan una manera rigurosa, consistente y fiable de verificar que un sistema se comporta acorde con su especificación.

HOL, PVS, Nqthm y ACL2, entre otros, se han aplicado con éxito en la verificación de sistemas *hardware*, principalmente microprocesadores. Estos sistemas electrónicos constituyen un campo de aplicación muy adecuado para los demostradores automáticos. Aunque su especificación lógica es relativamente simple, su complejidad combinatoria hace que el razonamiento sólo pueda ser abordado mediante un demostrador automático. La verificación formal de estos sistemas es muy importante, ya que incluso pequeños errores en el diseño de un microprocesador pueden tener serias y desastrosas consecuencias.

Por ejemplo, ACL2 se ha usado con éxito en proyectos como la verificación del microcódigo del algoritmo de división de coma flotante y de raíz cuadrada del procesador AMD K5 [54, 66], o la verificación del código RTL que implementa las operaciones elementales sobre números de coma flotante del procesador AMD Athlon [65].

Otro campo de aplicación de los demostradores automáticos es la demostración de resultados matemáticos, mediante pruebas completamente formalizadas. Por *formalización* entendemos la expresión de los resultados y las demostraciones en un lenguaje formal, con reglas estrictas, tanto sintácticas como semánticas [26]. No es posible llevar a la práctica esta formalización por medios humanos. Sin embargo, con la aparición de los ordenadores, esta tarea parece abordable.

Un trabajo pionero en este sentido fue el proyecto Automath [55], que diseña un lenguaje formal para describir las matemáticas y un programa para comprobar su corrección. La idea no es que el sistema descubra resultados, sino que compruebe las demostraciones. Aunque Automath ha tenido mucha influencia en desarrollos posteriores, actualmente está en desuso.

Otro proyecto con similares objetivos, que sigue desarrollándose con fuerza en la actualidad, es el llevado a cabo con Mizar. El usuario puede escribir textos matemáticos en el lenguaje formal de Mizar y el sistema comprueba su consistencia lógica y la corrección de

las referencias a otros textos escritos en el mismo lenguaje. De esta manera, se desarrolla una base de datos de textos matemáticos formalizados. Actualmente, el proyecto QED [61] trata de extender esta idea, con la intención de construir una biblioteca de conocimiento matemático formalizado.

Junto con el interés teórico que tiene el hecho de formalizar fragmentos de las matemáticas, el desarrollo de una teoría formal con la ayuda de un demostrador automático, supone comprender y analizar los teoremas con mayor detalle, rigor y claridad. Existen ejemplos relevantes de desarrollos formales de resultados no triviales. Harrison [27] desarrolla una teoría formal en HOL acerca de los números reales. Shankar [68] lleva a cabo una demostración automática en Nqthm del primer teorema de incompletitud de Gödel. Se han realizado numerosas pruebas formales de resultados en matemáticas y en lógica en los diferentes sistemas de demostración automática. Véase [33], por ejemplo.

Otro campo de aplicación de los demostradores automáticos, aunque quizá menos desarrollado, es la interacción con los sistemas de calculo simbólico. Esta interacción es deseable por varios motivos. En primer lugar, la resolución de problemas matemáticos requiere a veces la combinación de algoritmos algebraicos con la demostración de teoremas. Por otro lado, los demostradores se pueden usar para aplicar métodos formales al desarrollo de los sistemas de cálculo simbólico, lo que permitiría establecer formalmente teoremas de corrección acerca de los algoritmos usados.

En este sentido, una posible aproximación es la comunicación y cooperación entre sistemas, a través de la construcción de un entorno que combine la capacidad de calculo con la capacidad deductiva [12]. Otra aproximación consiste en programar un demostrador automático dentro de un sistema de cálculo simbólico. Es el caso de Analytica [13], un demostrador de teoremas de análisis elemental, desarrollado en el sistema de cálculo simbólico Mathematica.

Estas aproximaciones no están enfocadas para razonar formalmente sobre la corrección de los algoritmos usados. Thery [72] propone desarrollar algoritmos de cálculo simbólico en demostradores, que permitan tanto el razonamiento sobre los mismos como su ejecución. Como ejemplo relevante, Thery verifica en Coq una versión del algoritmo de Buchberger para la obtención de bases de Gröbner. Como hemos comentado anteriormente, a partir de la demostración constructiva se obtiene un programa en ML, que permite ejecutar el algoritmo.

El sistema ACL2 parece un buen candidato para construir un entorno en el que se pueda combinar el cálculo con la verificación de los algoritmos que lo efectúan. La ejecución de un algoritmo se puede realizar de manera eficiente en el lenguaje Common Lisp, mientras que la formalización de sus propiedades y la demostración de las mismas se puede llevar a cabo en la lógica de ACL2, ayudado por su demostrador automático. Uno de los objetivos fundamentales de esta memoria es el uso de ACL2 para razonar sobre un campo específico: la reescritura de términos.

Ecuaciones y reescritura

Las ecuaciones, como expresión de igualdad entre dos objetos, aparecen con muchísima frecuencia en matemáticas. El razonamiento con ecuaciones también es muy importante en computación: sistemas de cálculo simbólico, verificación de programas, tipos abstractos de datos y demostración automática, por citar algunos ejemplos.

El razonamiento ecuacional se formaliza con la lógica ecuacional. En ella, las sentencias

son ecuaciones entre términos de primer orden. Problemas típicos en la lógica ecuacional son, por ejemplo, determinar si una ecuación es consecuencia lógica de un conjunto de axiomas ecuacionales, buscar aquellos elementos para los cuales se verifica una ecuación (sus soluciones), o determinar si una ecuación es válida en un modelo inicial.

Una regla de reescritura es una ecuación orientada en un determinado sentido. Los sistemas de reescritura de términos son conjuntos de reglas que se usan para reemplazar un subtérmino de un término dado, por otro subtérmino "equivalente": las instancias del lado izquierdo de una regla se sustituyen por la correspondiente instancia del lado derecho. Este proceso se puede aplicar repetidamente hasta obtener, eventualmente, una forma normal en la que no es posible efectuar más reemplazamientos.

La idea de reducción o simplificación ha estado presente siempre en el desarrollo del álgebra. En sistemas de cálculo simbólico como Reduce o Macsyma, se usan las ecuaciones como reglas de reescritura. Los objetos más complejos se transforman para obtener objetos más simples pero equivalentes en cierto sentido. La idea de reducción también está presente en el paradigma de la programación funcional y, por supuesto, en las técnicas de deducción automática: el demostrador automático RRL [34] constituye un buen ejemplo del uso de técnicas de reescritura aplicadas a la demostración automática de teoremas.

Una propiedad deseable en un sistema de reescritura es la confluencia, que permite asegurar que un elemento no se puede reescribir a dos formas normales distintas. Otra propiedad fundamental es su terminación, que garantiza la no existencia de secuencias infinitas de reducción y, por tanto, la existencia de, al menos, una forma normal para cada elemento. Estos conceptos no son exclusivos de los sistemas de reescritura de términos, sino que son aplicables a cualquier tipo de relación de reducción. De la abstracción de estas propiedades surge el estudio de las reducciones abstractas, de las cuales la reescritura de términos es un caso particular.

Los sistemas de reescrituras que son confluentes y que terminan se denominan completos. Un sistema de reescritura completo proporciona un procedimiento de decisión de su teoría ecuacional: para ver si la igualdad de dos términos es consecuencia lógica de las reglas del sistema, basta comprobar si sus formas normales son iguales.

En 1970, Knuth y Bendix [44] propusieron un procedimiento para decidir si un sistema de reescritura que termina es completo o no. Basta comprobar si son iguales las formas normales de un cierto número finito de pares de términos, llamados pares críticos. También diseñan un procedimiento para transformar un sistema que no es completo, en uno equivalente pero completo. Este procedimiento se denomina algoritmo de completación y, esencialmente, consiste en añadir al conjunto de reglas aquellos pares críticos cuyas formas normales no son iguales. A partir de estas propuestas, se inicia una gran actividad investigadora en la materia y, hoy día, la reescritura de términos se puede considerar una importante rama del campo de la deducción automática.

Objetivos de la memoria

En esta memoria se presenta el uso del sistema ACL2 para razonar formalmente sobre la lógica ecuacional y los sistemas de reescritura. Los objetivos de esta memoria son:

- Desarrollar una teoría formal acerca de los términos de primer orden, la lógica ecuacional y la reescritura de términos, usando la lógica de ACL2 y su demostrador automático.

- Producir una biblioteca en ACL2 de resultados básicos acerca de los sistemas de reescritura de términos, que puedan ser reutilizados en posteriores trabajos de verificación.

- Desarrollar una serie de algoritmos ejecutables en Common Lisp, formalmente verificados, como parte de la teoría desarrollada.

Entre los resultados que forman parte de la teoría desarrollada, y que se han demostrado automáticamente en ACL2, destacan:

- Prueba de la estructura de retículo bien fundamentado del conjunto de los términos de primer orden respecto de la relación de subsunción, incluyendo la verificación del algoritmo de unificación de Martelli y Montanari.

- Buena fundamentación de la extensión a multiconjuntos de una relación bien fundamentada.

- Lema de Newman para reducciones abstractas.

- Teorema de pares críticos de Knuth y Bendix.

- Decidibilidad de teorías ecuacionales descritas por sistemas de reescritura completos.

Trabajos relacionados

Aunque no existe ninguna formalización conocida acerca de la lógica ecuacional y los sistemas de reescritura, existen varios trabajos relacionados que han sido desarrollados en otros sistemas:

- Se han realizado demostraciones automáticas de diversas propiedades de reducciones abstractas, la mayoría como parte de formalizaciones acerca del λ-cálculo. Por ejemplo, Huet [31] en Coq, o Nipkow [57] en Isabelle/HOL. Es difícil comparar estos trabajos con la formalización que se presenta en esta memoria, ya que nuestro objetivo es diferente y, sobre todo, las lógicas son muy distintas: la lógica de ACL2 es considerablemente menos expresiva que la de Coq y HOL. Un trabajo más cercano es la demostración del teorema de Church-Rosser para el λ-cálculo realizado por Shankar [67] en Nqthm. Aunque la formalización trata específicamente la reducción del λ-cálculo y no considera el caso abstracto, algunas de las ideas presentes están reflejadas en nuestro trabajo.

- Paulson [60] en LCF y Rouyer [63] en Coq han verificado un algoritmo de unificación de términos de primer orden. En ambos casos, el algoritmo verificado es recursivo en la estructura de los términos. En nuestro caso, hemos verificado el algoritmo de Martelli y Montanari, que está basado en reglas de transformación.

- En [22], Giesl presenta un cálculo para la demostración de teoremas inductivos acerca de funciones parciales. Como ilustración de la potencia de dicho cálculo, en [21] discute una prueba del lema de pares críticos de Knuth y Bendix[2].

[2] No queda claro hasta qué punto la demostración es automática.

Estructura de la memoria

El conjunto de definiciones, lemas y teoremas (*eventos* en general, según la terminología de ACL2) que desarrollan la teoría aquí presentada constituyen la base de esta tesis. Los eventos se estructuran en una serie de ficheros, llamados *libros*. Los libros están *certificados* en ACL2. Es decir, todas las definiciones son admisibles y existe una demostración automática de todos los teoremas.

La última versión de estos libros se puede encontrar en la siguiente página web:

`http://www-cs.us.es/~jruiz/acl2-rewr`

En esta dirección se incluyen instrucciones sobre cómo certificar los libros, así como un enlace a la página de ACL2, donde se encuentra la última versión del sistema.

Descripción de los capítulos

La memoria trata de explicar el desarrollo de la teoría contenida en los libros ACL2 que la acompañan y se estructura en los siguientes capítulos:

Capítulo 2: Una introducción a ACL2

Se describe el sistema ACL2, aportando la suficiente información a fin de que, sin conocimiento previo del sistema, se puedan entender los capítulos restantes. Puesto que ACL2 es a la vez un lenguaje de programación, una lógica para razonar sobre dicho lenguaje y un demostrador automático que ayuda en el razonamiento, el capítulo se divide en tres secciones principales, una por cada "componente" del sistema.

Capítulo 3: Términos y sustituciones

Se especifica la representación elegida para las entidades que van a ser objeto de estudio en la teoría desarrollada: términos, sustituciones y sistemas de ecuaciones (en una signatura).

Se estudia la influencia de esta representación en la definición de las funciones y en la automatización de la demostración de los teoremas de la teoría presentada. Concretamente, se muestra cómo se definen funciones por recursión en la estructura de los términos y cómo estas definiciones recursivas sugieren al demostrador esquemas inductivos que resultan adecuados para la demostración de conjeturas acerca de dichas funciones.

Un aspecto importante en la lógica de ACL2 es que las funciones son totales. En nuestra formalización esto supone que las funciones van a estar definidas tanto para objetos que representan los términos (*términos propios*), como para aquellos que no los representan (*términos impropios*). En este capítulo se explica cómo se aborda esta cuestión y cómo la teoría desarrollada engloba tanto a los términos propios como a los impropios.

El primer conjunto de teoremas demostrados automáticamente formaliza una serie de propiedades de los términos de primer orden relativas a su estructura de árbol.

Un término *subsume* a otro si es posible obtener el segundo mediante aplicación de una sustitución al primero. En este capítulo se formaliza la relación de subsunción entre términos. Para ello se define un algoritmo de *equiparación* que, a partir de un par de términos, encuentra, si existe, una sustitución que aplicada al primero devuelve el segundo. Este algoritmo se describe mediante un conjunto de reglas de transformación

que actúan sobre sistemas de ecuaciones hasta obtener un equiparador. La secuencia de transformaciones se puede aplicar de manera no determinista.

Se detalla la formalización y verificación del algoritmo de equiparación no determinista, así cómo la obtención de un algoritmo determinista, verificado y ejecutable, mediante instanciación funcional. Este algoritmo es la base para definir el preorden de subsunción entre términos, relación básica en el desarrollo de la teoría.

Por último, a partir de la relación de subsunción entre términos, se define la relación de subsunción entre sustituciones y se verifican sus principales propiedades.

Capítulo 4: El retículo de los términos de primer orden

En este capítulo se describe la demostración formal de que el conjunto de los términos de primer orden es un retículo bien fundamentado respecto del preorden de subsunción. La importancia de este resultado teórico radica en que su demostración hace necesaria la definición y verificación de una serie de algoritmos sobre términos de gran utilidad práctica.

En primer lugar, se define la equivalencia (respecto del preorden de subsunción) entre términos de primer orden y se demuestra que esta equivalencia coincide con la operación de *renombrado de variables*. Además se define y verifica un algoritmo específico de renombrado.

A continuación, se describe una demostración en ACL2 de la buena fundamentación del preorden de subsunción en el conjunto de los términos de primer orden.

Se define un algoritmo de *anti–unificación* de términos y se demuestra formalmente que dicho algoritmo obtiene un ínfimo, respecto del preorden de subsunción, de los dos términos que recibe como entrada.

El resultado más importante de este capítulo es la definición y verificación de un algoritmo de *unificación*. Este algoritmo encuentra la sustitución más general que hace iguales a dos términos dados. Como en el caso de la equiparación, el algoritmo se define a partir de un conjunto de reglas de transformación, que se aplican de manera no determinista (el conocido como algoritmo de unificación de Martelli y Montanari). A partir de la versión no determinista, podemos definir y verificar un algoritmo de unificación ejecutable, usando instanciación funcional.

Una vez verificadas las propiedades del algoritmo de unificación, éste se usa para demostrar la existencia del supremo de dos términos (siempre que éstos sean unificables).

Por último, se recopilan todas las propiedades anteriores para demostrar que el conjunto de los términos de primer orden en una signatura es un retículo bien fundamentado (añadiendo un elemento nuevo, mayor que todos los términos).

Capítulo 5: Multiconjuntos y pruebas de terminación

En este capítulo se define una herramienta, muy útil para probar la parada de funciones recursivas. En un principio, se había desarrollado como un resultado auxiliar para la demostración del lema de Newman (capítulo 6). No obstante, resulta lo suficientemente general para que se pueda usar en otros contextos.

Un multiconjunto es un conjunto en el que se permiten "repeticiones" de sus elementos. Si se tiene definida una relación en un conjunto A, es posible definir, a partir de ella, una *relación sobre los multiconjuntos finitos* con elementos de A. Intuitivamente, en esta relación, se obtiene un multiconjunto menor que otro si se eliminan algunos elementos de

éste y se añade una cantidad arbitraria, pero finita, de elementos menores. Se demuestra en ACL2 que esta relación entre multiconjuntos está bien fundamentada si lo está la relación original.

La importancia de este resultado reside en que las relaciones bien fundamentadas entre multiconjuntos se pueden usar para probar la terminación de algunas funciones recursivas cuya parada no es fácil de demostrar. La formalización en ACL2 de este resultado se hace de manera muy general, haciendo posible generar automáticamente relaciones concretas entre multiconjuntos y la demostración de su buena fundamentación.

El uso de esta herramienta se ilustra con las demostraciones de la terminación de dos funciones recursivas: una versión recursiva de cola de la función de Ackermann y una versión iterativa de la función 91 de McCarthy.

Capítulo 6: Reducciones abstractas

Antes de formalizar y razonar sobre las propiedades de los sistemas de reescritura de términos, es conveniente abstraer algunas propiedades y razonar sobre el concepto de *reducción*, como abstracción matemática de cualquier actividad que se asemeje a la progresiva transformación (o reducción) de un objeto, hasta llegar a cierta *forma normal*.

En este capítulo se describe la representación escogida para razonar en ACL2 sobre reducciones abstractas y se formalizan conceptos tales como la relación de equivalencia descrita por una reducción, normalización, confluencia, propiedad de Church-Rosser, noetherianidad y confluencia local. A continuación, se demuestran algunos resultados que resultan de utilidad en el capítulo siguiente.

El primero de estos resultados es la decidibilidad de la relación de equivalencia descrita por una reducción normalizadora con la propiedad de Church-Rosser. El segundo es el conocido como lema de Newman: toda reducción localmente confluente y noetheriana es confluente. Este resultado se prueba usando la herramienta de multiconjuntos construida en el capítulo anterior. Por último, a partir de los dos resultados previos, se demuestra la decidibilidad de la relación de equivalencia descrita por una reducción noetheriana y localmente confluente.

La principal característica de la formalización de las reducciones que se muestra en este capítulo, es que se realiza en un marco muy general, de manera que se pueden usar los resultados descritos, para cualquier reducción concreta. En particular, la reducción de reescritura de términos, como se describe en el siguiente capítulo.

Capítulo 7: Teorías ecuacionales y sistemas de reescritura

En este capítulo se formaliza en ACL2 la lógica ecuacional y su relación con la reescritura de términos. El concepto de consecuencia lógica de un conjunto de axiomas ecuacionales puede ser descrito a partir de la relación de equivalencia generada por una reducción definida en el conjunto de los términos de primer orden: el reemplazamiento de iguales por iguales, usando los axiomas. Ésta es la aproximación que se adopta en este capítulo para definir en ACL2 la teoría ecuacional asociada a un conjunto de axiomas ecuacionales. Una vez definida, se demuestra que es la menor congruencia que contiene a los axiomas.

Se define el concepto de sistema de reescritura, como conjunto de ecuaciones orientadas de derecha a izquierda. Para la reducción asociada a un sistema de reescritura, se definen la noetherianidad (a través del concepto de orden de reducción) y un algoritmo de cálculo de

formas normales, previa definición de un test de reducibilidad. Se verifican las principales propiedades asociadas a estos conceptos.

A continuación se estudia la confluencia de los sistemas de reescritura. El resultado central en la teoría de los sistemas de reescritura se debe a Knuth y Bendix: la confluencia de un sistema de reescritura se reduce al estudio de una cantidad finita de pares de términos, llamados *pares críticos*. En este capítulo se presenta una prueba automática en ACL2 del teorema de pares críticos de Knuth y Bendix.

Teniendo presente que la teoría ecuacional se ha presentado como la equivalencia descrita por una reducción concreta (la de reescritura), se pueden aplicar los resultados del capítulo anterior sobre reducciones abstractas y definir un teorema de decidibilidad para ciertas teorías ecuacionales. En la última sección de este capítulo se prueba en ACL2 que todo sistema de reescritura noetheriano cuyos pares críticos convergen (lo que se conoce como *sistema de reescritura completo*), describe una teoría ecuacional decidible.

Capítulo 8: Conclusiones

En este capítulo se resume brevemente el contenido de la memoria y se presentan algunos datos que permiten cuantificar la teoría desarrollada: número de definiciones y teoremas en cada libro y el número de sugerencias suministradas por el usuario (lo que mide en cierto modo el grado de automatización).

Se aportan algunas ideas acerca de las lecciones aprendidas en el desarrollo de este trabajo y se concluye con las perspectivas de trabajo futuro que se vislumbra tras los objetivos conseguidos en esta memoria.

Apéndices

Se incluyen tres apéndices:

- En el apéndice A, se presentan algunos conceptos matemáticos que se usan en los capítulos de la memoria.
- En el apéndice B se describe la *verificación de las protecciones* de las principales funciones presentadas en la teoría. Este es un proceso clave para aumentar la eficiencia de una función en ACL2, ya que permite su ejecución directamente en Common Lisp. Esencialmente, la verificación de protecciones consiste en demostrar formalmente que cualquier llamada a la función provoca llamadas a funciones dentro de los dominios esperados por las funciones de Common Lisp. Se presentan además unas tablas con datos de tiempos de ejecución de las funciones verificadas.
- En el apéndice C, se comentan algunos detalles adicionales sobre la demostración de los resultados presentados en los capítulos de la memoria. Estos detalles son de carácter técnico, relacionados principalmente con la automatización de las demostraciones, por lo que hemos preferido incluirlos en este apéndice.

Descripción de los libros ACL2

La correspondencia entre los capítulos y los libros ACL2 es la siguiente (todos los ficheros tienen extensión `.lisp`) :

- Capítulo 3:
  - `basic`: definiciones y propiedades básicas acerca de listas, conjuntos y listas de asociación.
  - `terms`: definiciones y propiedades básicas sobre términos, sustituciones y sistemas de ecuaciones; además, teoremas relacionados con la estructura de árbol de los términos.
  - `matching`: definición y verificación de un algoritmo de equiparación no determinista.
  - `subsumption`: definición y verificación del preorden subsunción entre términos.
  - `subsumption-subst`: definición y verificación de la relación de subsunción entre sustituciones.
- Capítulo 4:
  - `renamings`: formalización de la relación de equivalencia entre términos; definición y verificación de un algoritmo de renombrado de variables.
  - `subsumption-well-founded`: demostración de la buena fundamentación de la relación de subsunción.
  - `anti-unification`: prueba de la existencia del ínfimo de dos términos respecto de la relación de subsunción, mediante la definición y verificación de un algoritmo de anti–unificación.
  - `unification-pattern`: definición y verificación del algoritmo de unificación no determinista de Martelli y Montanari.
  - `unification`: definición y verificación de un algoritmo de unificación ejecutable.
  - `mg-instance`: prueba de la existencia de supremo, respecto de la relación de subsunción, de dos términos unificables.
  - `lattice-of-terms`: recopilación de los resultados anteriores para concluir que el conjunto de los términos de primer orden tiene una estructura de retículo bien fundamentado respecto del preorden de subsunción.
- Capítulo 5:
  - `multiset`: prueba de la buena fundamentación de la extensión a multiconjuntos de una relación bien fundamentada.
  - `defmul`: definición de una herramienta para la generación automática de órdenes bien fundamentados entre multiconjuntos.
  - `ackermann`: prueba de la terminación (usando multiconjuntos) de una versión recursiva de cola de la función de Ackermann.
  - `mccarthy-91`: prueba de la terminación (usando multiconjuntos) de una versión iterativa de la función 91 de McCarthy.
- Capítulo 6:
  - `abstract-proofs`: definiciones básicas sobre reducciones abstractas.

  - `confluence`: prueba de la decidibilidad de la relación de equivalencia descrita por una reducción normalizadora con la propiedad de Church-Rosser.
  - `newman`: demostración del lema de Newman.
  - `convergent`: demostración de la decidibilidad de la relación de equivalencia descrita por una reducción noetheriana con la propiedad de Church-Rosser.

- Capítulo 7:
  - `equational-theories`: definición de la teoría ecuacional descrita por un conjunto de ecuaciones.
  - `rewriting`: definición y verificación de un algoritmo de cálculo de formas normales respecto de un sistema de reescritura.
  - `critical-pairs`: demostración del teorema de pares críticos de Knuth y Bendix; definición y verificación de una función que calcula todos los pares críticos de un conjunto de ecuaciones.
  - `kb-decidability`: demostración de la decidibilidad de la teoría ecuacional descrita por un sistema de reescritura completo.

Otro dato de interés puede ser la dependencia entre las teorías desarrolladas en cada libro. En la figura 1.1, presentamos un grafo de dependencias entre los libros desarrollados. Las flechas unen cada libro con los libros donde se incluyen[3].

Como se observa, existen dos libros que engloban a casi toda la teoría desarrollada. Por un lado, la estructura de retículo bien fundamentado en el conjunto de los términos de primer orden (libro `lattice-of-terms.lisp`). Por otro lado, la decidibilidad de las teorías ecuacionales descritas por sistemas de reescritura completos (libro `kb-decidability.lisp`).

Cómo leer esta memoria

Con excepción de los capítulos de introducción y conclusión, y el dedicado a describir ACL2, los restantes capítulos de esta memoria se presentan con un estilo uniforme. Cada sección o subsección está dedicada a formalizar una serie algoritmos o conceptos (los que se han explicado más arriba) y a describir la verificación en ACL2 de los mismos, estructurándose, *grosso modo*, de la siguiente manera:

- Los conceptos y resultados se describen en los preliminares de cada sección, siguiendo la presentación estándar que usualmente aparece en la literatura de los sistemas de reescritura. En nuestro caso, hemos usado como principal referencia el libro de Baader y Nipkow, *"Term Rewriting and All That..."* [1]. A las demostraciones que se presentan en estos preliminares las llamaremos *demostraciones a mano*, en contraposición con las demostraciones automáticas que se han realizado con ACL2[4]. Estos preliminares permiten comparar ambos estilos de demostración y valorar las dificultades surgidas en la formalización correspondiente.

[3] Si la flecha es discontinua, esta inclusión es un evento local al libro (véase la página 55).

[4] Algunos autores las llaman *demostraciones informales*, frente a las demostraciones formales que se obtienen con un demostrador automático o con un comprobador de pruebas.

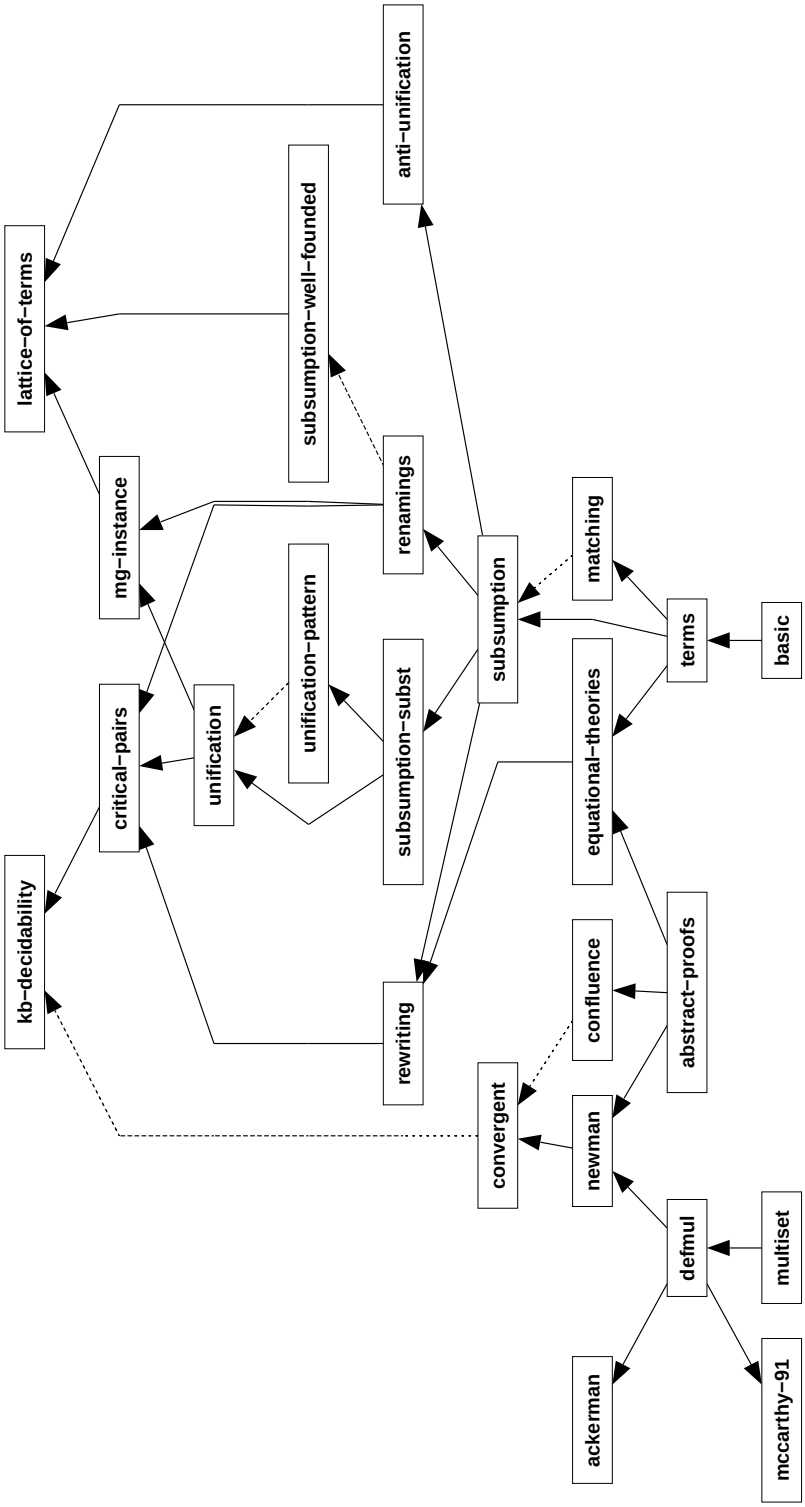

Figura 1.1: Libros ACL2 desarrollados y dependencias entre ellos

- A continuación se describe la definición del concepto o algoritmo en cuestión, y se enuncian sus principales propiedades, usando la lógica de ACL2. De esta manera, se formalizan los teoremas presentados en los preliminares.

- Por último, se describe la demostración de las propiedades anteriores, llevada a cabo en ACL2. Es decir, la secuencia de lemas y definiciones que conducen al demostrador a probar el teorema deseado. Estas descripciones intentan evitar, en la medida de lo posible, comentarios técnicos acerca del uso del demostrador automático y están más centradas en el aspecto lógico de la demostración.

- En algunas de las secciones (o subsecciones), es posible ampliar esta descripción de la demostración, con algunos comentarios más técnicos, relacionados con el demostrador automático. Estos comentarios están recopilados en el apéndice C, en el que supondremos conocido el funcionamiento del sistema.

La estructuración de las secciones de esta memoria permite distintas posibilidades en su lectura, según el nivel de detalle que se desee. En primer lugar, los preliminares de cada sección, junto con la descripción de las funciones y teoremas principales que se han formalizado en ACL2, proporcionan una visión general de la teoría presentada. En segundo lugar, la descripción de las demostraciones permite comprender la estrategia diseñada para abordar la prueba de los resultados de la teoría en la lógica de ACL2. En tercer lugar, los detalles técnicos sobre la demostración automática, que se encuentran en el apéndice C, complementan la descripción anterior y pueden ser de utilidad para el usuario de ACL2.

Los libros ACL2 cuyo desarrollo describe esta memoria, están a su vez profusamente comentados y estructurados en forma de secciones y subsecciones. Se ha intentado que estos comentarios describan paso a paso la estrategia de prueba adoptada en cada uno de ellos. Por tanto, la lectura de los libros supone una información complementaria y detallada.

Para mostrar las definiciones y teoremas formalizados, emplearemos la sintaxis propia de ACL2, que es esencialmente la misma que la de Common Lisp. Usaremos tipo de letra `truetype` para estas expresiones ACL2 y remarcaremos además los resultados más importantes en **negrita**.

Como ya se ha mencionado, sólo las definiciones, teoremas y lemas más importantes aparecen en esta memoria. El resto, necesarios para guiar a ACL2 hacia las demostraciones llevadas a cabo, se pueden consultar en los libros. Para una mayor claridad, hemos despojado a los eventos presentados en la memoria de cualquier tipo de anotación específica de ACL2 (como protecciones, sugerencias al demostrador y tipos de regla). Sólo han sido incluidas en el caso que la explicación lo requiriese.

Capítulo 2

Una introducción a ACL2

Presentamos aquí una introducción al sistema ACL2, haciendo especial énfasis en las características del mismo usadas en esta memoria. No pretendemos dar en este capítulo una descripción detallada, sino proporcionar la suficiente información para que el lector de esta memoria que desconozca el sistema, pueda comprender el resto de los capítulos de la misma.

ACL2 son las siglas de *A Computational Logic for Applicative Common Lisp*. El sistema ACL2 es, a la vez, un lenguaje de programación y una lógica que permite razonar sobre los programas construidos en ese lenguaje. Además, implementa un demostrador automático que proporciona ayuda al usuario del sistema, automatizando, en cierta medida, el razonamiento en la lógica.

El sistema ACL2 es descendiente directo del demostrador automático Nqthm, que fue desarrollado en la Universidad de Texas por Robert Boyer y J Stroother Moore (razón por la cual se conoce también como demostrador de Boyer y Moore), durante los años 70 y 80. Nqthm se ha usado con éxito en una gran cantidad de proyectos de verificación automática relacionados tanto con la formalización de teoremas matemáticos, como con la verificación formal de sistemas *hardware* y *software*. A medida que los proyectos abordados con Nqthm aumentaban su envergadura, se hizo patente la necesidad de adaptar el sistema para poder afrontarlos. En concreto, la principal deficiencia de Nqthm se produce cuando las especificaciones lógicas se usan, además de para razonar sobre el sistema modelizado, para ejecutarlo. Esta ejecución del modelo es importante para corroborar si es fidedigna la formalización respecto del objeto que se quiere formalizar. En Nqthm, la ejecución podría llegar a ser altamente ineficiente, sobre todo cuando el tamaño de la teoría aumentaba, ya que son los axiomas de la lógica los que proporcionan el mecanismo de ejecución.

En 1989, Boyer y Moore escriben la primera versión de ACL2, con la idea de construir una lógica que permitiera razonar sobre un subconjunto de un lenguaje de programación eficiente y de uso extendido: Common Lisp. La lógica de ACL2 es, esencialmente, la misma que la de Nqthm (conocida como lógica de Boyer y Moore), pero se modifica para poder razonar sobre un subconjunto aplicativo de Common Lisp. De esta manera, las funciones definidas en la lógica pueden ser ejecutadas eficientemente en cualquier implementación de Common Lisp.

Matt Kaufmann, autor de Pc-Nqthm (versión interactiva de Nqthm), se involucra también en el proyecto de ACL2. Con el tiempo, la dedicación de Boyer al proyecto decrece y la de Kaufmann aumenta, hasta que Boyer decide dejar de ser considerado coautor del sistema. En la actualidad, la última versión de ACL2 es la 2.5 y puede ser

considerada una obra de J Moore y de Matt Kaufmann.

Lo que sigue es una lista incompleta de teoremas verificados usando tanto ACL2 como Nqthm:

- el Teorema Fundamental del Cálculo [35] ,
- la ley de cuadrados recíprocos de Gauss [64],
- la indecidibilidad del problema de la parada [4],
- el teorema de Church-Rosser para el lambda-cálculo [67],
- el teorema de Ramsey [45],
- el teorema de incompletitud de Gödel [68],
- corrección de un algoritmo de la transformada rápida de Fourier [20].

Como hemos comentado anteriormente, Nqthm y ACL2 se han mostrado muy eficaces en probar la corrección de "sistemas digitales": diseños *hardware*, modelos de microprocesadores, *software*, etc. Por ejemplo, se han verificado:

- la "pila CLI", un sistema académico que incluye microprocesador, enlazador, ensamblador y compilador, y aplicaciones programadas en el lenguaje que acepta el compilador [53],
- 21 de las 22 rutinas de la biblioteca de cadenas de Berkeley C (compiladas para el procesador Motorola 6820) [74],
- programas de microcódigo de la ROM del procesador de señales digitales CAP, de Motorola [9],
- el microcódigo del algoritmo de división de coma flotante y de raíz cuadrada del procesador AMD K5 [54, 66],
- el código RTL que implementa las operaciones elementales sobre números de coma flotante del procesador AMD Athlon [65].

El lector interesado puede encontrar más referencias sobre éstos y otros proyectos de verificación desarrollados con Nqthm y ACL2 en las páginas web de los sistemas ([39] y [5]).

Existen diversas fuentes de información que recomendamos al lector para un estudio detallado de ACL2:

– La mejor introducción a ACL2 es el libro [37]. Esencialmente, nos hemos basado en esta referencia para confeccionar este capítulo. Viene acompañada de un segundo volumen [36] en el que se describen una serie de proyectos de verificación que se han llevado a cabo usando el sistema. En [38] también se proporciona una introducción rápida, explicando su evolución a partir de Nqthm.

– La página web de ACL2 [39] proporciona gran cantidad de información general sobre el sistema. Permite obtenerlo para su instalación y proporciona enlaces a documentos que describen múltiples aspectos de ACL2 y su uso. Además, se pueden encontrar varios tutoriales.

– Es también muy recomendable para el usuario ACL2 consultar la documentación existente sobre Nqthm, ya que, como se ha comentado, la lógica de ACL2 es esencialmente la lógica de Boyer y Moore. Además, las técnicas del demostrador automático de ACL2 son también similares a las de Nqthm. La referencia [6] constituye actualmente la mejor descripción de las técnicas que el demostrador automático emplea. En [7] se hace mayor énfasis en la descripción de la lógica de Boyer y Moore y en cómo formalizar problemas. Asimismo, se describe el método más apropiado para interactuar con el sistema.

– En el aspecto técnico, la información más completa sobre el sistema es el manual de referencia, cuya versión más actualizada se encuentra en [39].

En esta memoria, seguiremos la notación de [37] para hacer referencia a temas desarrollados más extensamente en el manual de ACL2: subrayaremos el término correspondiente, con tipo de letra `truetype`. Por ejemplo, si escribimos defthm estamos implícitamente indicando que el lector puede encontrar más información sobre `defthm` en la correspondiente entrada del manual de referencia.

La teoría formal presentada en esta memoria ha sido desarrollada usando la versión 2.5 del sistema. En lo que sigue se describen someramente el lenguaje de programación, la lógica y el demostrador automático de ACL2. A partir de estas tres facetas del sistema estructuramos este capítulo.

2.1 El lenguaje de programación ACL2

El lenguaje de programación ACL2 es una extensión de un subconjunto de Common Lisp. Este subconjunto se obtiene eliminando de Common Lisp aquellos elementos del mismo que producen efectos colaterales, como las variables globales y las operaciones destructivas. Se puede afirmar que ACL2 es, con algunas extensiones, un subconjunto *aplicativo* (o *funcional*) de Common Lisp. Es decir, las funciones definidas con ACL2 son funciones en el sentido matemático del término: su valor sobre un argumento dado es siempre el mismo y no depende de circunstancias externas a la función. Esta restricción permite simplificar considerablemente el razonamiento formal sobre los programas desarrollados.

ACL2 es un lenguaje de programación funcional eficiente: las funciones ACL2 se pueden compilar en cualquier compilador que siga el estándar Common Lisp. Como muestra más relevante, el mismo sistema ACL2 está completamente programado usando el lenguaje de programación de ACL2. Los modelos formalizados usando su lógica sirven, por tanto, como especificaciones lógicas de los sistema modelados y como instrumentos para la simulación de los mismos.

Aunque a continuación describimos brevemente el lenguaje, supondremos que el lector está familiarizado con el lenguaje Lisp y con su notación prefija. La segunda edición del manual de referencia Common Lisp [69] es actualmente el estándar *de facto* del lenguaje de programación Lisp. Remitimos al lector al mismo si desea una descripción más detallada de las funciones Common Lisp que aparezcan en esta memoria.

Tipos de datos en ACL2

Los *objetos* o *datos* de ACL2 se clasifican en cinco tipos distintos, que ilustramos a continuación con algunos ejemplos:

- Números: `5`, `-2`, `3/4`, `#c(2 1)`.
- Caracteres: `#\a`, `#\Space`.
- Cadenas de caracteres: `"Hola y adios"`.
- Símbolos: `t`, `nil`, `x`, `NWM::a`.
- Pares punteados: `(1 . 2)`, `(i j k)`.

Cada uno de estos tipos de datos se corresponden con el análogo en Common Lisp. Los cuatro primeros tipos se denominan *atómicos*. ACL2 soporta números enteros, racionales y complejos. Los números racionales se escriben como fracción de números enteros. Los números complejos que se manejan en ACL2 son aquellos cuyas partes real e imaginaria son números racionales.

Los símbolos en ACL2 están compuestos por dos cadenas de caracteres: su *nombre de paquete* y su *nombre de símbolo*. Por ejemplo, el símbolo cuyo nombre de paquete es `NWM` y cuyo nombre de símbolo es `a`, se escribe `NWM::a`. El manejo de paquetes en ACL2 para clasificar los símbolos es similar al de Common Lisp. En cada momento existe un paquete actual declarado por el usuario. Es posible obviar el nombre del paquete actual al escribir un símbolo. Por ejemplo, si el paquete actual es `NWM`, el símbolo `a` es lo mismo que el símbolo `NWM::a`. A menos que se especifique lo contrario, el paquete actual es el de nombre `ACL2` y a él pertenecen los símbolos de las funciones primitivas de ACL2.

Los símbolo `t` y `nil` se reservan para representar los valores booleanos, aunque desde el punto de vista de los predicados y condicionales, cualquier objeto distinto de `nil` se considera como representación del valor verdad.

El par punteado es la construcción básica que en ACL2 (y en Common Lisp) permite obtener objetos compuestos. Dos objetos cualesquiera de ACL2 se pueden agrupar para constituir un par ordenado. Por ejemplo, podemos agrupar el número `3` y el símbolo `x` en el par punteado que se nota `(3 . x)`. Los pares puntuados, a su vez, pueden ser componentes de otros pares punteados: `((3 . x) . (a . a))`. Como en Common Lisp, existe una notación especial para ciertos tipos de pares punteados: si un par punteado es de la forma `(`x `. nil)`, entonces lo escribimos como `(`x`)` y si es de la forma `(`x `. (...))`, entonces lo escribimos como `(`x`...)`. Por ejemplo, el par punteado `(1 . (2 . (3 . nil)))` se notará usualmente como `(1 2 3)`. Este tipo de pares ordenados, cuyo componente final es `nil`, se denominan *listas propias* (o simplemente listas) y son la manera más usual de agrupar en ACL2 (y en Lisp) una serie de objetos en uno sólo. El símbolo `nil` representa además a la lista vacía y alternativamente se denota por `()`.

Existe una construcción específica de listas que permite representar, en cierto modo, la asociación de unos datos a otros: las listas cuyos elementos son pares punteados se denominan *listas de asociación*. Por ejemplo, la lista de asociación `((a . 23) (b . 12) (c . 1))` representa la asociación de los valores `23`, `12` y `1`, a los símbolos `a`, `b` y `c`, respectivamente.

Expresiones en ACL2 y su significado

Un programa ACL2 consiste esencialmente en una serie de definiciones que asignan un determinado significado funcional a una serie de símbolos. Por ejemplo, la siguiente definición asocia al símbolo `fun` una función:

```
(defun fun (l)
  (append nil (cons (car (cdr l))
                    (cons (car l) (cdr (cdr l))))))
```

El comando `defun` se usa en ACL2 (lo mismo que en Common Lisp) para asignar una función a un símbolo (`fun` en este caso). La lista `(l)` del ejemplo se denomina *lista de parámetros* de la definición de la función. A continuación aparece una expresión denominada *cuerpo de la función*. Intuitivamente, una *expresión* es un objeto ACL2 dotado de un significado que describiremos más adelante. Pero antes, definamos de manera precisa qué entendemos por "expresión ACL2". Una *expresión simple* ACL2 puede ser:

- un símbolo de variable,
- un símbolo de constante,
- una expresión constante o
- la aplicación de una expresión funcional f, de n argumentos, a n expresiones e_1,..., e_n, notado (`f` e_1 ... e_n`)`.

Por ejemplo, el cuerpo de la definición anterior de `fun` es una expresión simple que tiene un símbolo de constante `nil`, un símbolo de variable `l` y aplicaciones de las funciones `cons`, `car`, `cdr` y `append`. Nótese que las expresiones simples son objetos ACL2.

Una expresión simple tiene un valor definido para cada asignación de objetos ACL2 a las variables de la expresión. Estos valores son a su vez objetos ACL2. Veamos con detalle cada tipo de expresión y sus valores asociados, todo ello respecto a una serie de definiciones de funciones y constantes que supondremos previamente realizadas[1]. El lector familiarizado con Lisp, observará que el modelo de evaluación que se describe es muy similar al de este lenguaje.

- Los *símbolos de constante* son `t`, `nil` y cualquier símbolo declarado mediante el comando `defconst`. Por ejemplo `(defconst *uno* 1)` declara el símbolo constante `*uno*` con valor `1`. Los nombres de los símbolos de constantes definidos con `defconst`, comienzan y terminan con el carácter `*`. El valor de `t` es `t`, el valor de `nil` es `nil` y el valor de un símbolo de constante declarado con `defconst` es el especificado en su definición.

- Un *símbolo de variable* es un símbolo no constante. El valor de una variable respecto de una asignación es el valor que dicho símbolo tiene asignado.

[1] Técnicamente hablando, el concepto de expresión depende de las definiciones de funciones y de constantes que se hayan realizado. Supondremos esta cuestión implícita en lo que sigue.

- Una *expresión constante* es un número, un carácter, una cadena de caracteres o cualquier expresión de la forma `(quote` x`)`, (abreviado por `'`x) donde x es un objeto ACL2. El valor de un número, carácter o cadena de caracteres es él mismo. El valor de `'`x es x.

- Una *expresión funcional* de n argumentos es o bien un símbolo de función de n argumentos, o bien una expresión lambda de la forma `(lambda (`$v_1 \ldots v_n$`)` *cuerpo*`)`, donde *cuerpo* es una expresión en la que las únicas variables libres que ocurren son $v_1 \ldots v_n$. Los símbolos de función pueden ser, a su vez, símbolos de funciones predefinidas en ACL2, o bien funciones definidas previamente con `defun`.

- Finalmente, veamos cuál es el valor de una expresión simple de la forma `(`$f\ e_1 \ldots e_n$`)`, respecto de una asignación de valores a sus variables, donde f es una expresión funcional de n argumentos y $e_1, \ldots, e_n$ son n expresiones simples. Este valor depende de que f sea un símbolo de función predefinido, un símbolo de función definido con `defun` o una lambda expresión. Supongamos que cada e_i se evalúa, respecto de la asignación dada, a un objeto ACL2 c_i.

  Si f es un símbolo de función predefinido, de n argumentos, entonces tiene asociado una función g de aridad n. Esta función hace corresponder un objeto ACL2 a cada combinación de n objetos ACL2. De esta manera, el valor de la expresión `(`$f\ e_1 \ldots e_n$`)` es $g(c_1, \ldots, c_n)$.

  Si f es un símbolo de función definido con `defun`, con lista de parámetros $v_1, \ldots, v_n$ y cuyo cuerpo es la expresión *cuerpo*, entonces el valor de la expresión `(`$f\ e_1 \ldots e_n$`)` es el valor de la expresión *cuerpo* respecto de la asignación que hace corresponder a cada variable v_i el objeto c_i. De la misma manera se define si f es la lambda expresión `(lambda (`$v_1 \ldots v_n$`)` *cuerpo*`)`.

El lenguaje de las expresiones se complica un poco al introducir las *macros*. Las macros proporcionan una herramienta para poder escribir de manera más compacta expresiones que no son simples pero que se "traducen" a una expresión simple. Por ejemplo, sería conveniente expresar la suma de elementos con el símbolo `+`, sea cual sea el número de elementos que se suman. Pero si escribimos `(+ x y)` y `(+ x y z)`, al menos una de las dos no es una expresión simple, ya que los símbolos tienen una aridad única. La solución está en considerar que se dispone de un símbolo de función predefinido `binary-+`, de aridad 2 y que las expresiones anteriores son simplemente abreviaturas de las expresiones simples `(binary-+ x y)` y `(binary-+ x (binary-+ y z))`, respectivamente. Esto se consigue mediante la definición del símbolo `+` como una macro.

El comando `defmacro` permite asociar una macro a un símbolo. Se comporta de manera parecida al comando del mismo nombre en Common Lisp. Si un objeto ACL2 es de la forma `(`$f\ o_1 \ldots o_n$`)` donde f es un símbolo que no define ninguna función, entonces no es ninguna expresión simple. En ese caso, si f está previamente definido con `defmacro`, el objeto anterior se denomina una *aplicación de una macro*. Los objetos ACL2 construidos a partir de expresiones simples y de aplicaciones de macros se denominan *expresiones*. Para obtener el valor en ACL2 de la expresión anterior, se aplica la función que define f sobre las expresiones o_i (no sobre sus valores, sino sobre los mismos objetos o_i) para obtener un objeto ACL2 *val* que sustituye a la expresión anterior. Entonces, el valor de la expresión es el valor de *val*.

En ACL2 existen muchas macros predefinidas. Algunas, como `+`, sirven para poder escribir expresiones con símbolos de función de aridad variable. Otras sirven para poder escribir expresiones que ni siquiera "parecen" expresiones simples. Es lo que en terminología Common Lisp se denominan formas especiales. Las formas especiales más usadas son las que se construyen con las macros `cond`, `case`, `let` y `let*`, cuya sintaxis y expansión a expresiones simples son análogas a las del mismo nombre en Common Lisp, por lo que pasamos por alto su descripción. Remitimos al lector al manual de ACL2 para una descripción precisa de las formas especiales y macros predefinidas en el sistema.

Las consideraciones hasta aquí hechas explican cómo cada expresión ACL2 tiene, respecto de una asignación de objetos ACL2 a las variables que ocurren en la expresión, un valor definido preciso, que a su vez es otro objeto ACL2. Obsérvese que el modelo de evaluación es similar al de Common Lisp, aunque considerablemente simplificado, ya que está restringido a un subconjunto aplicativo del lenguaje.

Un programa ACL2 consiste en una serie de definiciones (de funciones, de constantes, de macros,...), las cuales permiten dotar de significado a determinadas expresiones. La ejecución de un programa consiste en calcular los valores asignados a estas expresiones.

Algunas funciones y macros predefinidas

En las figuras 2.1, 2.2, y 2.3, listamos y explicamos brevemente algunas funciones y macros predefinidas en ACL2. La mayoría de ellas son funciones o macros Common Lisp y su comportamiento en ACL2 es análogo al que tienen en Common Lisp (más adelante damos una explicación más detallada de esta cuestión). Otras son específicas de ACL2.

`(acl2-numberp x)`	Reconocedor de números
`(integerp x)`	Reconocedor de números enteros
`(rationalp x)`	Reconocedor de números racionales
`(zerop x)`	`x=0`
`(zp x)`	`x=0` o no natural
`(= x y)`	Igualdad
`(< x y)`	Menor estricto
`(<= x y)`	Menor o igual
`(> x y)`	Mayor
`(>= x y)`	Mayor o igual
`(+ x y ...)`	Suma
`(* x y ...)`	Multiplicación
`(- x y)`	Resta
`(- x)`	Opuesto
`(/ x y)`	División
`(1+ x)`	Incremento en `1`
`(1- x)`	Decremento en `1`
`(nfix x)`	Conversión a número natural

Figura 2.1: Funciones y macros predefinidas en ACL2 (números)

Como en Common Lisp, existen macros que permiten abreviar una combinación de aplicaciones de las funciones `car` y `cdr`. Por ejemplo `(cadr l)` es una forma abreviada de

`(equal x y)`	Igualdad entre objetos
`(eql x y)`	Igualdad entre objetos
`(if p x y)`	Función If-then-else
`(and p1 p2 ...)`	Conjunción lógica
`(or p1 p2 ...)`	Disyunción lógica
`(implies p q)`	Implicación lógica
`(not p)`	Negación lógica
`(iff p q)`	Equivalencia lógica

Figura 2.2: Funciones y macros predefinidas en ACL2 (lógicas)

`(consp x)`	Reconocedor de pares punteados
`(atom x)`	Reconocedor de objetos atómicos
`(endp x)`	Reconocedor de objetos atómicos
`(cons x y)`	Constructor de pares punteados
`(car x y)`	Primer componente de un par punteado
`(cdr x y)`	Segundo componente de un par punteado
`(list x y ...)`	Lista con los elementos indicados
`(list* x ...z)`	Lista con los elementos y cola final indicados
`(first l)`	Primer elemento de una lista (igual que `car`)
`(rest l)`	Resto de una lista (igual que `cdr`)
`(second l)`	Segundo elemento de una lista
`(third l)`	Tercer elemento de una lista
`(nth n l)`	`n`-ésimo elemento de una lista
`(len l)`	Longitud de una lista
`(true-listp l)`	Reconocedor de listas
`(assoc x l)`	Búsqueda en listas de asociación

Figura 2.3: Funciones y macros predefinidas en ACL2 (listas)

la expresión `(car (cdr l))` y `(cddar l)` abrevia la expresión `(cdr (cdr (car l)))`.

El entorno de programación de ACL2

ACL2 se presenta ante el usuario siguiendo la idea Common Lisp de bucle "lee–evalúa–escribe"[2]. Es decir, lee una expresión introducida por el usuario, la evalúa y escribe el objeto ACL2 obtenido. Las expresiones que se pueden evaluar en el bucle "lee–evalúa–escribe" de ACL2 no pueden contener variables libres, de manera que tienen un valor perfectamente definido. Veamos un ejemplo de interacción con el usuario, en el que se evalúan tres expresiones:

```
ACL2 !>(+ 1 2)
3
ACL2 !>(car '(a b c))
A
```

[2]En inglés, *read–eval–print loop*.

```
ACL2 !>(and (consp '(a b)) (acl2-numberp 3))
T
```

La cadena "`ACL2 !>`" se denomina *"prompt"* del sistema y entre otras cosas, permite conocer el paquete de símbolos actual. El bucle "lee–evalúa–escribe" da un significado especial a expresiones que comienzan con palabras clave que designan comandos en ACL2. Estos son los comandos más usados en el entorno de programación, algunos de los cuales ya hemos visto cómo se usan y para qué. Consúltese el manual de ACL2 para más detalles.

Comandos	**Descripción**
`(defpkg` *paquete* `...)`	Definición de paquetes
`(defconst` *∗simbolo∗* `...)`	Definición de símbolos de constante
`(defun` *f* `...)`	Definición de funciones
`(defmacro` *f* `...)`	Definición de macros
`(ld "`*fichero*`")`	Carga un fichero con definiciones

ACL2 y Common Lisp

Como hemos comentado, el lenguaje de ACL2 es una *extensión* de un *subconjunto aplicativo* de Common Lisp. Veamos qué quieren decir, en este contexto, cada una de estas palabras:

- *Aplicativo:* ACL2 no contempla aquellas características de Common Lisp que tienen efectos colaterales y que podrían hacer que una función ACL2 no se comportara como una función en el sentido matemático.
- *Subconjunto de Common Lisp:* toda función predefinida en ACL2, que también esté predefinida en Common Lisp, actúa, sobre aquellos objetos ACL2 que también son objetos Common Lisp, tal y como se define en el manual de referencia de Common Lisp. Por ejemplo, la expresión `(car '(x y))` se evalúa a `x` en cualquier implementación de Common Lisp y en consecuencia lo mismo ocurre en ACL2.
- *Extensión:* existen algunas funciones predefinidas en ACL2 que no se encuentran en el estándar Common Lisp. Por ejemplo, la función `acl2-numberp` es una función predefinida en ACL2, que no es tal en Common Lisp. Más adelante comentaremos también algo sobre el tratamiento de los multivalores.

Como veremos en la siguiente sección, el valor de una función predefinida en ACL2 viene determinado por los axiomas de la lógica, que definen su comportamiento. Existen axiomas que especifican el comportamiento de aproximadamente 170 funciones que también son funciones predefinidas Common Lisp. Estas funciones Common Lisp son aquéllas del estándar [69] que cumplen los siguientes requisitos:

1. Tienen una semántica aplicativa.
2. No dependen del estado actual del sistema, de parámetros implícitos o de tipos de datos que no sean los tipos de datos de ACL2.
3. Están completamente especificadas, sin ambigüedad e independientes de la máquina.

Aunque Lisp es un lenguaje sin tipos, el estándar Common Lisp especifica el dominio sobre el que están definidas cada una de sus funciones primitivas (su *dominio pretendido*). El valor que toma una función predefinida Common Lisp fuera de su dominio pretendido se deja al arbitrio de la implementación del sistema Common Lisp correspondiente (en general se obtiene un error).

Recuérdese, sin embargo, que al explicar el valor que toman las expresiones en ACL2, se dijo que todo símbolo de función f, predefinido en ACL2 y de n argumentos, tiene asociado una función g de aridad n, que devuelve un objeto *para cada combinación de n objetos*. Si los argumentos de entrada de una función predefinida en ACL2, que también esté predefinida en Common Lisp, son objetos del dominio pretendido, la función ACL2 se comporta como lo haría en Common Lisp. Sin embargo, la función ACL2 también devuelve un valor cuando actúa sobre objetos fuera del dominio pretendido. Es decir, *las funciones ACL2 son funciones totales*.

Mediante un mecanismo lógico, llamado *protección*[3], el usuario de ACL2 puede especificar el dominio pretendido de las funciones que define. Las protecciones de las funciones Common Lisp están predefinidas de antemano y son las que se especifican en el estándar. La *verificación de protecciones* es el proceso de demostrar que una función respeta las protecciones de todas las funciones que usa en su definición, siempre que sus argumentos de entrada respeten la protección de la función. En el apéndice B de esta memoria se discute con más detalle la verificación de protecciones para las principales funciones definidas en la teoría que presenta esta memoria, así como su relación con la eficiencia en la ejecución.

Por último, comentemos algo sobre el uso de *multivalores* en ACL2. Esta es una de las características de ACL2 que extienden a Common Lisp, que se usa con frecuencia en esta memoria. ACL2 no soporta las funciones Common Lisp `values` y `multiple-value-bind` para el manejo de multivalores[4] y en su lugar usa los siguientes mecanismos:

- `(mv` $e_1 \ldots e_n$`)` evalúa cada uno de los e_i y devuelve como valor los n resultados.
- `(mv-let (`$v_1 \ldots v_n$`)` *exp cuerpo*`)` evalúa *exp*, que debe ser una expresión que devuelve un multivalor de n valores y evalúa *cuerpo* respecto de la asignación que asocia cada v_i al i-ésimo resultado devuelto por *exp*.

2.2 La lógica de ACL2

Presentamos en esta sección una introducción a otro de los componentes del sistema ACL2: una lógica que permite razonar sobre las funciones definidas con el lenguaje de programación presentado en la sección anterior. En [41] el lector puede consultar una descripción precisa, detallada y formal de la lógica de ACL2. Aquí sólo comentaremos aquellos aspectos que permitan una mejor comprensión de la teoría presentada en esta memoria.

El siguiente ejemplo, tomado de [37], nos permite ilustrar algunas cuestiones que se discutirán a continuación. Supongamos definida, en el lenguaje de ACL2, una función de un argumento llamada `mergesort`, que ordena listas de números. Supongamos también que queremos comprobar que dicha función es correcta. Para ello debemos verificar que devuelve una lista con los mismos elementos que la de entrada y además ordenada. Centrémonos en la primera de estas propiedades. Una primera aproximación sería comprobarlo

[3] Del inglés *guard*.

[4] Según el manual de referencia de ACL2, debido a que su significado lógico parece difícil de caracterizar.

manualmente ejecutando en el bucle "lee–evalúa–escribe" algunos ejemplos. Así, ejecutaríamos `(mergesort '(4 1 7 3))` y obtendríamos `(1 3 4 7)`, que efectivamente es una permutación de `(4 1 7 3)`, como podemos comprobar fácilmente a mano. Sin embargo, podríamos automatizar en cierto modo este proceso de comprobación, definiendo en ACL2 una función `perm`, que implementara el concepto de permutación mediante un predicado binario. Podríamos entonces ejecutar varios ejemplos con `mergesort` y comprobar con `perm` que el resultado que devuelve es una permutación del dato de entrada. Por ejemplo:

```
ACL2 !>(let ((x '(4 1 7 3)))
         (perm x (mergesort x)))
T
```

Podríamos ejecutar repetidamente esta expresión, cambiando el valor de `x` en la asignación que realiza `let`. En cada uno de esos casos, si la definición de `perm` y de `mergesort` es correcta, comprobaríamos que la expresión `(perm x (mergesort x))` devuelve el valor `t`. Sin embargo, una cantidad finita de comprobaciones de este tipo nunca serviría para concluir que el valor de `(perm x (mergesort x))` es *siempre* `t`, *para cualquier valor de* `x`. Para llegar a esta conclusión, recurrimos a la lógica matemática.

Una lógica matemática viene descrita por un lenguaje de fórmulas, una serie de axiomas y unas reglas de inferencia que permiten derivar nuevas fórmulas a partir de los axiomas. Demostrar un *teorema* consiste en derivarlo a partir de los axiomas usando las reglas de inferencia. Es posible, además, dotar de significado a las fórmulas, de tal manera que los axiomas son verdades aceptadas y las reglas de inferencias obtienen verdades a partir de verdades. Por tanto, los teoremas son verdades en los modelos que dotan de significado a los axiomas y a las reglas de inferencia. En lo que sigue, supondremos al lector familiarizado con todos estos conceptos básicos de lógica matemática.

El lenguaje de la lógica de ACL2 se define a partir de las expresiones definidas anteriormente, que se combinan usando conectivas lógicas proposicionales y el símbolo de igualdad. Las definiciones de las funciones de un programa se pueden ver como axiomas en la lógica de ACL2, estableciendo que ciertas expresiones son iguales a otras. Las reglas de inferencia, que veremos más adelante, son las de una lógica proposicional con igualdad, junto con un principio de inducción. Si a partir de los axiomas y reglas de inferencia probamos, por ejemplo, la fórmula `(perm x (mergesort x)) = t` y asumimos que los axiomas y reglas de inferencia de la lógica reflejan fielmente el modelo de evaluación explicado en la sección anterior, entonces habremos *probado formalmente* que el valor de `(mergesort x)` es una permutación de `x` *para cualquier* asignación de un objeto ACL2 a `x` (siempre que `perm` defina realmente el concepto de permutación).

El ejemplo anterior muestra también un aspecto interesante que merece la pena comentar. La función `perm` se ha definido para poder expresar, mediante una fórmula de la lógica, una de las propiedades que esperamos que cumpla la función `mergesort`. En la sección anterior, las definiciones de función en ACL2 se habían presentado como reglas de cálculo. En esta sección tendremos otro punto de vista: las definiciones son axiomas de la lógica que pueden ser usados para razonar sobre los programas. Algunas especificarán, además, procesos de cálculo (como `mergesort`) y otras servirán para expresar conceptos y propiedades (como `perm`). Estamos usando, por tanto, *el mismo lenguaje* para razonar y para calcular.

La relación entre la lógica de ACL2 y su lenguaje de programación puede explicarse desde dos puntos de vista. El que hemos visto hasta ahora, en el que los axiomas de la

lógica se pueden ver como una especificación formal de los valores que una función debe tomar al interpretarla como regla de cálculo. El otro punto de vista consiste en considerar que tenemos una lógica que puede ser ejecutada. Es decir, es posible demostrar en la lógica, que las expresiones que representan llamadas de funciones sobre objetos concretos son iguales a un determinado objeto ACL2, que es lo que en la sección anterior hemos llamado su valor como expresión.

En lo que sigue, definiremos con más detalle los axiomas y reglas de inferencia de la lógica de ACL2. Como ya hemos señalado, una descripción precisa de la misma es [41]. Puesto que la lógica de ACL2 es muy similar a la lógica de Boyer y Moore (la lógica de Nqthm), recomendamos también consultar los capítulos 2, 3 y 4 de [7].

Una última cuestión, antes de pasar a describir la lógica: cuando en lo que sigue hablemos de teoremas, nos referiremos a fórmulas que se han derivado en la lógica, usando axiomas y reglas de inferencia. Cuando hablemos de "meta–teoremas" nos referiremos a teoremas obtenidos razonando (informalmente) sobre la lógica de ACL2, o a teoremas externos a la lógica.

Una lógica de primer orden sin cuantificadores

ACL2 es una lógica de primer orden sin cuantificadores[5]. Los *términos* de la lógica son las expresiones, tal y como se definieron en la sección anterior. Los *operadores lógicos* son la igualdad $=$ y las conectivas proposicionales habituales: $\neg$, $\vee$, $\wedge$, $\rightarrow$ and $\leftrightarrow$. Una *fórmula atómica* es una igualdad de la forma $e_1 = e_2$, donde e_1 y e_2 son términos. Las *fórmulas* se definen recursivamente de la manera habitual: las fórmulas atómicas son fórmulas y si ϕ_1 y ϕ_2 son fórmulas, entonces $\phi_1 \vee \phi_2$, $\phi_1 \wedge \phi_2$, $\phi_1 \rightarrow \phi_2$, $\phi_1 \leftrightarrow \phi_2$ y $\neg\phi_1$ son fórmulas. Un ejemplo de fórmula es el siguiente:

$$(\texttt{(integerp x)} = \texttt{t}) \wedge (\texttt{(integerp y)} = t) \rightarrow (\texttt{(integerp (+ x y))} = \texttt{t})$$

Puesto que las fórmulas se definen a partir de las expresiones y éstas dependen de las definiciones (de función, constante, etc.) previamente realizadas, el concepto de fórmula se define respecto de una determinada *historia* de definiciones previas. Por ejemplo, un símbolo de función se podrá usar en una fórmula dependiendo de si previamente se ha introducido o no. Más adelante comentaremos este punto con detalle. Por el momento, supondremos que se han realizado una serie de definiciones de símbolos de función, de macros y de constantes, entre los que se encuentran los símbolos predefinidos en ACL2.

Los axiomas y reglas de inferencias que se presentan en esta sección formalizan el hecho de que la lógica de ACL2 es una extensión de primer orden y sin cuantificadores de la lógica proposicional con igualdad.

Los primeros axiomas y reglas de inferencia de la lógica de ACL2 son aquellos que tratan de la parte proposicional. Un único esquema de axioma es necesario para ello:

- *Esquema de axioma proposicional*: $(\neg\phi \vee \phi)$.

Lo que sigue son cuatro reglas de inferencia para la parte proposicional de la lógica:

- *Expansión*: a partir de ϕ_2 se deriva $\phi_1 \vee \phi_2$.

[5] La lógica de ACL2 permite cuantificaciones, pero tal característica no se usa en esta memoria, por lo que omitimos su descripción.

- *Contracción*: a partir de $\phi \vee \phi$ se deriva ϕ.
- *Asociatividad*: a partir de $\phi_1 \vee (\phi_2 \vee \phi_3)$ se deriva $(\phi_1 \vee \phi_2) \vee \phi_3$.
- *Corte*: a partir de $\phi_1 \vee \phi_2$ y de $\neg\phi_1 \vee \phi_3$ se deriva $\phi_2 \vee \phi_3$.

Como se observa, el esquema de axioma y las reglas de inferencia anteriores solo describen propiedades de $\vee$ y de $\neg$. Las fórmulas $\phi_1 \wedge \phi_2$, $\phi_1 \rightarrow \phi_2$ y $\phi_1 \leftrightarrow \phi_2$ las podemos considerar abreviaturas de fórmulas expresadas usando, únicamente, $\vee$ y $\neg$. Por ejemplo, $\phi_1 \wedge \phi_2$ es una abreviatura de $\neg((\neg\phi_1) \vee (\neg\phi_2))$. Además, consideraremos que $e_1 \neq e_2$ es una abreviatura de $\neg(e_1 = e_2)$.

Los tres axiomas siguientes se refieren a la igualdad:

- *Axioma de reflexividad*: $(\texttt{x} = \texttt{x})$.
- *Esquema de axioma de la igualdad respecto a funciones*:

$$[(\texttt{x1} = \texttt{y1}) \wedge \ldots \wedge (\texttt{x}n = \texttt{y}n)] \rightarrow [\texttt{(f x1...x}n\texttt{)} = \texttt{(f y1...y}n\texttt{)}]$$

- *Axioma de igualdad*:

$$[(\texttt{x1} = \texttt{y1}) \wedge (\texttt{x2} = \texttt{y2})] \rightarrow [(\texttt{x1} = \texttt{x2}) \rightarrow (\texttt{y1} = \texttt{y2})]$$

Por último, la regla de instanciación:

- *Instanciación*: A partir de ϕ se deriva $\sigma(\phi)$.

σ representa una sustitución, o asignación de términos a variables. La notación $\sigma(\phi)$ representa el resultado de aplicar la sustitución σ a la fórmula ϕ, lo que significa que cada variable x que ocurre libre en ϕ se sustituye por el término que σ asigna a x. Nótese que esta última regla de inferencia nos permite considerar las variables de una fórmula lógica en ACL2 como *universalmente cuantificadas*.

Los axiomas de ACL2

Las funciones predefinidas en ACL2 están definidas mediante axiomas que especifican su comportamiento. No daremos aquí la lista completa de estos axiomas, puesto que superan la centena. Veamos, sin embargo, algunos de ellos.

Los cuatro axiomas siguientes especifican el comportamiento de las funciones `car`, `cons` y `cdr` y el tipo de dato par punteado:

- `(consp (cons x y))` $=$ `t`.
- `(consp x)` $=$ `t` $\rightarrow$ `(cons (car x) (cdr x))` $=$ `x`.
- `(car (cons x y))` $=$ `x`.
- `(cdr (cons x y))` $=$ `y`.

De manera análoga, se axiomatizan los restantes tipos de datos y funciones primitivas. Como ya hemos comentado en la sección anterior, los axiomas son consistentes con la especificación estándar de las funciones Common Lisp (por lo que respecta al comportamiento en sus dominios pretendidos). También existen axiomas que especifican los valores de estas funciones fuera de estos dominios.

Los cinco axiomas siguientes especifican el comportamiento de los valores booleanos y de las funciones predefinidas `if` y `equal`:

- `t` $\neq$ `nil`.
- `x` $=$ `nil` $\rightarrow$ `(if x y z)` $=$ `z`.
- `x` $\neq$ `nil` $\rightarrow$ `(if x y z)` $=$ `y`.
- `x` $=$ `y` $\rightarrow$ `(equal x y)` $=$ `t`
- `x` $\neq$ `y` $\rightarrow$ `(equal x y)` $=$ `nil`

Los siguientes axiomas especifican las "funciones proposicionales":

- `(not p)` $=$ `(if p nil t)`.
- `(implies p q)` $=$ `(if p (if q t nil) t)`.
- `(iff p q)` $=$ `(if p (if q t nil) (if q nil t))`.

Los símbolos `and` y `or` están definidos como macros (debido a su aridad variable) y abrevian expresiones construidas con `if`. Por ejemplo `(and` p_1 $p_2 \ldots$`)` se expande a `(if` p_1 `(and` $p_2 \ldots$`) nil)` y `(and` p_1`)` es lo mismo que p_1. Análogas consideraciones podemos hacer para `or`. Esto, junto con los axiomas anteriores, significa que todas las funciones proposicionales se pueden expresar mediante `if`, `t` y `nil`.

Una reseña importante que afectará a nuestra notación en lo que sigue. Téngase en cuenta que estas funciones y macros proposicionales (`or`, `and`, `not`, `implies`, `iff`), junto con `equal`, permiten establecer una correspondencia entre las fórmulas y los términos. Por ejemplo, el término `(implies (and p q) (equal r s))` se corresponde con la fórmula (`p` $\neq$ `nil`$\wedge$`q` $\neq$ `nil`) $\rightarrow$ `r` $=$ `s`. Si s es un término y ϕ_s la fórmula obtenida reemplazando las ocurrencias de funciones proposicionales y `equal` por el correspondiente operador lógico, entonces es posible demostrar en la lógica la fórmula $s \neq$ `nil` $\leftrightarrow \phi_s$. Esto nos permite usar el siguiente convenio, abusando del lenguaje: si hablamos de un término t en un contexto donde se espera una fórmula, nos estamos refiriendo en realidad a la fórmula $t \neq$ `nil`. Como veremos en la sección siguiente, el demostrador automático maneja términos en lugar de fórmulas, siguiendo esta correspondencia.

Ordinales en ACL2

Antes de presentar la última regla de inferencia (la de inducción), debemos explicar la noción de ordinal en ACL2. Supondremos al lector familiarizado con el concepto de ordinal. Una buena referencia para el estudio de la teoría de ordinales es [28]. Siguiendo a Goodstein [23], es posible representar de manera constructiva los ordinales menores que ε_0, mediante números naturales y listas. La tabla de la figura 2.4 muestra algunos ejemplos

Ordinal	Objeto ACL2
0	`0`
1	`1`
2	`2`
3	`3`
...	...
ω	`(1 . 0)`
$\omega + 1$	`(1 . 1)`
$\omega + 2$	`(1 . 2)`
...	...
$\omega \cdot 2$	`(1 1 . 0)`
$\omega \cdot 2 + 1$	`(1 1 . 1)`
...	...
$\omega \cdot 3$	`(1 1 1 . 0)`
...	...
ω^2	`(2 . 0)`
...	...
$\omega^2 + \omega \cdot 5 + 7$	`(2 1 1 1 1 1 . 7)`
...	...
ω^3	`(3 . 0)`
...	...
ω^ω	`((1 . 0) . 0)`
...	...
$\omega^\omega + \omega^{85} + \omega^3 \cdot 2 + 5$	`((1 . 0) 85 3 3 . 5)`
...	...
$\omega^{(\omega^2)}$	`((2 . 0) . 0)`
...	...
$\omega^{(\omega^\omega)}$	`(((1 . 0) . 0) . 0)`
...	...

Figura 2.4: Ordinales en ACL2

de la correspondencia entre ordinales (expresados en forma normal de Cantor) y objetos ACL2 que los representan:

Como se observa, los números naturales son ordinales ACL2. Un objeto de la forma `(`o_1 o_2 ... o_n `.` n`)` es un ordinal ACL2 si a su vez los objetos o_i son ordinales ACL2 distintos de `0`, en orden decreciente y n es un número natural. Intuitivamente, los o_i se corresponden con las potencias de ω cuando el ordinal está expresado en forma normal de Cantor. Los coeficientes (naturales) en la forma normal de Cantor se traducen aquí en repeticiones de las potencias. El número n es el "término independiente" de su forma normal. En ACL2 se define una función `e0-ordinalp` para reconocer aquellos objetos que representan ordinales. El siguiente axioma especifica el comportamiento de dicha función:

```
(e0-ordinalp x)
  =
```

```
(if (consp x)
    (and (e0-ordinalp (car x))
         (not (equal (car x) 0))
         (e0-ordinalp (cdr x))
         (or (atom (cdr x))
             (not (e0-ord-< (car x) (cadr x)))))
    (and (integerp x) (>= x 0))))
```

La función `e0-ord-<` define el orden entre objetos ACL2 que representan ordinales, implementando el orden usual entre ordinales:

```
(e0-ord-< x y)
  =
(if (consp x)
    (if (consp y)
        (if (e0-ord-< (car x) (car y))
            t
            (if (equal (car x) (car y))
                (e0-ord-< (cdr x) (cdr y))
                nil))
        nil)
    (if (consp y)
        t
        (< (if (rationalp x) x 0)
           (if (rationalp y) y 0)))))
```

Es posible demostrar, como meta–teorema, que `e0-ord-<` es una relación bien fundamentada sobre el conjunto de objetos ACL2 que representan ordinales. Según el teorema A.11, esto significa que no es posible obtener una cadena infinita descendente (respecto de `e0-ord-<`) de objetos ACL2 que verifiquen `e0-ordinalp`. Asumir esto es fundamental para la lógica de ACL2, como veremos en la regla de inducción y en el principio de definición. Este meta–teorema es inmediato si se admite que los objetos que verifican `e0-ordinal-p` representan a los ordinales menores que ε_0.[6]

Relacionado con lo expuesto en esta subsección, uno de los axiomas de ACL2 especifica el comportamiento de una función predefinida llamada `acl2-count`, que asocia un ordinal (más concretamente un número natural) a cada objeto ACL2 (véase `acl2-count`), midiendo en cierto modo su tamaño. Así, el tamaño de un par punteado es la suma de los tamaños de sus dos componentes. El tamaño de un número depende de si es entero, racional o complejo. Por ejemplo, el tamaño de un número entero es su valor absoluto. El tamaño de un carácter es 0 y el de una cadena de caracteres su longitud.

El principio de inducción

Veamos ahora la última regla de inferencia de la lógica de ACL2: el principio de inducción. Este principio está basado en el principio de inducción bien fundamentada y su corrección

[6] Es posible, en cualquier caso, hacer una prueba de este meta–teorema sin hacer mención a conexión alguna con los ordinales. Véase `proof-of-well-foundedness`.

queda justificada por la buena fundamentación de `e0-ord-<` sobre los objetos que verifican `e0-ordinal-p` (véase el teorema A.11):

- *Principio de inducción*: La fórmula ϕ se deriva a partir de las siguientes fórmulas:
  - ◇ *Caso base*:
    `(implies (and (not` q_1`) ... (not` q_k`))` ϕ`)`
  - ◇ *Casos de inducción*: Para cada $1 \leq i \leq k$,
    `(implies (and` q_i
    $\sigma_{i,1}(\phi)$
    ...
    $\sigma_{i,h_i}(\phi)$`)`
    ϕ`)`

  donde $q_1, \ldots, q_k$ son términos, $\sigma_{i,j}(1 \leq i \leq k, 1 \leq j \leq h_i)$ son sustituciones y las siguientes fórmulas son teoremas, para cierto término m:
  - ◇ `(e0-ordinalp` m`)`
  - ◇ Para cada i, j tales que $1 \leq i \leq k$ y $1 \leq j \leq h_i$,
    `(implies` q_i `(e0-ord-<` $\sigma_{i,j}(m)$ m`))`

  Decimos entonces que ϕ se demuestra por *inducción en las variables* del término m, denominado *medida*.

La idea es que para probar una fórmula ϕ mediante el principio de inducción, dividimos el problema en $k + 1$ casos. Cada uno de estos casos viene descrito por una condición expresada por el término q_i (casos inductivos), o bien por la negación de todos los q_i (caso base). Para la demostración de cada uno de los casos inductivos se permite asumir que determinadas instancias de la fórmula ϕ son ciertas. En concreto, para probar el caso correspondiente a q_i se suponen ciertas un número h_i de instancias $\sigma_{i,j}(\phi)$, $1 \leq j \leq h_i$. Cada una de estas instancias se denomina una *hipótesis de inducción*. Para que este esquema de demostración sea correcto, se debe demostrar previamente que cada una de las hipótesis de inducción asume que la fórmula ϕ es cierta para elementos cuya medida (dada por una expresión m que toma valores que verifican `e0-ordinalp`) es menor respecto de la relación bien fundamentada `e0-ord-<`.

Supongamos, por ejemplo, que queremos demostrar la fórmula

```
(equal (long (append l1 l2)) (+ (long l1) (long l2)))
```

donde `long` es una función definida que calcula la longitud de una lista (véase su definición en la sección siguiente). Podemos aplicar el principio de inducción para demostrar esta conjetura. Por ejemplo, si $i = 1$, $h_1 = 1$, $q_1 =$ `(consp l1)`, $\sigma_{1,1} = \{$`l1` $\mapsto$ `(cdr l1)`$\}$ y la medida m viene definida por la expresión `(acl2-count l1)`, entonces la fórmula anterior se tendrá si se prueban las siguientes fórmulas:

```
(implies (not (consp l1))
         (equal (long (append l1 l2)) (+ (long l1) (long l2))))
```

```
(implies (and (consp l1)
              (equal (long (append (cdr l1) l2))
                     (+ (long (cdr l1)) (long l2))))
         (equal (long (append l1 l2)) (+ (long l1) (long l2))))

(e0-ordinalp (acl2-count l1))

(implies (consp l1) (e0-ord-< (acl2-count (cdr l1)) (acl2-count l1)))
```

La primera fórmula es la correspondiente al caso base y la segunda, al único caso de inducción. La tercera y cuarta fórmulas se corresponden con las conjeturas acerca de la medida.

Merece la pena destacar en este punto varios aspectos interesantes del principio de inducción recién expuesto:

- La ausencia de cuantificadores en la lógica de ACL2 hace que el principio de inducción no se pueda formular con hipótesis de inducción del tipo "*para todo y menor que* x, la fórmula ϕ es cierta para y". Esto supondría una cuantificación universal en el antecedente de una implicación (es decir, una cuantificación existencial). En lugar de esto, necesitamos especificar un cantidad *finita* de instancias de la fórmula en cada caso de inducción.

- La medida ordinal m que justifica la inducción debe ser *la misma* en todos los casos de inducción.

- En la descripción del esquema de inducción se considera como caso base la conjunción de la negación de cada uno de los casos inductivos. En la práctica este caso base se puede dividir en varios subcasos, a los que llamaremos también casos base.

Teorías desarrolladas en la lógica de ACL2

Decimos que una fórmula ϕ se puede *demostrar (o probar) directamente* a partir de un conjunto de axiomas Ax si se puede derivar a partir de ellos aplicando las reglas de inferencia anteriormente descritas: las del cálculo proposicional con igualdad, la regla de instanciación y el principio de inducción. En principio, el único conjunto Ax de axiomas de que disponemos es el de los axiomas primitivos descritos anteriormente. Sin embargo, si queremos usar la lógica de ACL2 para formalizar sistemas, ésta no puede verse como un conjunto "estático" de axiomas. Debemos ser capaces de definir nuevos conceptos y de añadir axiomas que expresen propiedades sobre estos conceptos.

Por ello, se definen los llamados *principios de extensión*. Una teoría en la lógica de ACL2 "evoluciona" bajo el control de un usuario que hace uso de estos principios de extensión con la ayuda del demostrador automático, como veremos en la sección siguiente.

Los principios de extensión usados con mayor frecuencia por los usuarios de ACL2 para construir sus teorías son dos:

- El *principio de definición*, que permite introducir como axiomas las definiciones de nuevas funciones. Como veremos en la sección siguiente, el comando `defun` permite al usuario del sistema ACL2 hacer uso de este principio.

- El *principio de encapsulado*, que permite introducir nuevas funciones definidas parcialmente, asumiendo sobre ellas determinadas propiedades, que se introducen como axiomas. El comando `encapsulate` permite al usuario de ACL2 hacer uso de este principio.

La aplicación de un principio de extensión se denomina *evento*. Existen más principios de extensión en la lógica de ACL2, como por ejemplo la definición de constantes o la introducción de un axioma arbitrario. En [41] y en el manual de referencia de ACL2, el lector puede obtener una descripción detallada sobre los principios de extensión de ACL2 y los eventos correspondientes. Cada uno de los principios de extensión tiene asociados:

- Un conjunto de símbolos de función introducidos por el evento, con una aridad determinada. Por ejemplo, el principio de definición introduce el símbolo definido. El principio de encapsulado introduce los símbolos nuevos que aparecen en los axiomas asumidos.

- Un conjunto de axiomas introducidos por el evento. Por ejemplo, si aplicamos el principio de definición para definir `f` con `(defun f` $(x_1 \ldots x_n)$ *cuerpo*`)`, se introduce el axioma `(f` $x_1 \ldots x_n$`)` $=$ *cuerpo*. Con el principio de encapsulado se introducen los axiomas asumidos.

Una *historia* h es una secuencia finita de eventos tal que h es vacía, o es el resultado de añadir al final de una historia h' un evento *admisible* E respecto de h'. En este último caso, podremos notar a la historia h como h', E. Para cada principio de extensión se define el criterio de admisibilidad respecto de una historia. Por ejemplo, el principio de definición requiere, entre otros requisitos, que la función definida termine. El principio de encapsulado requiere que los axiomas asumidos se verifiquen, al menos, para ciertas funciones testigo.

El concepto de historia representa intuitivamente la construcción de una teoría por un usuario ACL2. Inicialmente se parte de un lenguaje con los símbolos predefinidos y de un conjunto de axiomas primitivos, descritos anteriormente. Con cada evento, se amplía el lenguaje con símbolos nuevos y se añaden nuevos axiomas. Tiene sentido, por tanto, definir el concepto de teorema respecto de una historia determinada, lo que hacemos a continuación.

Las fórmulas de una historia son la expresiones construidas usando los símbolos de función introducidos por cada uno de sus eventos, junto con los símbolos de funciones predefinidas. Los axiomas de una historia son los axiomas introducidos por cada uno de sus eventos, junto con los axiomas primitivos. Decimos entonces que una fórmula ϕ de una historia h es un *teorema respecto de* h si ϕ se puede demostrar directamente a partir de los axiomas de h.

En el demostrador automático, la prueba de un teorema se intenta mediante el comando `defthm`. El desarrollo de una teoría en la lógica de ACL2 consiste en la aplicación de principios de extensión a partir de la teoría base, construyendo así una historia. Intercalados con estos eventos (`defun` o `encapsulate`) están los teoremas respecto de la historia vigente en cada momento (usando `defthm`).

Excepto el principio de extensión correspondiente a la inclusión de un axioma arbitrario, el resto de principios de extensión tienen la propiedad de preservar la consistencia de

la teoría que extienden. Por tanto, si una teoría se construye mediante principios de extensión distintos de la adición de axiomas arbitrarios, podemos asegurar que es consistente, siempre que lo sea la teoría base.

En el sistema, la adición de un axioma arbitrario se lleva a cabo mediante el comando `defaxiom`. No se recomienda su uso, a menos que se quiera correr el riesgo de que la teoría construida sea inconsistente. La teoría desarrollada en esta memoria no usa `defaxiom`.

A continuación vamos a describir con más detalle los principios de definición y de encapsulado que, como ya hemos señalado, son los principios de extensión más comúnmente usados.

Principio de definición

El principio de definición nos permite añadir axiomas que definen nuevas funciones. A la hora de admitir como axioma una definición de función, debemos tener cuidado de que no introduzca inconsistencias en la teoría desarrollada.

Por ejemplo, supongamos que definimos una función f mediante el axioma `(`f `x) = y`. Si permitiéramos añadir este axioma a una teoría, podríamos derivar, por ejemplo, el teorema `t=nil`, ya que por instanciación se deduciría que `(`f `1) = nil` y que `(`f `1) = t`. Por tanto, el principio de definición debe incluir entre sus requisitos de admisibilidad el prohibir las variables libres en las definiciones.

No necesariamente la introducción de una inconsistencia aparece, como en el ejemplo anterior, por la existencia de variables libres en las definiciones. Obsérvese el siguiente axioma: `(`f `x) = (1+ (`f `x))`. Este axioma contradice a la fórmula `i` $\neq$ `(1+ i)`, que puede ser demostrada a partir de los axiomas primitivos. Nótese que el axioma anterior es un caso típico de definición recursiva que no termina. En consecuencia, otro de los requisitos para que una aplicación del principio de definición sea admitida es que defina una función cuya terminación, para cualquier dato de entrada, se tenga asegurada.

Antes de introducir de manera precisa el principio de definición, debemos definir el concepto de "influencia" que, intuitivamente, expresa las condiciones que se pueden asumir como ciertas cuando se produce la llamada de una función que aparece como subtérmino de una expresión. Más concretamente, decimos que un término t *influye* en la ocurrencia de un subtérmino s de otro término e si, o bien e contiene un subtérmino de la forma `(if` t p q`)` y la ocurrencia de s en e está en p, o bien t es de la forma `(not` t'`)`, e contiene un subtérmino de la forma `(if` t' p q`)` y la ocurrencia de s en e está en q. Por ejemplo, consideremos la expresión `(if p (if (if q a r) r b) c)`. En la primera ocurrencia de `r` en esta expresión influyen tanto `p` como `(not q)`. En la segunda ocurrencia influyen `p` y `(if q a r)`.

En lo que sigue describimos el principio de definición de la lógica de ACL2. Es decir, especificamos el criterio de admisibilidad respecto de una historia dada y el símbolo de función y el axioma que introduce. Usaremos el símbolo $=_{def}$ para describir una definición introducida por el principio de definición.

- *Principio de definición*: Dada una historia h, la definición

  `(`f $x_1 \ldots$ x_n`)` $=_{def}$ *cuerpo*

  es admisible respecto de h si:

  - f es un símbolo de función nuevo (es decir, no aparece en el lenguaje de h),

- ◇ cada x_i, $1 \leq i \leq n$, es un símbolo de variable distinto,
- ◇ *cuerpo* es un término en el lenguaje de h ampliado con el símbolo f de aridad n, cuyas variables libres están entre las x_i,
- ◇ existe un término m en el lenguaje de h (que llamaremos *medida de terminación*) respecto del cual es posible demostrar, en h, las siguientes *conjeturas de terminación*:
  - ∗ `(e0-ordinalp` m`)`.
  - ∗ Por cada ocurrencia en *cuerpo* de un subtérmino de la forma `(`$f\ u_1 \ldots u_n$`)` (es decir, por cada *llamada recursiva*) la siguiente fórmula:
    `(implies (and` $t_1\ \ldots\ t_k$`)`
    `(e0-ord-<` $\sigma(m)\ m$`))`
    donde $t_1, \ldots, t_k$ son los términos que influyen en dicha ocurrencia y σ la sustitución $\{x_1 \mapsto u_1, \ldots, x_n \mapsto u_n\}$.

Si es admisible, el evento correspondiente a la definición anterior amplía h de la siguiente manera:

- se introduce en el lenguaje un nuevo símbolo de función f de aridad n y
- se añade el axioma `(`$f\ x_1 \ldots x_n$`)` $=$ *cuerpo*.

Como ilustración, considérese el siguiente ejemplo de definición de una función `replace-list` que reemplaza el `n`-ésimo elemento de `l` por `x`:

```
(replace-list l n x)
  =def
(if (endp l)
    nil
  (if (zp n)
      (cons x (cdr l))
    (cons (car l) (replace-list (cdr l) (- n 1) x)))))
```

Esta definición se admite respecto de cualquier historia h en la que `replace-list` no se haya definido previamente. En particular, tomando como medida de terminación el término `(acl2-count l)`, se verifican todas las conjeturas de terminación. Nótese que en la llamada recursiva `(replace-list (cdr l) (- n 1) x)` los términos que influyen son `(not (endp l))` y `(not (zp n))`. Con esas condiciones, es fácil demostrar que se verifica `(e0-ord-< (acl2-count (cdr l)) (acl2-count l))`. Nótese que pueden existir medidas de terminación distintas para las cuales se verifiquen las conjeturas. En este caso, por ejemplo, sería posible justificar la admisión usando también una medida sobre el argumento `n`.

Los requisitos para la admisión de una definición garantizan que, extendiendo una teoría mediante el principio de definición, se preserva la consistencia de la misma. Más aún, se puede demostrar que la extensión es *conservativa*, de tal manera que no es posible demostrar teoremas nuevos sobre las funciones previamente definidas. Véase [40] para una desmostración rigurosa y precisa de esta afirmación.

Como se ha mencionado anteriormente, el usuario de ACL2 aplica el principio de definición mediante el comando `defun`. De esta manera, `defun` juega un doble papel, ya que

a la vez que permite definir una función en el lenguaje de programación de ACL2, introduce el axioma correspondiente en la lógica, comprobando previamente la admisibilidad del evento[7]. La medida de terminación la puede suministrar el usuario o bien es escogida mediante una heurística. Daremos más detalles sobre esta cuestión en la siguiente sección.

Encapsulados

El principio de encapsulado permite introducir determinados símbolos de función, sin especificar completamente la función que representan, sino asumiendo sólamente una serie de propiedades que los definen parcialmente. Para que una aplicación del principio de encapsulado sea admisible respecto de una historia y se preserve la consistencia, ha de demostrarse previamente que existen funciones (llamadas "testigos locales") que verifican las propiedades que se quieren asumir como axiomas. Una vez probado esto, las definiciones correspondientes a los testigos locales se pueden "olvidar" y los símbolos de función introducidos por el principio de encapsulado quedan parcialmente especificados, únicamente mediante las propiedades asumidas.

Por ejemplo, supongamos que tenemos una historia h y que queremos asumir un axioma ϕ que expresa una determinada propiedad sobre un símbolo de función nuevo f de aridad n. El principio de encapsulado permite añadir ϕ como axioma siempre que sea posible extender la historia h mediante la aplicación de un evento que defina f por aplicación del principio de definición y en esa historia extendida sea posible *demostrar ϕ como teorema*. Si este es el caso, entonces diremos que dicha aplicación del principio de encapsulado es admisible respecto de h y el evento correspondiente introducirá f como símbolo de función de aridad n y ϕ como único axioma acerca de f. Definimos a continuación el principio de encapsulado de manera más precisa.

- *Principio de encapsulado*: Dada una historia h, el evento

  $Asumir\ \phi\ acerca\ de\ las\ funciones\ f_1, \ldots, f_n.$

  es admisible respecto de h si:

  - ◇ Cada uno de los f_i, $1 \leq i \leq n$, es un símbolo de función nuevo, es decir, no aparecen en el lenguaje de h.
  - ◇ ϕ es una fórmula en el lenguaje de h, ampliado con los símbolos f_i, cada uno de ellos de aridad n_i, respectivamente.
  - ◇ existen eventos D_1,..., D_n, tales que cada D_i es el evento correspondiente a una aplicación del principio de definición para el símbolo f_i con aridad n_i y se verifica que $h' = h, D_1, \ldots, D_n$ es una historia y ϕ un teorema en h'.

  Si es admisible, el evento anterior amplía la teoría de h de la siguiente manera:

  - se introduce en el lenguaje los n símbolos de función f_i con aridad n_i, $1 \leq i \leq n$,
  - se añade el axioma ϕ.

[7] Siempre que se use en modo lógico, lo que ocurre por defecto. Es posible usar `defun` en modo programa de manera que no se aplica el principio de definición y no se incluye axioma alguno en la teoría. En este caso, no se realiza comprobación de admisibilidad (véase `defun`).

Las definiciones introducidas por los eventos D_i se denominan *testigos locales* del principio de encapsulado. Nótese que los testigos locales sólo se usan para asegurar la admisibilidad del encapsulado, ya que los axiomas correspondientes a las definiciones locales no se conservan. El axioma introducido por el evento correspondiente a un encapsulado preserva la consistencia de la teoría que se extiende. Véase [40] para una demostración rigurosa de esta afirmación.

Supongamos, por ejemplo, el siguiente evento:

Asumir `(implies (consp l) (member (sel l) l))` *acerca de* `sel`

Esta aplicación del principio de encapsulado es admisible y permite introducir un símbolo `sel` asumiendo que representa a una función que selecciona un elemento de su argumento de entrada, siempre que éste sea una lista no vacía. Para su admisión, podemos usar el testigo local `(sel l)` $=_{def}$ `(car l)`. El usuario de ACL2 puede aplicar el principio de encapsulado mediante el comando `encapsulate`. Así, este ejemplo se corresponde con el siguiente comando:

```
(encapsulate

 ((sel (lst) t))

 (local (defun sel (lst) (car lst)))

 (defthm sel-selects
   (implies (consp l) (member (sel l) l))))
```

La primera línea es una descripción de la aridad de las funciones que se introducen (denominada *signatura*). En este caso, se declara que la función `sel` es de aridad uno, con un argumento de salida. Los testigos locales se definen usando `defun`, pero declarados como locales mediante `local`. Los axiomas introducidos se declaran mediante `defthm`. Para que este evento sea admitido, las definiciones locales deben ser admisibles y los axiomas declarados con `defthm` demostrados en una teoría en la que las definiciones locales están activas. Fuera del alcance de un `encapsulate` sólo los teoremas no locales se asumen como ciertos.

Las funciones introducidas mediante un encapsulado *no se pueden ejecutar*, ya que su especificación parcial puede que no sea suficiente como para poder deducir el valor que toma para cualquier dato de entrada.

El comando `encapsulate` es bastante más genérico que el presentado aquí y es una potente herramienta que permite el desarrollo estructurado de teorías. En los sucesivos capítulos de esta memoria se verán más ejemplos de uso de `encapsulate`, ya que se usa repetidas veces para construir la teoría desarrollada.

Instanciación funcional

Supongamos que como consecuencia de una aplicación del principio de encapsulado, se ha introducido un axioma ϕ acerca de un símbolo de función f. Entonces parece razonable afirmar que para cualquier teorema ψ acerca de f y cualquier función g que cumpla una propiedad "análoga" a ϕ , existe un teorema "análogo" a ψ acerca de g.

Definamos de manera un poco más precisa qué entendemos por "análogo" en este contexto. Si θ es una fórmula en la que aparece el símbolo f y k es un símbolo de función[8] con la misma aridad que f, la fórmula $\theta[f \mapsto k]$ es la obtenida a partir de θ reemplazando f por k.

Con esta notación, la idea anterior queda expresada como sigue: si ϕ es el axioma introducido por el principio de encapsulado acerca de un símbolo de función f y ψ es un teorema acerca de f, podemos concluir $\psi[f \mapsto g]$, para cualquier función g tal que $\phi[f \mapsto g]$ sea un teorema. Esta afirmación es un caso particular de una regla de inferencia derivada, llamada *instanciación funcional*.

Decimos que la instanciación funcional es una regla de inferencia *derivada* porque es posible probar (como meta–resultado) que si una fórmula se obtiene por instanciación funcional en una historia h, entonces existe una demostración de la misma en h usando únicamente las reglas de inferencia de la lógica proposicional con igualdad, instanciación e inducción.

La regla de instanciación funcional es mucho más general que la descrita aquí. Queda fuera de los objetivos de este capítulo introductorio una descripción detallada y precisa de la misma, que puede consultarse en `lemma-instance`, en `constraint` y en [3]. En la siguiente sección veremos algo más sobre la regla derivada de instanciación funcional, al discutir el consejo `functional-instance`.

2.3 El demostrador automático de ACL2

En esta sección describimos someramente el tercer pilar sobre el que descansa el sistema ACL2: un demostrador automático para la lógica descrita en la sección anterior. Este demostrador proporciona asistencia al usuario en el desarrollo de teorías y en la demostración de teoremas de las mismas.

El demostrador es automático en el sentido de que una vez comienza un intento de demostración, no existe ninguna posibilidad por parte del usuario de interactuar. Sin embargo, es un demostrador interactivo en un sentido más intuitivo. El punto de vista que más se ajusta a la realidad es el que contempla al demostrador como un "asistente" que permite verificar que una determinada fórmula es realmente un teorema. Pero debe quedar claro que la estrategia para abordar una demostración no trivial debe ser diseñada por el usuario, en la mayoría de los casos inspirada en una demostración realizada a mano previamente.

Esta estrategia se le comunica al demostrador mediante la construcción de lo que llamaremos un *mundo lógico*. Éste se construye mediante la demostración previa de una serie de teoremas o lemas, que el demostrador interpreta en forma de *reglas* y que dirigen su comportamiento al intentar abordar la prueba automática de un teorema. Si una demostración de una fórmula se culmina con éxito, ésta se codifica en forma de regla, incrementando la información disponible en el mundo lógico y permitiendo su uso en demostraciones de sucesivos teoremas.

Aunque se trata de automatizar la lógica descrita en la sección anterior, por obvias razones prácticas el demostrador no aplica las reglas de inferencia elementales que se han descrito, sino que construye demostraciones que dan pasos de deducción mucho mayores, aunque seguros. Por ejemplo, es posible probar que toda tautología de la lógica propo-

[8] Podría ser también una lambda expresión.

sicional es un teorema en la lógica de ACL2. Esto permite usar métodos automáticos alternativos para comprobación de tautologías, en lugar de tratar de obtener cada vez la secuencia de axiomas y reglas de inferencia elementales que llevan a una demostración detallada de las mismas. De la misma manera, podemos demostrar una fórmula por distinción de casos. O hacer uso de reemplazamiento de iguales por iguales. Es decir, si se ha demostrado que $e_1 = e_2$, entonces es correcto reemplazar, durante un intento de demostración, cualquier ocurrencia de una instancia de e_1 por la correspondiente instancia de e_2.

Nuestro objetivo en lo que sigue será dar una introdución de las siguientes ideas:

- El mundo lógico como conjunto de reglas que influyen en el comportamiento del demostrador a la hora de abordar un intento de demostración automática de una conjetura.
- Algunos de los procesos y heurísticas que aplica el demostrador durante un intento de demostración.
- Y por último, y quizá lo más importante, cómo debe ser la interacción entre el usuario y el demostrador para conseguir, entre ambos, demostrar formal y automáticamente propiedades no triviales.

2.3.1 El funcionamiento del demostrador

Un ejemplo de demostración automática

Puede ser ilustrativo mostrar la salida devuelta por el demostrador ante un intento de prueba de un teorema que termina con éxito. Supongamos que queremos demostrar que la longitud de la concatenación de dos listas es la suma de sus longitudes. Definimos en primer lugar las funciones `long` y `app`, que implementan, respectivamente, el cálculo de la longitud y la concatenación de listas[9]:

```
(defun long (l1)
  (if (atom l1)
      0
    (1+ (long (cdr l1)))))

(defun app (l1 l2)
  (if (atom l1)
      l2
    (cons (car l1) (app (cdr l1) l2))))
```

Estas definiciones son admitidas, en base al principio de definición. Más adelante comentaremos esta cuestión, pero téngase en cuenta que no sólo el comando `defthm` es el que invoca al demostrador automático: el comando `defun`, en el caso de definición de funciones recursivas, necesita demostrar las correspondientes conjeturas de terminación para que una función sea admitida.

El comando `defthm` permite iniciar un intento de prueba, (bien desde el bucle "lee–evalúa–escribe" del sistema o cargándolo desde un fichero):

[9] Las funciones `len` y `append` son primitivas ACL2 análogas. Aunque en la práctica no necesitaríamos definir `long` y `app`, lo hacemos aquí para mayor claridad en la exposición.

```
ACL2 !>(defthm long-app
         (equal (long (app l1 l2))
                (+ (long l1) (long l2))))
```

Este comando produce la siguiente salida por parte del demostrador, de manera completamente automática:

```
Name the formula above *1.

Perhaps we can prove *1 by induction.  Three induction schemes are
suggested by this conjecture.  Subsumption reduces that number to two.
However, one of these is flawed and so we are left with one viable
candidate.

We will induct according to a scheme suggested by (APP L1 L2).  If
we let  (:P L1 L2) denote *1 above then the induction scheme we'll
use is
(AND (IMPLIES (AND (NOT (ATOM L1)) (:P (CDR L1) L2))
              (:P L1 L2))
     (IMPLIES (ATOM L1) (:P L1 L2))).
This induction is justified by the same argument used to admit APP,
namely, the measure (ACL2-COUNT L1) is decreasing according to the
relation E0-ORD-< (which is known to be well-founded on the domain
recognized by E0-ORDINALP).  When applied to the goal at hand the above
induction scheme produces the following two nontautological subgoals.

Subgoal *1/2
(IMPLIES (AND (NOT (ATOM L1))
              (EQUAL (LONG (APP (CDR L1) L2))
                     (+ (LONG (CDR L1)) (LONG L2))))
         (EQUAL (LONG (APP L1 L2))
                (+ (LONG L1) (LONG L2)))).

By the simple :definition ATOM we reduce the conjecture to

Subgoal *1/2'
(IMPLIES (AND (CONSP L1)
              (EQUAL (LONG (APP (CDR L1) L2))
                     (+ (LONG (CDR L1)) (LONG L2))))
         (EQUAL (LONG (APP L1 L2))
                (+ (LONG L1) (LONG L2)))).

This simplifies, using the :definitions APP and LONG, primitive type
reasoning and the :rewrite rules CDR-CONS and COMMUTATIVITY-OF-+, to

Subgoal *1/2''
(IMPLIES (AND (CONSP L1)
              (EQUAL (LONG (APP (CDR L1) L2))
```

```
                    (+ (LONG L2) (LONG (CDR L1)))))
         (EQUAL (+ 1 (LONG (APP (CDR L1) L2)))
                (+ (LONG L2) 1 (LONG (CDR L1))))).

But simplification reduces this to T, using linear arithmetic, primitive
type reasoning and the :type-prescription rule LONG.

Subgoal *1/1
(IMPLIES (ATOM L1)
         (EQUAL (LONG (APP L1 L2))
                (+ (LONG L1) (LONG L2)))).

By the simple :definition ATOM we reduce the conjecture to

Subgoal *1/1'
(IMPLIES (NOT (CONSP L1))
         (EQUAL (LONG (APP L1 L2))
                (+ (LONG L1) (LONG L2)))).

But simplification reduces this to T, using the :definitions APP, FIX
and LONG, primitive type reasoning, the :rewrite rule UNICITY-OF-0
and the :type-prescription rule LONG.

That completes the proof of *1.

Q.E.D.

Summary
Form:  ( DEFTHM LONG-APP ...)
Rules: ((:DEFINITION APP)
        (:DEFINITION ATOM)
        (:DEFINITION FIX)
        (:DEFINITION LONG)
        (:DEFINITION NOT)
        (:FAKE-RUNE-FOR-LINEAR NIL)
        (:FAKE-RUNE-FOR-TYPE-SET NIL)
        (:REWRITE CDR-CONS)
        (:REWRITE COMMUTATIVITY-OF-+)
        (:REWRITE UNICITY-OF-0)
        (:TYPE-PRESCRIPTION LONG))
Warnings:  None
Time:  0.05 seconds (prove: 0.01, print: 0.02, other: 0.02)
 LONG-APP
```

Como se observa, la descripción de la demostración obtenida se realiza en lenguaje natural (en inglés) y no se detalla a nivel de pasos de inferencia elementales. En concreto, esta demostración se ha obtenido mediante una aplicación del principio de inducción,

distinción de casos y simplificación mediante reemplazamiento de iguales por iguales (por ejemplo, al aplicar el axioma correspondiente a la definición de `app`).

El mundo lógico

A diferencia de lo que ocurre con el ejemplo anterior, lo más usual, siempre que el demostrador intenta una prueba por primera vez, es que no se complete con éxito, lo que ocurrirá con toda seguridad si el teorema que se pretende demostrar no es un resultado trivial. Esto puede significar que la conjetura no sea un teorema. Pero en la mayoría de los casos, la causa del fallo es que ha de proporcionarse un mundo lógico con la suficiente información para que la demostración automática se complete con éxito. Como ya hemos comentado, el papel del usuario es construir un mundo lógico que proporcione un entorno adecuado para la demostración automática de teoremas de una teoría determinada.

Este mundo lógico se construye esencialmente mediante la introducción de nuevas definiciones y la demostración de teoremas que el demostrador utiliza en forma de reglas. Por ejemplo el teorema `long-app` anterior, una vez demostrado, entra a formar parte del mundo lógico, almacenado como regla de reescritura. Esta regla puede ser interpretada como una instrucción al demostrador para que durante los intentos sucesivos de prueba, cualquier ocurrencia de una instancia del término `(long (app l1 l2))` se reescriba a la correspondiente instancia del término `(+ (long l1) (long l2))`.

La figura 2.5, tomada de [37], muestra la relación existente entre el demostrador automático, el usuario y el mundo lógico. El usuario introduce en el demostrador definiciones, conjeturas y posiblemente alguna sugerencia para los intentos de demostración. El demostrador devuelve al usuario el intento de prueba generado, que puede tener éxito o no. Por su parte, el demostrador proporciona al mundo lógico los teoremas que consigue demostrar, almacenados en forma de reglas (usualmente de reescritura). Estas reglas influyen a su vez en los intentos de prueba que el demostrador realiza.

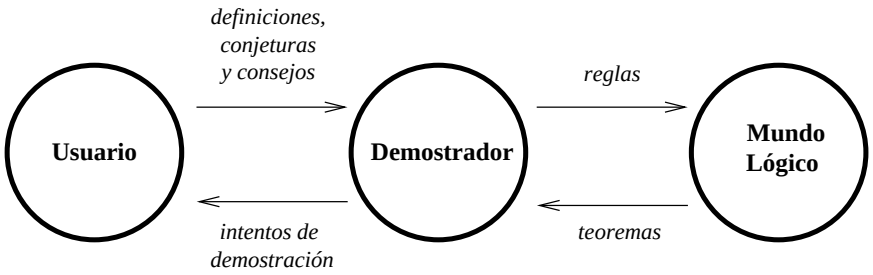

Figura 2.5: Flujo de datos en ACL2

La sintaxis general del comando `defthm` es la siguiente:

`(defthm` *nombre formula*
`:rule-classes` *< lista de tipos de reglas >*
`:hints` *consejos*`)`

De esta manera, se genera un intento de prueba de *formula* y en caso de ser completado con éxito, se almacena en el mundo lógico según lo especificado por la *lista de tipos de reglas*, con nombre *nombre*. Si se especifican *consejos*, el demostrador atiende la

sugerencia del usuario en el intento de prueba, modificando su comportamiento por defecto (comentaremos esto más adelante). Tanto `:rule-classes` como `:hints` son opcionales. Véase `defthm` para más detalles.

El tipo de regla más común es `:rewrite`, indicando que el teorema ha de almacenarse como regla de reescritura. De hecho, si no se especifica `:rule-classes`, el teorema será almacenado como regla de reescritura. Las reglas de reescritura pueden verse como una indicación al demostrador de que se reemplacen determinados términos por otros equivalentes. Las definiciones realizadas con `defun`, si son admisibles, se almacenan como reglas `:definition`, similares a las reglas de reescritura, pero su uso es algo distinto.

Existen hasta 17 tipos de reglas (véase `rule-classes`). Haremos referencia a algunas de ellas cuando describamos los procesos que aplica el demostrador para intentar demostrar una fórmula.

Toda regla del mundo lógico puede estar habilitada o deshabilitada (lo que llamamos su *status*). El demostrador sólo usa las reglas habilitadas en cada momento. De esta manera, el usuario también puede influir en el comportamiento del demostrador, cambiando el *status* de algunas reglas bien globalmente o, en un intento de prueba específico mediante el consejo adecuado. Consúltese `in-theory`.

Cada una de las acciones que secuencialmente modifican el mundo lógico las denominamos *eventos*. Nótese que esta terminología es consistente con el hecho de haber definido anteriormente el concepto de evento como la aplicación de un principio de extensión de la lógica. Cualquier aplicación de un principio de extensión en la lógica tiene su correspondiente evento en el demostrador, que lo implementa. Por ejemplo, el evento correspondiente a `defun` implementa el principio de definición. Además de estos principios de extensión, consideramos como evento, por ejemplo, la demostración de un teorema con `defthm` o el cambio de *status* de una regla.

Organización del demostrador

Describimos a continuación, en líneas generales, los procesos que aplica el demostrador durante un intento de prueba generado mediante una llamada a `defthm`[10]. Recuérdese que el efecto de cada uno de estos procesos está determinado en gran medida por el mundo lógico activo en cada momento.

En la figura 2.6 se muestra gráficamente la organización del demostrador. El círculo central puede interpretarse como una *cesta* de fórmulas, conteniendo en cada momento aquellas fórmulas pendientes de demostración. Inicialmente, la cesta contiene la conjetura que introduce el usuario mediante `defthm`. Alrededor de la cesta, en la figura se representan cada uno de los seis procesos que se pueden aplicar a una conjetura para conseguir una demostración de la misma. Estos procesos han de verse como transformaciones que, a partir de una fórmula dada, obtienen n fórmulas tales que su demostración bastaría para poder concluir la fórmula de entrada. Como caso particular, si $n = 0$ la fórmula está probada. Una vez escogida una fórmula de la cesta, ésta pasa secuencialmente por cada uno de los procesos (en la figura, en el sentido de las agujas del reloj). Si uno de estos procesos no fuera aplicable, la fórmula pasa a ser tratada por el siguiente proceso. Si llega a un proceso aplicable, entonces las n fórmulas en las que se transforma la conjetura se incluyen en la cesta y el proceso comienza de nuevo. Si una fórmula pasa por todos los

[10] Este comportamiento puede cambiar si el usuario incluye sugerencias mediante `:hints`.

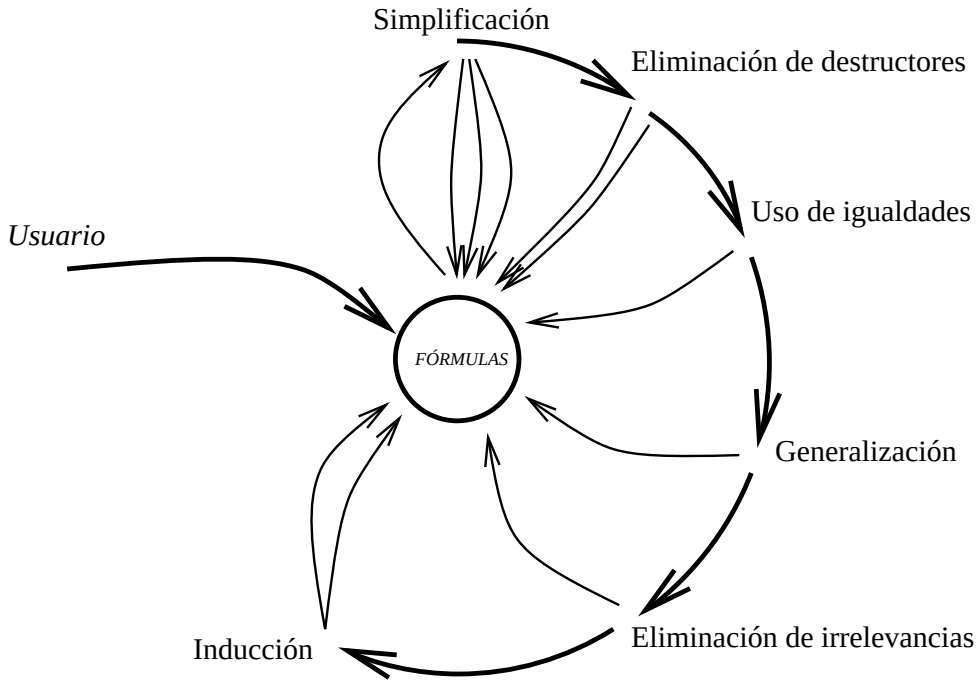

Figura 2.6: Organización del demostrador

procesos y ninguno es aplicable, el demostrador devuelve fallo. Si la cesta se vacía, la demostración se ha completado con éxito.

Las seis técnicas de prueba son las siguientes:

1. **Simplificación**: es el principal proceso de los que aplica el demostrador y su comportamiento viene determinado por las reglas de reescritura presentes, que sirven para simplificar las fórmulas a partir de lemas previamente demostrados por el usuario. Es el único de los procesos que obtiene un conjunto de fórmulas equivalentes a la conjetura que recibe como entrada.

2. **Eliminación de destructores**: este proceso permite expresar ciertos términos de la conjetura haciendo mención expresa a la forma en que han sido construidos. Por ejemplo, si `l` es un par punteado y en una conjetura aparece `(car l)` y `(cdr l)`, este proceso podría escribir `l` como `(cons l1 l2)` y entonces hablar de `l1` y `l2` en lugar de `(car l)` y `(cdr l)`, respectivamente. Usualmente, esta técnica heurística hace más fácil la demostración de una conjetura. El comportamiento de este proceso viene determinado por las reglas de eliminación (tipo `:elim`) que estén presentes en el mundo lógico.

3. **Uso de equivalencias**: si la conjetura a demostrar es una implicación con una igualdad `(equal` e_1 e_2`)` entre sus hipótesis, este proceso hace uso de esta igualdad para sustituir, en el resto de la conjetura, e_1 por e_2. Una vez usada, esta igualdad se hace desaparecer de la conjetura. Este proceso también se denomina *fertilización cruzada*.

4. **Generalización**: este proceso heurístico intenta encontrar un término que ocurre

tanto en las hipótesis como en la conclusión de la conjetura y los sustituye por una variable, obteniendo así una conjetura más general que la original. El comportamiento de este proceso viene determinado por las reglas de generalización (tipo `:generalize`) que estén presentes en el mundo lógico.

5. **Eliminación de irrelevancias**: este proceso intenta detectar hipótesis irrelevantes en la conjetura y eliminarlas.

6. **Inducción**: Este es el único de los procesos que se corresponde directamente con una de las reglas de inferencia primitivas: el principio de inducción. Se trata de elegir, mediante una heurística, el esquema de inducción adecuado, buscando la medida que justifica tal inducción, la división en casos base e inductivos y las hipótesis de inducción correspondiente a cada caso inductivo. El comportamiento del demostrador en este proceso está influido por las definiciones de las funciones recursivas que aparecen en la conjetura.

En el ejemplo de sesión presentado, se puede observar cómo esta metodología para intentar demostrar una conjetura queda reflejada en la salida que devuelve el demostrador. La conjetura inicial se nombra como `*1` y se coloca en la cesta de fórmulas. Cada vez que el demostrador coloca una fórmula en la cesta, ésta se numera. Al considerar la fórmula, cada uno de los seis procesos se intenta aplicar. En este caso, ninguno de los cinco primeros es aplicable. Al aplicar inducción sobre la conjetura `*1`, aparecen dos conjeturas, nombradas por el sistema como subobjetivos `*1/2` y `*1/1` y colocadas en la cesta. El paso siguiente es intentar aplicar la rueda de procesos a `*1/2`. En este caso, se puede aplicar simplificación y obtener la fórmula `*1/2'`. Ésta se puede simplificar a la fórmula nombrada `*1/2''` que nuevamente se simplifica a `t`. Queda pendiente únicamente el subobjetivo `*1/1`. Al aplicar simplificación se obtiene la fórmula que el sistema nombra como `*1/1'`, que pasa a ser ahora la única fórmula de la cesta. Esta fórmula se puede simplificar, ahora a `t`, con lo que la cesta se vacía y la conjetura inicial queda probada.

Los procesos que se aplican con más frecuencia durante un intento de prueba, son los de simplificación y los de inducción, que describiremos a continuación con más detalle. No comentaremos más acerca del resto de procesos. El lector interesado puede consultar [6, 37], donde encontrará una descripción detallada de cada uno de ellos. Hemos de decir, sin embargo, que es posible usar el demostrador sin conocer al detalle cada una de estas técnicas.

Inducción y recursión

Obsérvese que cualquier intento por parte del sistema de aplicar inducción para probar una conjetura, necesita especificar un esquema de inducción adecuado. Si recordamos el principio de inducción descrito en la sección anterior, el sistema debe encontrar:

- una función de medida ordinal m,
- los casos de inducción en los que se divide la conjetura, dados por unas condiciones $q_1, \ldots, q_k$ (el caso base viene determinado por la negación de éstos) y
- las hipótesis de inducción para cada caso q_i, dadas por sustituciones $\sigma_{i,j}$, $1 \leq j \leq h_i$.

Además, debe ser teorema que m toma valores ordinales y que cada una de las hipótesis de inducción se asumen para elementos cuya medida es menor, respecto de `e0-ord-<`.

Un esquema de inducción, aunque correcto, puede ser inadecuado para probar una conjetura, ya que alguno de sus casos base o inductivos puede no ser un teorema. Es crucial para el éxito de una prueba por inducción que el esquema elegido sea el adecuado. El proceso de inducción en ACL2 consiste en aplicar técnicas heurísticas para encontrar un esquema de inducción adecuado. Como se observa en el gráfico de la figura 2.6, el proceso correspondiente a la inducción sólo se aplica si la conjetura ha permanecido estable respecto a los cinco procesos anteriores.

Para encontrar un esquema de inducción adecuado para probar una conjetura, la clave está en analizar detalladamente las funciones recursivas que aparecen en la misma. Es de esperar que al tomar como hipótesis de inducción aquellas instancias que se corresponden con las llamadas recursivas de tales funciones, estas hipótesis sean justamente las que se necesitan para probar cada uno de los pasos de inducción.

Más concretamente, obsérvese la dualidad existente entre los principios de definición y de inducción:

- La medida ordinal que justifica la terminación de una definición puede ser usada como medida que justifica una aplicación del principio de inducción.
- Cada contexto que influye en una llamada recursiva puede ser tomado como un caso inductivo.
- Por cada una de las llamadas recursivas que se producen en un contexto dado, se tiene una hipótesis de inducción distinta.

Así pues, toda función recursiva que ha sido admitida usando el principio de definición *sugiere* un esquema de inducción, que podría usarse en la demostración de conjeturas en las que aparezca tal función. Esta idea es aprovechada por el sistema para encontrar un esquema de inducción que en muchos casos será el adecuado para probar una conjetura. Como ilustración, expliquemos la prueba por inducción que el sistema realiza del teorema `long-app` del ejemplo anterior. Recuérdese que el teorema se expresa como sigue:

```
(defthm long-app
   (equal (long (app l1 l2))
          (+ (long l1) (long l2))))
```

En este caso, el esquema de inducción que lleva a una demostración es el que sugiere la función `app`. Previamente a la demostración del teorema anterior, la definición de `app` se introdujo en el mundo lógico. Esta es la salida del demostrador en dicho evento:

```
ACL2 !>(defun app (l1 l2)
         (if (atom l1)
             l2
           (cons (car l1) (app (cdr l1) l2))))

The admission of APP is trivial, using the relation E0-ORD-< (which
is known to be well-founded on the domain recognized by E0-ORDINALP)
and the measure (ACL2-COUNT L1).  We observe that the type of APP is
```

```
described by the theorem (OR (CONSP (APP L1 L2)) (EQUAL (APP L1 L2) L2)).
We used primitive type reasoning.

Summary
Form:  ( DEFUN APP ...)
Rules: ((:FAKE-RUNE-FOR-TYPE-SET NIL))
Warnings:  None
Time:  0.02 seconds (prove: 0.00, print: 0.00, other: 0.02)
 APP
```

Obsérvese que el sistema escoge automáticamente una medida ordinal (en este caso, `(acl2-count l1)`) e invoca al demostrador para probar las correspondientes conjeturas de terminación. A diferencia de lo que ocurre con el comando `defthm`, el sistema no muestra la salida correspondiente a la demostración de una conjetura de terminación.

La función `app` sugiere un esquema de inducción que, según la correspondencia entre inducción y recursión descrita anteriormente, es:

- Como medida ordinal, `(acl2-count l1)`.
- Como único caso inductivo, `(not (atom l1))`. En consecuencia, el caso base es `(not (not (atom l1)))`
- Como única hipótesis de inducción, aquella que se obtiene sustituyendo `l1` por `(cdr l1)`.

Este esquema de inducción es precisamente el escogido por el demostrador al probar el teorema `long-app`. Recuérdese el mensaje que aparece en la salida correspondiente a la prueba de `long-app`:

```
...
Perhaps we can prove *1 by induction.  Three induction schemes are
suggested by this conjecture.  Subsumption reduces that number to two.
However, one of these is flawed and so we are left with one viable
candidate.

We will induct according to a scheme suggested by (APP L1 L2).  If
we let  (:P L1 L2) denote *1 above then the induction scheme we'll
use is
(AND (IMPLIES (AND (NOT (ATOM L1)) (:P (CDR L1) L2))
              (:P L1 L2))
     (IMPLIES (ATOM L1) (:P L1 L2))).
This induction is justified by the same argument used to admit APP,
namely, the measure (ACL2-COUNT L1) is decreasing according to the
relation E0-ORD-< (which is known to be well-founded on the domain
recognized by E0-ORDINALP).  When applied to the goal at hand the above
induction scheme produces the following two nontautological subgoals.
...
```

Cuando una fórmula se procesa por el método de inducción, ACL2 calcula todos los esquemas de inducción sugeridos por las funciones recursivas que aparecen en la misma y mediante una heurística escoge uno de ellos. El esquema se presenta en pantalla. En este caso, `(:P L1 L2)` representa a la conjetura. Nótese que *no es necesario* demostrar que el esquema de inducción es correcto, ya que eso es una consecuencia inmediata de que la función que sugiere el esquema (`app` en este caso) ha sido previamente aceptada por el principio de definición.

El esquema de inducción anterior produce los dos siguientes subobjetivos, que serán incluidos en la cesta de fórmulas pendientes de demostrar. El primero de ellos es el caso inductivo y el segundo de ellos el caso base.

```
Subgoal *1/2
(IMPLIES (AND (NOT (ATOM L1))
              (EQUAL (LONG (APP (CDR L1) L2))
                     (+ (LONG (CDR L1)) (LONG L2))))
         (EQUAL (LONG (APP L1 L2))
                (+ (LONG L1) (LONG L2)))).

Subgoal *1/1
(IMPLIES (ATOM L1)
         (EQUAL (LONG (APP L1 L2))
                (+ (LONG L1) (LONG L2)))).
```

Simplificación y reescritura

El proceso de simplificación es el más importante de todos los que aplica el demostrador. Esta técnica de prueba es una combinación de procesos que transforman la fórmula en un conjunto de fórmulas equivalente. Entre estas técnicas se encuentran:

- Aplicación de procedimientos de decisión relativos al cálculo proposicional, la igualdad y la aritmética lineal.
- Codificación, mediante información de tipos, de las condiciones que se pueden asumir como ciertas en cada ocurrencia de un término en una conjetura.
- Reescritura de los subtérminos que aparecen en la conjetura, usando reglas de reescritura y definiciones.
- Normalización proposicional y eliminación de fórmulas subsumidas.

El lector puede consultar [6, 37] si quiere una descripción detallada de cada uno de estos procesos. Por su importancia, describiremos brevemente aquí la reescritura de términos.

En esencia, el proceso de reescritura consiste en reemplazar unos términos por otros que previamente se han probado equivalentes. Estas equivalencias se especifican en el sistema en forma de reglas de reescritura, que a su vez se generan a partir de los teoremas demostrados previamente.

Veamos algunos ejemplos de teoremas (tomados de la teoría presentada en esta memoria) y comentemos las reglas de reescritura que generan:

```
(defthm equal-len-replace-list
  (equal (len (replace-list l i x))
         (len l)))

(defthm nth-replace-list-same-position
  (implies (and (integerp i)
                (<= 0 i)
                (< i (len l)))
           (equal (nth i (replace-list l i x)) x)))

(defthm member-append
  (iff (member x (append a b))
       (or (member x a) (member x b))))

(defthm eliminate-preserves-setp
  (implies (setp l)
           (setp (eliminate x l))))
```

En lo que sigue, usamos las letras α, β y γ para representar a términos cualesquiera. El primer teorema se corresponde con una regla de reescritura que reemplaza términos de la forma `(len (replace-list` α β γ`))` por `(len` α`)`.

El segundo teorema genera una regla de reescritura que indica al sistema que reemplace términos de la forma `(nth i (replace-list` α β γ`))` por γ, siempre que se pueda establecer `(integerp` β`)`, `(<= 0` β`)` y `(<` β `(len` α`))`. Esta es una *regla de reescritura condicional*, puesto que indica un reemplazamiento sujeto a que se verifiquen ciertas premisas.

La regla de reescritura del tercer teorema permite reemplazar `(member` α `(append` β γ`))` por `(or (member` α β`) (member` α γ`))`. El usar `iff` en lugar de `equal` restringe este reemplazamiento a situaciones en las que sólo el valor de verdad es relevante (por ejemplo, en hipótesis de fórmulas). Es posible reescribir respecto de relaciones de equivalencia distintas de `equal`, siempre que se restrinjan los lugares en los que el reemplazamiento es posible. En el siguiente apartado comentaremos esta cuestión.

El último de los teoremas debe ser interpretado como si su conclusión fuera `(iff (setp (eliminate x l)) t)`. En general, si un teorema tiene como conclusión un término de la forma `(`p $\ldots$`)` y p no es una relación de equivalencia, ésta debe ser interpretada como si fuera `(iff (`p $\ldots$`) t)`. Si por el contrario, la conclusión es de la forma `(not (`p $\ldots$`))`, se interpreta como si fuera `(iff (`p $\ldots$`) nil)`.

Las reglas de reescritura del mundo lógico (las que estén habilitadas) constituyen un sistema de reescritura, que es empleado por el sistema durante el proceso de simplificación para obtener versiones normalizadas de los términos que aparecen en las conjeturas. Obsérvese, por ejemplo, cómo el proceso de simplificación sirve para demostrar el subobjetivo `*1/1'`:

```
Subgoal *1/1'
(IMPLIES (NOT (CONSP L1))
         (EQUAL (LONG (APP L1 L2))
                (+ (LONG L1) (LONG L2)))).
```

```
But simplification reduces this to T, using the :definitions APP, FIX
and LONG, primitive type reasoning, the :rewrite rule UNICITY-OF-0
and ...
```

En este caso se ha usado una regla de reescritura, denominada `unicity-of-0`, regla obtenida a partir de uno de los axiomas primitivos de ACL2. Obsérvese que además de esta regla de reescritura, se han usado las definiciones de las funciones `app`, `fix` y `long`. Los axiomas que se obtienen mediante el principio de definición dan lugar a reglas de reescritura, pero su aplicación en el caso de definiciones recursivas está restringida, mediante ciertas comprobaciones heurísticas que evitan la no terminación. Estas reglas generadas a partir de definiciones tienen tipo `:definition`. Cuando una regla de definición se aplica, decimos que la definición se *expande*.

La reescritura de un término se produce en un determinado contexto que codifica las condiciones que podemos asumir como ciertas cuando se reescribe un término. Por ejemplo, si la conjetura es de la forma `(implies (and` p_1 ... p_n`)` q`)`, al reescribir un subtérmino de q podemos asumir que $p_1, \ldots, p_n$ son ciertos. Estas suposiciones se codifican en forma de información acerca del tipo de dato de determinados términos. Para establecer las premisas de una regla de reescritura condicional, el demostrador intenta reducir cada una de ellas a `t`, *usando sólamente reescritura e información sobre tipos*. Así pues, el uso de reglas de reescritura condicionales hace que se genere un proceso de encadenamiento hacia atrás que se aplica recursivamente. No es el objetivo de esta introducción dar una descripción detallada del proceso de reescritura que implementa ACL2. Téngase en cuenta además que la reescritura es sólo uno de los subprocesos que forman parte de la simplificación. Volvemos a remitir al lector a las referencias [6, 37] para una descripción detallada de esta técnica de prueba.

El éxito o fracaso en el desarrollo de una teoría formal usando el demostrador automático de ACL2, depende en gran medida de cómo se generen reglas de reescritura a partir de un teorema. Nótese que un mismo teorema puede ser expresado de diferentes maneras, cada una de ellas dando lugar a una regla de reescritura distinta. En general, es más recomendable especificar las reglas de manera que reemplacen términos por otros más simples. Esta noción de simplicidad debe ser determinada por el usuario ACL2; en cualquier caso, todas las reglas de reescritura que se generen deben de ser coherentes con la noción de simplicidad adoptada.

Reescritura respecto de congruencias

Como hemos comentado, las reglas de reescritura pueden especificar un reemplazamiento de términos por otros equivalentes, aunque tal relación de equivalencia no sea `equal`. En ese caso, el reemplazamiento ha de restringirse a determinados contextos. Por ejemplo, en el caso de `iff`, los reemplazamientos quedan restringidos a lugares en los que sólo el valor proposicional del término es importante.

El usuario puede declarar determinadas funciones binarias como relaciones de equivalencia y especificar en qué lugares está permitido reemplazar términos por otros equivalentes respecto de dicha relación. Veámoslo con un ejemplo.

La función `equal-set` es una de las funciones de la teoría presentada en esta memoria

e implementa la igualdad conjuntista entre listas[11]:

```
(defun equal-set (x y)
  (and (subsetp x y) (subsetp y x)))
```

Podemos declarar esta función binaria como relación de equivalencia, mediante la siguiente llamada al comando `defequiv`:

```
(defequiv equal-set)
```

Esta llamada genera el intento de demostrar que `equal-set` verifica las propiedades reflexiva, simétrica y transitiva. Si se completa con éxito (como en este caso), el sistema marca la función como relación de equivalencia.

Podemos también indicar al demostrador que el segundo argumento de la función `member` puede ser reemplazado por cualquier elemento equivalente respecto de la igualdad conjuntista. El comando `defcong` nos permite este tipo de declaraciones. En este caso concreto:

```
(defcong equal-set iff (member x y) 2)
```

De esta manera, se le indica al demostrador que la reescritura, usando términos equivalentes respecto de `equal-set`, está permitida cuando el subtérmino reescrito aparece como segundo argumento de `member`, siempre que sea en contextos donde el valor proposicional sea lo relevante. Esta llamada a `defcong` genera el intento de prueba de las correspondientes conjeturas que nos permiten afirmar esto.

Una vez definida una equivalencia y los lugares donde se permite reemplazamiento respecto de la misma, las reglas de reescritura que se definan usando dicha relación de equivalencia instruirán al demostrador sobre los reemplazamientos que puede realizar. Por ejemplo, la siguiente regla de reescritura permite reemplazar subtérminos de la forma `(make-set` α`)` por α, siempre que éstos figuren como argumentos de funciones que se han demostrado congruentes respecto de `equal-set` (por ejemplo, el segundo argumento de `member`). Aquí `make-set` es una función que elimina repeticiones de una lista:

```
(defthm equal-set-make-set
  (equal-set (make-set l) l))
```

En general, si *equiv* es una relación que el sistema reconoce como de equivalencia, cada vez que se prueba un teorema de la forma:

```
(implies (and hip1 hip2 ... hipn)
         (equiv izq der))
```

y se declara como regla de reescritura (lo cual ocurre por defecto), se genera una regla que instruye al demostrador para que durante los intentos de prueba se reemplacen subtérminos que sean instancias de *izq* por la correspondiente instancia de *der*, siempre y cuando se establezcan las correspondientes instancias de $hip_1, \ldots, hip_n$. Estos reemplazamientos

[11] El predicado `subsetp` está predefinido en Common Lisp y devuelve `t` si y sólo si todo miembro del primer argumento lo es del segundo.

sólo podrán ser realizados en posiciones previamente declaradas por `defcong`. La relación `equal` es un caso particular de relación de equivalencia, que permite reemplazamiento en cualquier posición.

En la secciones 5.2.6 y 4.1.4 se describen ejemplos de relaciones de equivalencia y congruencias usadas para desarrollar la teoría presentada en esta memoria.

Indicaciones al demostrador

El comportamiento general del demostrador a la hora de abordar un intento de prueba es el que se ha descrito previamente. Sin embargo, es posible por parte del usuario proporcionar determinadas sugerencias (o consejos) que permitan al demostrador "tomar un atajo". Estas sugerencias se incluyen como parámetro opcional al comando `defthm`, usando la palabra clave `:hints`. En una sugerencia se especifica además el subobjetivo del intento de prueba al cual va dirigida. Véase `hints` para más información.

La mayoría de los lemas y teoremas que se incluyen en la teoría presentada en esta memoria se demuestran sin necesidad de sugerencias. En aquellos teoremas que sí las usan, éstas son principalmente de cuatro tipos.

- *Usar una instancia de un teorema previamente demostrado.* En ciertas ocasiones, el mecanismo de reescritura no es adecuado para poder usar un teorema demostrado previamente. Por ejemplo, en el caso en el que las hipótesis en una regla de reescritura condicional tengan variables que no aparezcan en la conclusión (variables *libres*). En tal caso, el establecimiento de las hipótesis de la regla exige al demostrador "adivinar" el valor adecuado de dicha variable. Con frecuencia, esto hace que una regla de reescritura con variables libres no se pueda aplicar.

  En situaciones como ésta, el usuario puede proporcionar explícitamente al demostrador una instancia del teorema que pretende usar. Esto se consigue mediante una sugerencia con la palabra clave `:use`:

  `:use (:instance` *lema* `(`v_1 e_1`) ... (`v_n e_n`))`

  Esta sugerencia indica al demostrador que en el intento de prueba de la conjetura incluya como hipótesis adicional el teorema *lema* (previamente probado), instanciado mediante la sustitución $\{v_1 \mapsto e_1, \ldots, v_n \mapsto e_n\}$, donde cada v_i es una variable y cada e_i un término.

  Este tipo de consejos son los más usados en el desarrollo de la teoría descrita en esta memoria.

- *Usar una instancia funcional de un teorema previamente demostrado.* Mediante este consejo, el usuario puede aplicar la regla de inferencia derivada de instanciación funcional, descrita en la sección anterior. Este consejo se expresa de la siguiente manera:

  `:use (:functional-instance` *lema* `(`f_1 g_1`) ... (`f_n g_n`))`

  donde *lema* es el nombre de un teorema[12] previamente demostrado, cada f_i es un

[12] O, recursivamente, una instancia de un lema, obtenida bien usando `:instance` o bien `:functional-instance`.

símbolo de función y cada g_i es un símbolo de función o una lambda expresión[13] de la misma aridad que f_i.

Para que esta sugerencia se pueda aplicar, previamente se intenta demostrar que cada g_i verifica las correspondientes instancias funcionales de los axiomas que asertan propiedades acerca del correspondiente f_i. Si este intento de demostración termina con éxito, entonces la correspondiente instancia funcional de *lema* es un teorema, justificado por la regla de inferencia (derivada) de instanciación funcional.

Una vez verificadas las restricciones sobre cada g_i, el demostrador incluye en el intento de prueba de la conjetura una hipótesis adicional obtenida a partir del teorema *lema* instanciado funcionalmente mediante el reemplazamiento de cada f_i por g_i.

Hay varios ejemplos de uso de instanciación funcional en el desarrollo de la teoría que se presenta en esta memoria (véanse, por ejemplo, las páginas 347 y 366).

- *Habilitación y deshabilitación.* Ya hemos comentado que es posible habilitar o deshabilitar reglas del mundo lógico. Este cambio en el *status* de una regla puede hacerse bien globalmente, o bien localmente, afectando sólo a un intento de prueba determinado, en cuyo caso se incluye como sugerencia.

  Por ejemplo, a veces es necesario deshabilitar determinadas reglas de reescritura para que no reescriban determinados términos o no entren en conflicto con la estrategia de reescritura de otras reglas. De la misma manera se pueden deshabilitar la definición de algunas funciones, evitando así su expansión durante un intento de prueba.

  Otras veces, la presencia de determinadas reglas de reescritura sobrecarga el proceso que el demostrador dedica a la reescritura, por lo que es conveniente mantenerlas globalmente deshabilitadas y habilitarlas localmente en determinados intentos de prueba, mediante el correspondiente consejo.

  La palabra clave `:in-theory` hace posible la habilitación y deshabilitación de reglas, de la siguiente manera:

  `:in-theory (enable` r_1 ... r_n`)`

  `:in-theory (disable` r_1 ... r_n`)`

  De esta manera, con `enable` habilitamos las reglas $r_1, \ldots, r_n$ y con `disable` las deshabilitamos. Si se desea una descripción más detallada, consúltese `theories`.

- *Indicación de un esquema de inducción.* En algunos casos (los menos), la heurística que utiliza el sistema para generar un esquema de inducción no encuentra el esquema adecuado para abordar una prueba de una conjetura. El usuario puede indicar al sistema el esquema de inducción que considere conveniente, de la siguiente manera:

  `:induct (`f x_1 ... x_n`)`

  El único medio que tiene el usuario de indicar al sistema un esquema de inducción es mediante una función recursiva. En concreto, este consejo de inducción hace

[13] El cuerpo de una lambda expresión usada para obtener una instancia funcional puede tener, bajo ciertas restricciones, variables libres.

que el sistema aborde la demostración de una conjetura, aplicando en primer lugar el proceso de inducción, usando como esquema el sugerido por la función f. En muchos casos, la función f se define exprésamente para tal cometido. El lector puede encontrar un ejemplo del uso de consejo de inducción en la página 187.

Relaciones bien fundamentadas en ACL2

Podemos definir en ACL2 una relación bien fundamentada *rel* definida sobre un conjunto de objetos que satisfacen una propiedad *mp*, mediante la demostración de un teorema de la siguiente forma:

```
(and (implies (mp x) (e0-ordinalp (fn x)))
     (implies (and (mp x)
                   (mp y)
                   (rel x y))
              (e0-ord-< (fn x) (fn y))))
```

El predicado *mp* es un predicado que sirve para definir el conjunto de objetos que están ordenados de una manera bien fundamentada mediante *rel*. La función *fn* (llamada función de *inmersión*) es una función monótona que hace corresponder un ordinal a cada objeto de medida. Es evidente que este teorema implica que *rel* es una relación bien fundamentada sobre *mp* (si asumimos que `e0-ord-<` lo es sobre `e0-ordinalp`). Como caso particular, cuando *mp* es la función con valor constante igual a `t`, es posible omitir las referencias a *mp* en el enunciado del teorema anterior.

La única relación bien fundamentada primitiva en la lógica de ACL2 es `e0-ord-<`, definida en el conjunto de objetos que satisfacen `e0-ordinalp`. El usuario añade relaciones bien fundamentadas probando teoremas como éste, que se guardan como reglas de tipo `:well-founded-relation`.

Estas relaciones bien fundamentadas se pueden usar para probar la terminación de funciones recursivas. En los casos en los que el demostrador no sea capaz de demostrar automáticamente la terminación de una función, el usuario puede indicar, al definirla con `defun` y mediante la palabra clave `:measure`, una función de medida, para que el sistema intente demostrar que dicha medida decrece en cada llamada recursiva. Esta medida ha de tomar valores en el dominio de definición de una relación bien fundamentada, que también se puede indicar mediante la palabra clave `:well-founded-relation`. Es decir, la siguiente definición:

```
(defun f (args)
   (declare (xargs :measure m
                   :well-founded-relation rel))
   cuerpo)
```

provoca el intento de demostración de la terminación de la función f probando que la medida m decrece respecto de la relación bien fundamentada *rel*, en cada llamada recursiva de f. Es posible demostrar, como meta–teorema, que esta prueba de terminación de f se puede reducir (usando el teorema de buena fundamentación previamente probado para *rel*) a una prueba de terminación usando `e0-ord-<`. Si no se especifica una relación bien fundamentada, se usa `e0-ord-<`.

Mecanismos de estructuración de teorías

El desarrollo de una teoría en ACL2 no es lineal. Como veremos en el siguiente apartado, en la demostración de un teorema, usualmente se necesitan lemas que, actuando como reglas, instruyen al demostrador sobre la demostración del teorema requerido. Estos lemas, a su vez, pueden requerir de otros lemas y así sucesivamente, hasta llegar a lemas cuya prueba es lo suficientemente simple como para que pueda ser encontrada por el demostrador.

Esto hace necesario unos mecanismos de estructuración de teorías, ya que determinados lemas sólo son necesarios en determinados contextos y no deberían estar presentes en el mundo lógico cuando se abordan otros intentos de prueba. Esencialmente, existen dos mecanismos de estructuración de teorías en ACL2.

El primero de ellos es el que proporcionan los *libros*. Un libro en ACL2 es un fichero conteniendo una secuencia de eventos que desarrollan una teoría, principalmente definiciones y teoremas. Los eventos que no interesan fuera del entorno en el que se demuestran los teoremas de un libro, se declaran como locales mediante el comando `local`. Si *ev* es un evento que aparece en un libro como `(local` *ev*`)`, se dirá que es local en el libro.

Los libros se pueden certificar, usando el comando `certify-book`, lo que asegura que todos los eventos del mismo son admisibles. En particular, todas sus definiciones se admiten por el principio de definición y todos sus teoremas tienen una demostración. Los libros que se hayan certificado se pueden incluir, como punto de partida para el desarrollo de otros libros, usando `include-book`. El certificar un libro asegura, además, que los eventos no locales son independientes de los eventos locales del mismo. Véase `books` para más información.

La manera habitual de estructurar el desarrollo de una teoría es dividirla en libros, cada uno de ellos dedicado a una parte importante de la formalización. Cada libro tiene una serie de definiciones y teoremas no locales que se "exportarán" cuando éste sea incluido en otro libro y que constituyen el conjunto de eventos importantes del mismo. Los eventos locales de cada libro ayudan a la demostración de los teoremas no locales del mismo, pero no salen hacia afuera.

Dentro de un libro, es posible también tener una cierta estructuración de los eventos. El comando `encapsulate`, además de permitir introducir funciones parcialmente definidas, se puede usar para "esconder" los lemas necesarios en la prueba de un teorema, sin necesidad de crear un fichero. Por ejemplo:

```
(encapsulate
 ()

 (local (defthm lema1 ...))
 (local (defthm lema2 ...))

 (defthm teorema ...))
```

En este caso, los lemas `lema1` y `lema2`, son resultados locales en el encapsulado, que permiten construir un mundo lógico adecuado para demostrar `teorema`. Fuera del encapsulado, sólo `teorema` es "visible". Como se observa, este encapsulado tiene una signatura vacía `()` y por tanto no se está usando para definir parcialmente ninguna función. Su uso aquí tiene como objetivo limitar el alcance de `lema1` y `lema2`, estructurando así la prueba de `teorema`.

2.3.2 Cómo usar el demostrador

Una vez hemos descrito algunos aspectos del funcionamiento del demostrador, describimos aquí cómo usarlo para obtener demostraciones automáticas de resultados no triviales.

El Método

Como hemos mencionado anteriormente, el demostrador es automático en el sentido de que una vez comienza un intento de prueba, no es posible interactuar con él (salvo para cancelar el intento). Volvemos a insistir que el demostrador rara vez encontrará una prueba directa de un resultado no trivial. Es labor del usuario el construir un mundo lógico que instruya adecuadamente al demostrador para conseguir una demostración. Es decir, se trata de definir las funciones auxiliares y demostrar los lemas previos necesarios que permitan a ACL2 demostrar el resultado deseado.

Para ello, explicamos aquí lo que los autores del sistema llaman "El Método" [37]: es decir, el procedimiento que en su opinión es el más adecuado para guiar al demostrador hacia la prueba de un resultado. Antes de aplicar "El Metodo", es necesario cumplir dos prerrequisitos importantes:

- *Prerrequisito 1: formalización del resultado en la lógica de ACL2.* Formalizar un problema en ACL2 consiste en expresarlo mediante fórmulas de la lógica, de manera que se convierta en un problema de deducción. Recuérdese que la lógica de ACL2 es una lógica constructiva, de primer orden y sin cuantificadores, por lo que a veces no es trivial enunciar, mediante fórmulas de la misma, el resultado que se pretende demostrar. Sin embargo, existen ejemplos de que problemas complejos y muy abstractos se han formalizado tanto en la lógica de Boyer y Moore como en la lógica de ACL2. Una adecuada formalización es fundamental para poder abordar con éxito el problema en el demostrador automático. En el capítulo 3 de [7] se presentan numerosos ejemplos de formalización usando la lógica de Boyer y Moore. En esta memoria haremos especial énfasis en las cuestiones relativas a la formalización.

- *Prerrequisito 2: tener una prueba (informal) del resultado.* Es decir, al menos esquematicamente, es conveniente conocer previamente una prueba (en la logica de ACL2) del resultado, puesto que se trata de guiar al demostrador hacia esa prueba. Por supuesto, tal prueba no debe ser detallada, ya que es esa precisamente una de las tareas del demostrador. El demostrador *no descubre* demostraciones, sino que completa los "huecos" que existen entre una prueba esquemática e informal y una prueba formal.

La manera más usual de usar ACL2 es en conjunción con un editor de textos. En nuestro caso (y en el de la mayoría de los usuarios), usamos Emacs. De esta manera, ejecutamos ACL2 en un *"buffer"* donde recogemos su salida. Simultáneamente, en un fichero de texto vamos desarrollando estructuradamente una teoría, mediante definiciones, teoremas y demás eventos, con la idea de que finalmente éste fichero constituya un libro ACL2 que se pueda certificar.

Durante su desarrollo, el fichero de eventos se puede ver intuitivamente divido en dos partes, separadas por una *barrera* imaginaria. Por encima de la barrera, se encuentra la lista de los eventos que ya han sido llevados a cabo con éxito y por debajo, los que aún quedan por hacer. Inicialmente, el fichero de eventos contiene el teorema principal que se

desea demostrar, p (escrito con un comando `defthm`), además de las definiciones necesarias para poder enunciarlo. La barrera se encuentra justo encima de p. Es decir, la lista de eventos por hacer contiene sólamente al teorema principal. La demostración se completará cuando p cruce la barrera.

A continuación describimos "El Metodo", estructurado en cuatro pasos. Creemos que la mejor manera de hacerlo es presentarlo tal y como lo hacen los autores del sistema. Por tanto, traducimos directamente de [37]:

- *PASO 1: Pensar acerca de la demostración del primer teorema de la lista de eventos por hacer. Estructurar la demostración como una inducción seguida de una simplificación o bien únicamente como una simplificación. ¿Se han demostrado ya todos los lemas que se necesitan? Es decir, ¿están estos lemas en la lista de eventos ya realizados? Si es así, ir al paso 2. En caso contrario, añadir los lemas necesarios al principio de los eventos por hacer y repetir el paso 1.*

- *PASO 2: Llamar al demostrador sobre el primer teorema en la lista de de teoremas por hacer y dejar que la salida vaya apareciendo en el "buffer" de Emacs donde se ejecuta ACL2. Cancelar la demostración si tarda más de unos segundos.*

- *PASO 3: Si el demostrador terminó la demostración con éxito, mover la barrera un evento hacia adelante y volver al paso 1.*

- *PASO 4: En caso contrario, analizar la salida del intento de prueba, en el "buffer" de emacs, comenzando desde el principio (no por el final). Básicamente, debe buscarse el primer punto en el intento de prueba, que se desvía de la prueba que se tiene en mente. Modificar el "buffer" de eventos, como consecuencia de este análisis. Más adelante comentaremos esto más extensamente. Usualmente, esto significa añadir una serie de lemas a la lista de eventos por hacer, justo delante del teorema cuya demostración se acaba de intentar. O también podría significar que hay que añadir sugerencias al teorema actual. En cualquier caso, despues de las modificaciones realizadas, volver al paso 1.*

Intuitivamente, la prueba de un resultado no trivial en ACL2 se puede ver estructurada en forma de árbol. Cada nodo del árbol es un lema o teorema que tiene como descendientes directos los lemas previos que se necesitan para su demostración. En la raiz se encuentra el resultado principal que se quiere demostrar. Cada nodo del árbol se demuestra, esencialmente, usando simplificación y, posiblemente, inducción. La idea básica en "El Método" es la construcción incremental de este árbol de prueba. Es el usuario el que decide la composición de este árbol de prueba en sus niveles más altos. Los niveles más bajos del árbol se completan mediante análisis de los intentos fallidos de demostración. Básicamente, ésta es la estrategia que hemos seguido para desarrollar la teoría que se presenta en esta memoria. El lector puede consultar [42] donde se explica con todo detalle la demostración de un resultado en Nqthm, siguiendo la estrategia descrita.

Análisis de las pruebas fallidas

Para llevar a cabo el paso 4 de "El Metodo", es necesario deducir, a partir del análisis de una prueba fallida, el lema o lemas necesarios que, añadidos al mundo lógico, permitan reconducir la demostración. Recuérdese que la situación en el paso 4 es la siguiente: se

intenta demostrar algo que se piensa que es un teorema y se cree que el demostrador debería ser capaz de probarlo en el mundo lógico construido hasta el momento. Es destacable que a veces, como consecuencia de este análisis, el usuario se da cuenta de que la conjetura no es teorema y necesita ser modificada. En la mayoría de los casos, lo que ocurre, sin embargo, es que aún es necesario proporcionar más información al mundo lógico para que se complete la prueba. Ilustramos esta cuestión con un ejemplo.

Supongamos definido un predicado `setp` que comprueba si su único argumento es una lista sin repeticiones. Para el desarrollo de la teoría presentada en esta memoria, hemos necesitado como lema que la propiedad `setp` se conserva si eliminamos de la lista un elemento. Es decir, si `eliminate` es la función que implementa la eliminación de elementos de una lista, se trata de probar el siguiente resultado:

```
(defthm eliminate-preserves-setp
 (implies (setp l)
          (setp (eliminate x l))))
```

Este resultado parece lo suficientemente obvio como para que no necesitemos más lemas previos, ya que una inducción en la longitud de `l` (por ejemplo, siguiendo el esquema sugerido por `eliminate`) debería bastar. Intentamos, pues, que ACL2 lo demuestre automáticamente. Sin embargo, el intento de demostración en ACL2 resulta fallido. Este es el principio de la salida de esta prueba fallida:

```
Name the formula above *1.

Perhaps we can prove *1 by induction.  Two induction schemes are suggested
by this conjecture.  These merge into one derived induction scheme.

We will induct according to a scheme suggested by (ELIMINATE X L).
If we let  (:P L X) denote *1 above then the induction scheme we'll
use is
(AND (IMPLIES (AND (NOT (ENDP L))
                   (NOT (EQUAL X (CAR L)))
                   (:P (CDR L) X))
              (:P L X))
     (IMPLIES (AND (NOT (ENDP L))
                   (EQUAL X (CAR L))
                   (:P (CDR L) X))
              (:P L X))
     (IMPLIES (ENDP L) (:P L X))).
This induction is justified by the same argument used to admit ELIMINATE,
namely, the measure (ACL2-COUNT L) is decreasing according to the relation
E0-ORD-< (which is known to be well-founded on the domain recognized
by E0-ORDINALP).  When applied to the goal at hand the above induction
scheme produces the following five nontautological subgoals.

Subgoal *1/5
(IMPLIES (AND (NOT (ENDP L))
              (NOT (EQUAL X (CAR L)))
```

```
              (SETP (ELIMINATE X (CDR L)))
              (SETP L))
         (SETP (ELIMINATE X L))).

By the simple :definition ENDP we reduce the conjecture to

Subgoal *1/5'
(IMPLIES (AND (CONSP L)
              (NOT (EQUAL X (CAR L)))
              (SETP (ELIMINATE X (CDR L)))
              (SETP L))
         (SETP (ELIMINATE X L))).

This simplifies, using the :definitions ELIMINATE and SETP, primitive
type reasoning, the :rewrite rules CAR-CONS and CDR-CONS and the :type-
prescription rules ELIMINATE and SETP, to

Subgoal *1/5''
(IMPLIES (AND (CONSP L)
              (NOT (EQUAL X (CAR L)))
              (SETP (ELIMINATE X (CDR L)))
              (NOT (MEMBER (CAR L) (CDR L)))
              (SETP (CDR L)))
         (NOT (MEMBER (CAR L)
                      (ELIMINATE X (CDR L))))).

The destructor terms (CAR L) and (CDR L) can be eliminated by using
CAR-CDR-ELIM to replace L by (CONS L1 L2), generalizing (CAR L) to
L1 and (CDR L) to L2.  This produces the following goal.

Subgoal *1/5'''
(IMPLIES (AND (CONSP (CONS L1 L2))
              (NOT (EQUAL X L1))
              (SETP (ELIMINATE X L2))
              (NOT (MEMBER L1 L2))
              (SETP L2))
         (NOT (MEMBER L1 (ELIMINATE X L2)))).

This simplifies, using primitive type reasoning, to

Subgoal *1/5'4'
(IMPLIES (AND (NOT (EQUAL X L1))
              (SETP (ELIMINATE X L2))
              (NOT (MEMBER L1 L2))
              (SETP L2))
         (NOT (MEMBER L1 (ELIMINATE X L2)))).
```

```
We generalize this conjecture, replacing (ELIMINATE X L2) by EE and
restricting the type of the new variable EE to be that of the term
it replaces, as established by ELIMINATE.  This produces

Subgoal *1/5'5'
(IMPLIES (AND (TRUE-LISTP EE)
              (NOT (EQUAL X L1))
              (SETP EE)
              (NOT (MEMBER L1 L2))
              (SETP L2))
         (NOT (MEMBER L1 EE))).

Name the formula above *1.1.
```

... sigue ...

El intento de prueba continúa, generando 51 subobjetivos (el último de los cuales aparece numerado como `*1.1.1.1/1'''`) y fallando finalmente. La prueba debe empezar a analizarse por el principio: con frecuencia, un paso equivocado al principio hace que se generen subobjetivos que ni siquiera son teoremas, lo que provoca el fallo posterior. Así que en la práctica, sólo las primeras líneas de una prueba fallida son importantes para este análisis. En este caso, hemos presentado la salida del intento de prueba justo hasta el punto donde se genera una conjetura que no es cierta. Es el subobjetivo `*1/5'5'`, que ha sido obtenido por el proceso de generalización. Hasta ese punto, el intento de prueba seguía el camino esperado: inducción sugerida por `eliminate`, que produce cinco casos. Durante el primero de estos casos (subobjetivo `*1/5`) se aplica simplificación y eliminación de destructores, obteniendo el subobjetivo `*1/5'4'`. Finalmente se aplica generalización y se obtiene el mencionado `*1/5'5'`. Este subobjetivo no se transforma por ninguno de los procesos previos a la inducción, por lo que finalmente se le asigna el número `*1.1` y se guarda en la cesta para un intento posterior de prueba por inducción.

Puesto que `*1.1` no es cierto, debemos centrar nuestra atención en el último subobjetivo que sí pensamos que es cierto:

```
Subgoal *1/5'4'
(IMPLIES (AND (NOT (EQUAL X L1))
              (SETP (ELIMINATE X L2))
              (NOT (MEMBER L1 L2))
              (SETP L2))
         (NOT (MEMBER L1 (ELIMINATE X L2)))).
```

Pensemos qué necesitaría el sistema para poder probar este subobjetivo por simplificación. La idea es que `(eliminate x l2)` es subconjunto de `l2` y por tanto si algo no es miembro de `l2`, tampoco lo será de `(eliminate x l2)`. El resto de hipótesis son irrelevantes. Probemos, pues, el siguiente lema:

```
(defthm member-eliminate
    (implies (not (member y l))
             (not (member y (eliminate x l)))))
```

Este resultado se demuestra automáticamente en ACL2. La razón por la que no se demostró durante el intento de prueba anterior es que la existencia de hipótesis adicionales e irrelevantes hizo que el proceso de generalización transformara la conjetura antes de ser abordada por inducción, apareciendo un subobjetivo que no era cierto.

Con el lema `member-eliminate` presente como regla de reescritura en el mundo lógico, la demostración de `eliminate-preserves-setp` se completa ahora con éxito, siguiendo la prueba por inducción que inicialmente teníamos esquematizada.

Este ejemplo ilustra cómo analizar la salida de una prueba fallida, para obtener los lemas necesarios que permitan modificar el mundo lógico y reconducir la prueba. Como se ha descrito, se trata de inspeccionar el intento de prueba desde el principio, buscando la primera "desviación" de la prueba imaginada previamente. Existen dos tipos de situaciones donde, con más frecuencia, un intento de prueba puede divergir del preconcebido:

- *Procesos "peligrosos"*: el proceso de simplificación es el único que obtiene fórmulas equivalentes a las de entrada. Usualmente, la eliminación de destructores también obtiene fórmulas equivalentes. El resto de procesos los podemos denominar "peligrosos", ya que pueden reducir un subobjetivo que es un teorema a otro que ya no lo es. Es el caso de la generalización del ejemplo presentado. Un subobjetivo que no puede ser simplificado más y que es tomado por otro proceso, es un buen candidato para sugerir un lema. Este lema ha de añadir la correspondiente regla al mundo lógico que permita la simplificación de la conjetura.

- *Esquemas de inducción inadecuados*: otro punto que debe ser examinado es el esquema escogido para demostrar una fórmula por inducción y compararlo con la prueba por inducción que previamente se ha esquematizado. En este caso puede que un consejo `:induct` reconduzca la situación.

Aunque una prueba se complete con éxito, es recomendable inspeccionarla. Puede ocurrir que el sistema encuentre una prueba completamente diferente de la que uno espera. Esto a veces significa que el resultado está mal expresado (un error típico es incluir hipótesis que son falsas, lo que hace el resultado trivialmente cierto). Incluso puede significar que la formalización no refleja con fidelidad aquello que se intenta modelar.

Aún cuando la prueba automática refleje la prueba buscada, un estudio de la misma puede llevarnos también a incluir algunos lemas previos. Por ejemplo, si en un resultado se usa inducción varias veces, puede que sea interesante incluir algunos de los subobjetivos demostrados por inducción, como lemas que se usarán en la demostración de posteriores resultados.

Sumario

En este capítulo, hemos visto una introducción al sistema ACL2, dividida en tres partes:

- El lenguaje de programación de ACL2 y su relación con Common Lisp.

- La lógica de ACL2, una lógica de primer orden sin cuantificadores y con un principio de inducción. Además, hemos descrito dos importantes principios de extensión, que permiten añadir nuevos axiomas, preservando la consistencia: el principio de definición y el principio de encapsulado.

- El demostrador automático, describiendo su funcionamiento y "El Método" más adecuado para interactuar con el mismo.

Capítulo 3

Términos y sustituciones

Este capítulo está dedicado a formalizar en ACL2 los conceptos de término y sustitución de primer orden. Tales conceptos son el principal objeto de estudio en la teoría desarrollada en esta memoria. La formalización presentada en este capítulo constituye la base sobre la cual se desarrollan los capítulos posteriores: el retículo de los términos, las teorías ecuacionales y los sistemas de reescritura.

En primer lugar, se discuten cuestiones relativas a la representación escogida en ACL2 para razonar sobre los términos de primer orden y la influencia que esta representación tiene sobre la teoría desarrollada y sobre la automatización de las demostraciones. Igualmente, se discute la representación de las sustituciones y de los sistemas de ecuaciones.

La segunda sección muestra algunas propiedades de los términos de primer orden relativas a la estructura de árbol de los mismos: los conceptos de posición, ocurrencia, subtérmino y reemplazamiento se definen en ACL2 y se demuestran algunas propiedades básicas relacionadas, que serán de utilidad más adelante.

En la siguiente sección, se define la relación de subsunción entre términos. Para ello es necesario verificar un algoritmo de equiparación entre pares de términos que encuentra, si existe, una sustitución que aplicada al primero obtiene el segundo. Tal algoritmo de equiparación se define mediante un conjunto de reglas de transformación que actúan sobre sistemas de ecuaciones, de manera no determinista. A partir de este algoritmo genérico, se obtiene un algoritmo de subsunción ejecutable, que aplica las reglas de transformación siguiendo una estrategia concreta. Las propiedades de este algoritmo se obtienen de manera directa de las propiedades del algoritmo no determinista, usando para ello la regla derivada de instanciación funcional. La corrección y completitud del algoritmo de equiparación definido nos permite expresar de manera constructiva la relación de subsunción entre términos y probar que es un preorden.

La relación de subsunción se puede extender de los términos a las sustituciones. El algoritmo de equiparación previamente definido nos ayudará a definir de manera constructiva la relación de subsunción entre sustituciones. En la cuarta sección se define y verifica en ACL2 tal concepto.

3.1 Términos de primer orden, sustituciones y ecuaciones

En esta sección presentamos los conceptos de término de primer orden y de sustitución, y su formalización en la lógica de ACL2. Aunque no se presenta ningún resultado de enver-

gadura, su importancia radica en sentar las bases para la representación de los elementos sobre los que se razonará más adelante. Las definiciones y teoremas que se presentan en esta sección se encuentran en el libro `terms.lisp`.

3.1.1 Preliminares

Presentamos en lo que sigue el tratamiento estándar que se hace del concepto de término de primer orden y de sustitución en cualquier libro de texto sobre la materia. Las definiciones de esta sección están tomadas esencialmente de [1].

Definición 3.1 *Una* **signatura** Σ *es una familia de conjuntos* $\langle \Sigma_n : n \in \omega \rangle$. *Si* $f \in \Sigma_n$ *para cierto* $n \in \omega$, *decimos que se trata de un* **símbolo de función** *de* **aridad** n. *Si* $c \in \Sigma_0$, *decimos que* c *es un* **símbolo de constante**.

Nótese que los conjuntos Σ_n no tienen que ser disjuntos entre sí, por lo que se permiten símbolos de función con más de una aridad (**aridad variable**). Supondremos en lo que sigue que cada Σ_n es un conjunto numerable.

Ejemplo 3.2 La signatura $\Sigma_G = \langle \Sigma_{G,n} : n \in \omega \rangle$, tal que $\Sigma_{G,0} = \{e\}$, $\Sigma_{G,1} = \{i\}$, $\Sigma_{G,2} = \{*\}$ y $\Sigma_{G,n} = \emptyset$ para $n > 2$, define el lenguaje de la *teoría de grupos*. Aquí, $*$ es un símbolo de aridad 2, i es un símbolo con aridad 1 y e es un símbolo de constante.

Definición 3.3 *Sea* Σ *una signatura y* X *un conjunto numerable (llamado conjunto de* **variables***). El conjunto de los* **términos de primer orden** *(o simplemente* **términos***) de tipo* Σ *sobre* X, *notado* $T(\Sigma, X)$, *se define recursivamente de la siguiente manera:*

- $X \subseteq T(\Sigma, X)$.
- *si* $f \in \Sigma_n$ *y* $t_1, \ldots, t_n \in T(\Sigma, X)$, *entonces* $f(t_1, \ldots, t_n) \in T(\Sigma, X)$.

Es decir, las variables son términos y mediante la aplicación de un símbolo de función de aridad n a n términos, se obtiene un nuevo término.

Ejemplo 3.4 Sea Σ_G la signatura del ejemplo 3.2 y $X = \{x, y, z\}$. Entonces $*(x, i(e()))$ y $*(x, *(i(z), e()))$ son dos términos de tipo Σ_G sobre X. A veces, los símbolos de función binarios (de aridad 2) se representarán de manera infija, usando paréntesis si fuera necesario. Además, si no existiera confusión con los símbolos de variable, los términos consistentes en un símbolo de constante, no necesitarán paréntesis. Teniendo en cuenta estas observaciones, los anteriores términos se podrán notar por $x * i(e)$ y $x * (i(z) * e)$, respectivamente.

En lo que sigue, supondremos que Σ es una signatura y que X es un conjunto de variables. Sin pérdida de generalidad, supondremos que X contiene a los símbolos x, y, z, u y v, y sus correspondientes versiones subindicadas. Además, reservaremos las letras a, b, c, d y e para los símbolos de constante.

La definición recursiva del conjunto $T(\Sigma, X)$, permite probar determinados resultados sobre términos mediante inducción en la estructura de los mismos. Además, permite la definición recursiva de funciones sobre términos. La inducción y recursión estructural en los términos queda justificada y descrita por los dos siguientes teoremas:

Teorema 3.5 (Principio de inducción en la estructura de los términos). *Sea P una propiedad definida sobre los elementos de $T(\Sigma, X)$ cumpliendo:*

- *$P(x)$ se verifica para todo $x \in X$ (**caso base**).*
- *Para todo $f \in \Sigma_n$ y $t_1, \ldots, t_n \in T(\Sigma, X)$ se tiene que $P(t_1), \ldots, P(t_n)$ implican $P(f(t_1, \ldots, t_n))$ (**caso inductivo**).*

Entonces se verifica $P(t)$ para todo $t \in T(\Sigma, X)$.

Teorema 3.6 (Definición por recursión en la estructura de los términos). *Sea A un conjunto y $h : X \to A$, $g : T(\Sigma, X) \times A^* \to A$ dos funciones. Entonces existe una única función $F : T(\Sigma, X) \to A$ tal que:*

- *$F(x) = h(x)$, si $x \in X$.*
- *$F(t) = g(t, (F(t_1), \ldots, F(t_n)))$, si $t = f(t_1, \ldots, t_n) \in T(\Sigma, X)$.*

Algunas de las siguientes funciones se definen por recursión en la estructura de los términos:

Definición 3.7 *Definimos las siguientes funciones sobre $T(\Sigma, X)$:*

- *El* **conjunto de variables** *de un término t, $\mathcal{V}(t)$:*

$$\mathcal{V}(t) = \begin{cases} \{x\} & \text{si } t = x \in X \\ \bigcup_{i=1}^{n} \mathcal{V}(t_i) & \text{si } t = f(t_1, \ldots, t_n) \end{cases}$$

  Por extensión, definimos el **conjunto de variables** *de un conjunto de términos T, $\mathcal{V}(T) = \bigcup_{t \in T} \mathcal{V}(t)$.*

- *La* **longitud** *de un término t, $l(t)$:*

$$l(t) = \begin{cases} 1 & \text{si } t = x \in X \\ 1 + \sum_{i=1}^{n} l(t_i) & \text{si } t = f(t_1, \ldots, t_n) \end{cases}$$

  (es decir, el número de símbolos de t).

- *El* **tamaño** *de un término t, $|t|$:*

$$|t| = \begin{cases} 0 & \text{si } t = x \in X \\ 1 + \sum_{i=1}^{n} |t_i| & \text{si } t = f(t_1, \ldots, t_n) \end{cases}$$

  (es decir, el número de símbolos de función de t).

- *La* **apertura de un término** *t, $v(t) = |\mathcal{V}(t)|$ (es decir, el número de variables distintas de t).*

Ejemplo 3.8 Si $t = f(x, g(x, y))$, entonces $\mathcal{V}(t) = \{x, y\}$, $v(t) = 2$, $l(t) = 5$ y $|t| = 2$.

Además del concepto de término de primer orden, otro de los conceptos que manejaremos constantemente será el de sustitución de primer orden, que definimos a continuación.

Definición 3.9 *Una* **sustitución** *en $T(\Sigma, X)$ (o simplemente una sustitución) es una función $\sigma : X \to T(\Sigma, X)$ tal que $\sigma(x) \neq x$ sólo para una cantidad finita de variables $x \in X$. El conjunto finito de variables tal que $\sigma(x) \neq x$ se denomina el* **dominio** *de σ y se nota por $\mathcal{D}(\sigma)$. El* **codominio** *de σ, notado $\mathcal{C}(\sigma)$, es el conjunto definido por $\mathcal{C}(\sigma) = \{\sigma(x) \mid x \in \mathcal{D}(\sigma)\}$.*

Toda sustitución σ en $T(\Sigma, X)$ se puede **extender** a una función $\widehat{\sigma}$ definida en todo el conjunto $T(\Sigma, X)$. La definición de $\widehat{\sigma}$ es recursiva en la estructura de los términos.

Definición 3.10 *Sea σ una $T(\Sigma, X)$-sustitución. Definimos la* **aplicación de la sustitución σ a un término** *$t \in T(\Sigma, X)$, $\widehat{\sigma}(t)$:*

- $\widehat{\sigma}(x) = \sigma(x)$, *si* $x \in X$
- $\widehat{\sigma}(f(t_1, \ldots, t_n)) = f(\widehat{\sigma}(t_1), \ldots, \widehat{\sigma}(t_n))$

Abusando de la notación, en lo sucesivo, escribiremos $\sigma(t)$ en lugar de $\widehat{\sigma}(t)$ y supondremos que una sustitución está definida en todo el conjunto $T(\Sigma, X)$.

Proposición 3.11 *Sea $t \in T(\Sigma, X)$ y σ una sustitución en $T(\Sigma, X)$. Entonces:*

a) Si σ' es una sustitución en $T(\Sigma, X)$ tal que para todo $x \in \mathcal{V}(t)$, $\sigma(x) = \sigma'(x)$, entonces $\sigma(t) = \sigma'(t)$.

b) Si $\mathcal{D}(\sigma) \cap \mathcal{V}(t) = \emptyset$, entonces $\sigma(t) = t$.

Demostración:
Ambos apartados se tienen de manera muy sencilla por inducción en la estructura del término t. Veamos el apartado a) (el apartado b) es análogo). Si $t = x \in X$, entonces se tiene por hipótesis. Si $t = f(t_1, \ldots, t_n)$, podemos suponer por hipótesis de inducción que $\sigma(t_i) = \sigma'(t_i)$ para $i \in \{1, \ldots, n\}$ (ya que σ y σ' coinciden en las variables de t_i). Entonces $\sigma(t) = \sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n)) = f(\sigma'(t_1), \ldots, \sigma'(t_n)) = \sigma'(t)$. □

Como muestra la anterior proposición, el hecho de considerar sustituciones de dominio finito no es una restricción importante, ya que usualmente estaremos interesados en el efecto que la sustitución tiene sobre un conjunto finito de términos y por tanto sobre un conjunto finito de variables. Además, esto nos permite una sencilla representación de una sustitución: si $\mathcal{D}(\sigma) \subseteq \{x_1, \ldots, x_n\}$, entonces es usual notar a σ de la siguiente manera:

$$\sigma = \{x_1 \mapsto \sigma(x_1), \ldots, x_n \mapsto \sigma(x_n)\}$$

Ejemplo 3.12 Un ejemplo de sustitución en $T(\Sigma_G, X)$ es

$$\delta = \{x \mapsto i(z), u \mapsto i(z), y \mapsto u * v, z \mapsto e\}$$

Si $t = x * (i(z) * y)$, entonces $\delta(t) = i(z) * (i(e) * (u * v))$

Definición 3.13 *Si σ y δ son sustituciones definidas en $T(\Sigma, X)$, entonces la* **composición** *de δ con σ, notado $\sigma\delta$, es la composición usual entre funciones.*

Nótese que la composición de dos sustituciones es una sustitución, ya que su dominio está contenido en el conjunto finito $\mathcal{D}(\sigma) \cup \mathcal{D}(\delta)$. El siguiente teorema nos muestra una definición alternativa de la composición de sustituciones.

Teorema 3.14 *Sean σ y δ tales que $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ y $\delta = \{y_1 \mapsto s_1, \ldots, y_m \mapsto s_m\}$. Entonces*

$$\sigma\delta = \{y_1 \mapsto \sigma(s_1), \ldots, y_m \mapsto \sigma(s_m), x_{i_1} \mapsto t_{i_1}, \ldots, x_{i_k} \mapsto t_{i_k}\}$$

donde $x_{i_1}, \ldots, x_{i_k}$ son las variables de $\mathcal{D}(\sigma)$ que no están en $\mathcal{D}(\delta)$.

Demostración:
Notemos por τ la sustitución

$$\{y_1 \mapsto \sigma(s_1), \ldots, y_m \mapsto \sigma(s_m), x_{i_1} \mapsto t_{i_1}, \ldots, x_{i_k} \mapsto t_{i_k}\}$$

Hemos de probar que para todo $t \in T(\Sigma, X)$, $\tau(t) = \sigma(\delta(t))$. Lo haremos por inducción en la estructura de los términos:

Para el caso base de la inducción, hemos de probar que si $t = x \in X$, entonces $\tau(x) = \sigma(\delta(x))$. Hemos de distinguir dos casos. Si $x = y_i \in \mathcal{D}(\delta)$, entonces es evidente que $\tau(x) = \sigma(s_i) = \sigma(\delta(x))$. Si por el contrario $x \notin \mathcal{D}(\delta)$, entonces $\tau(x) = \sigma(x) = \sigma(\delta(x))$.

Caso inductivo: supongamos que $t = f(t_1, \ldots, t_n)$ y que para $i \in \{1, \ldots, n\}$, se tiene que $\tau(t_i) = \sigma\delta(t_i)$. Entonces, $\sigma(\delta(t)) = \sigma(\delta(f(t_1, \ldots, t_n))) = f(\sigma\delta(t_1), \ldots, \sigma\delta(t_n)) = f(\tau(t_1), \ldots, \tau(t_n)) = \tau(t)$. □

Definición 3.15 *Si $A \subseteq X$ es un conjunto finito de variables y σ una sustitución definida en $T(\Sigma, X)$, entonces la* **restricción** *de σ a A, notada por $\sigma_{|A}$, es la sustitución tal que $\sigma_{|A}(x) = \sigma(x)$ para toda variable $x \in A$ y $\sigma(x) = x$ para el resto de variables.*

Bajo determinadas condiciones, es posible definir la sustitución unión de dos sustituciones, operación que difiere un poco de la unión conjuntista de funciones:

Definición 3.16 *Sean $\sigma = \{y_1 \mapsto s_1, \ldots, y_m \mapsto s_m\}$ y $\tau = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ dos sustituciones tales que $\mathcal{D}(\sigma) \cap \mathcal{D}(\tau) = \emptyset$. Entonces definimos la sustitución* **unión** *de ambas, notada $\sigma \cup \tau$, como la sustitución*

$$\sigma \cup \tau = \{y_1 \mapsto s_1, \ldots, y_m \mapsto s_m, x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}.$$

A continuación definimos los conceptos de ecuación y de sistema de ecuaciones.

Definición 3.17 *Una* **ecuación** *en $T(\Sigma, X)$ es un par (s, t), notado $s \approx t$, tales que $s, t \in T(\Sigma, X)$. Decimos que s y t son los* **lados izquierdo** *y* **derecho**, *respectivamente, de la ecuación $s \approx t$. Un* **sistema de ecuaciones** *en $T(\Sigma, X)$ es un conjunto finito de ecuaciones en $T(\Sigma, X)$.*

Ejemplo 3.18 Un ejemplo de sistema de ecuaciones en $T(\Sigma_G, X)$ es

$$G = \{i(x) * x \approx e, (x * y) * z \approx x * (y * z), e * x \approx x\}$$

El conjunto de ecuaciones G sirve como conjunto de axiomas para la teoría de grupos libres.

Podemos extender algunas definiciones dadas para términos, a las análogas para sistemas de ecuaciones.

Definición 3.19 *Dado un sistema de ecuaciones* $S = \{s_1 \approx t_1, \ldots, s_n \approx t_n\}$*, definimos:*

- *El* **conjunto de variables** *de* S*, notado* $\mathcal{V}(S)$*, es* $\mathcal{V}(S) = \bigcup_{i=1}^{n}(\mathcal{V}(s_i) \cup \mathcal{V}(t_i))$.
- *El* **tamaño** *de* S*, notado* $t(S)$*, es* $t(S) = \sum_{i=1}^{n}(|s_i| + |t_i|)$.
- *La* **aplicación de una sustitución** σ *a* S*, notado* $\sigma(S)$*, es el sistema* $\sigma(S) = \{\sigma(s_1) \approx \sigma(t_1), \ldots, \sigma(s_n) \approx \sigma(t_n)\}$

Una determinada clase de sistemas de ecuaciones se pueden ver como sustituciones, lo que queda descrito por la siguiente definición:

Definición 3.20 *Un* **sistema–sustitución** *(en* $T(\Sigma, X)$*) es un sistema de ecuaciones (en* $T(\Sigma, X)$*) tal que los lados izquierdos de todas sus ecuaciones son variables y tal que no tiene dos ecuaciones distintas con el mismo lado izquierdo. Si el sistema* $S = \{x_1 \approx t_1, \ldots, x_n \approx t_n\}$ *es un sistema–sustitución, la* **sustitución asociada** *a* S*, notada* $\vec{S}$*, se define por* $\vec{S} = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$

3.1.2 Representación de los términos de primer orden en ACL2

La idea general para representar los términos de primer orden en ACL2 es usar listas y notación prefija. Por ejemplo, el término $f(x, h(y), g(h(h(u))))$, donde f, g y h son símbolos de función, será representado por el objeto ACL2 `(f x (h y) (g (h (h u))))`. De la misma manera, el término $x*(i(z)*e)$ se representará por `(* x (* (i z) (e)))`. En general, el término $f(t_1, \ldots, t_n)$ será representado por la lista `(f` $O_1 \ldots O_n$`)`, donde $O_1, \ldots, O_n$ son, respectivamente, los objetos ACL2 que representan a los términos $t_1, \ldots, t_n$. En lo que sigue, se presentan las definiciones que formalizan esta idea intuitiva.

Términos de primer orden respecto de una signatura

Antes de dar la definición formal de término de primer orden, debemos formalizar el concepto de signatura. Representaremos una signatura Σ identificándola con una función binaria `signat`. El significado intuitivo de tal función es el siguiente: `(signat f n)` es distinto de `nil` si y sólo si `f` es un símbolo de la signatura, de aridad `n`. Puesto que la teoría que se pretende desarrollar no se refiere a ninguna signatura en particular, usaremos el comando `defstub` para poder razonar sobre tal función `signat`, sin definirla de manera particular:

```
(defstub signat (* *) => *)
```

El evento `defstub` permite definir un nuevo símbolo de función en la lógica de ACL2. No se asume ningún axioma sobre el mismo, pero permite usar el símbolo (`signat` en este caso) como una función (en este caso de dos argumentos de entrada y uno de salida). Los teoremas obtenidos para esta signatura general, se pueden instanciar fácilmente para signaturas concretas, sin más que hacer uso de la regla de inferencia derivada de instanciación funcional. Por ejemplo, en el caso de la teoría de grupos, la función que definiría la correspondiente signatura podría ser:

```
(defun signat-g (symb n)
  (let* ((sigma '((* 2) (i 1) (e 0)))
         (found (assoc symb sigma))
         (arity (cdr found)))
    (and found (member n arity))))
```

Una vez definido el concepto de signatura, vamos a definir los objetos ACL2 que representan a los términos de primer orden en una signatura.

En primer lugar, las variables serán representadas por aquellos objetos ACL2 que verifiquen el predicado `acl2-numberp` (números) o bien `symbolp` (símbolos) o bien `characterp` (caracteres). En ACL2, tales objetos se reconocen por el predicado `eqlablep`[1] (predefinido). La macro[2] `variable-s-p`, por tanto, reconoce a los objetos que representarán a los términos variables (nótese que la definición es independiente de la signatura):

```
(defmacro variable-s-p (x) `(eqlablep ,x))
```

Para definir aquellos objetos ACL2 que van a representar a los términos en una signatura, definiremos también al mismo tiempo y de manera mutuamente recursiva las *listas de términos*. La función `term-s-p-aux`, que recibe como argumentos un indicador `flg` y un objeto `x`, implementa un reconocedor de los objetos que representan un término o una lista de términos en una signatura:

```
(defun term-s-p-aux (flg x)
  (if flg
      (if (atom x)
          (variable-s-p x)
        (if (signat (car x) (len (cdr x)))
            (term-s-p-aux nil (cdr x))
          nil))
    (if (atom x)
        (equal x nil)
      (and (term-s-p-aux t (car x))
           (term-s-p-aux nil (cdr x))))))
```

La interpretación de tal función es la siguiente:

- Si `flg` es distinto de `nil`, `(term-s-p-aux flg x)` es `t` si y sólo si `x` representa a un término (en una signatura).
- `(term-s-p-aux nil x)` es `t` si y sólo si `x` representa a una lista de términos (en una signatura).

[1] Tal nombre es debido a que son esos precisamente los objetos que en ejecución se pueden comparar mediante el predicado de igualdad `eql`.

[2] Recuérdese que `defmacro` (página 2.1) nos permite definir abreviaturas. En este caso, nos permite usar `variable-s-p` en lugar de `eqlablep`, mejorando la legibilidad de las fórmulas. Usaremos `defmacro` en este sentido a lo largo de la memoria. Desde el punto de vista lógico es lo mismo que usar `defun`. Sin embargo, su comportamiento respecto al mecanismo de reescritura del demostrador es distinto.

Según esta definición, un objeto ACL2 representa un término de primer orden en una signatura si es atómico y verifica el predicado `variable-s-p` (en cuyo caso representa a una variable), o bien si no es atómico, su `cdr` es una lista de términos en dicha signatura (la lista de argumentos) y su `car` es un símbolo de la signatura, de aridad igual a la longitud de la lista de sus argumentos (en cuyo caso representa un término no variable). Como se observa, es análoga a la definición 3.3. Por otro lado, un objeto ACL2 representa a una lista de términos de primer orden en una signatura si es una lista propia tal que cada uno de sus elementos es un término en la signatura.

El motivo de restringir la representación de variables a objetos que verifiquen el predicado `eqlablep` es que de esta manera se mejora la eficiencia de las funciones que actúan sobre términos de primer orden (permitiendo usar `eql` en lugar de `equal`). La misma razón justifica el restringir las listas de argumentos a listas propias (permitiendo usar `endp` en lugar de `atom`). En el apéndice B se comenta esta cuestión con detalle.

El esquema mutuamente recursivo de `term-s-p-aux` se usa con frecuencia en nuestra formalización, para definir funciones sobre términos: se introduce un argumento extra `flg`, indicando si estamos ante un objeto que representa a un término (`flg`$\neq$`nil`), o bien ante un objeto que representa a una lista de términos (`flg`=`nil`). Al considerar un término no variable, usualmente tendremos que analizar recursivamente la lista de sus argumentos. De la misma manera, al analizar una lista de términos, cada uno de sus elementos será analizado como término. Nótese que este tipo de definiciones son *definiciones por recursión en la estructura de los términos*, tal y como se presentó en el teorema 3.6. Tales definiciones por recursión se admiten por el principio de definición de la lógica de ACL2. La prueba de terminación de funciones definidas usando este tipo de recursión estructural es sencilla y automática, ya que todo elemento que no verifica el predicado `atom` verifica el predicado `consp` y por tanto su `car` y su `cdr` son objetos cuyo valor respecto a la función `acl2-count` es menor.

El hecho de definir a la misma vez términos y listas de términos, mediante recursión mutua, permitirá que muchos de los teoremas que se presentan en lo sucesivo establezcan, a la misma vez, que ciertas propiedades son ciertas tanto para términos como para listas de términos. Además, esta definición mutuamente recursiva será especialmente adecuada para que el demostrador automático seleccione durante un intento de prueba de un teorema, un esquema de inducción muy cercano a la inducción estructural, siendo éste en la mayoría de los casos el adecuado para obtener una prueba (véase 3.1.3).

En cualquier caso, nuestro interés primordial se centrará en los objetos que representan términos. Es por ello que definimos la siguiente macro `term-s-p`, que reconocerá a aquellos objetos que (en nuestra formalización) representan a los términos de primer orden en una signatura.

```
(defmacro term-s-p (x) `(term-s-p-aux t ,x))
```

Términos propios e impropios

La lógica de ACL2 es una lógica de funciones totales. Esto significa que toda función definida en ACL2 tiene su resultado completamente especificado para cualquier dato de entrada. Así, si queremos definir formalmente una función ACL2 que actúe sobre un conjunto de términos de primer orden, no podemos restringir su definición al conjunto de objetos ACL2 que representan tales términos. La función también tendrá un valor definido sobre objetos que no representan términos en ninguna signatura. Por ejemplo,

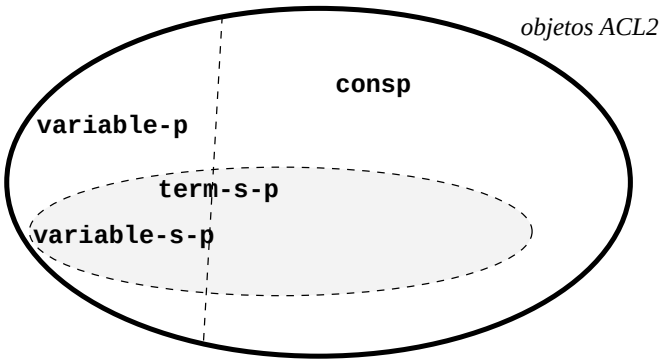

Figura 3.1: Representación de los términos de primer orden en ACL2

el par punteado `(1 . 2)` no representa a ningún término de primer orden en ninguna signatura, según la definición dada por `term-s-p`. Sin embargo cualquier función `fun` que definamos asignará un resultado a `(fun '(1 . 2))`, aún cuando el dominio en el que se pretende definir `fun` sea un conjunto de términos de primer orden.

Lejos de ser un problema, en nuestro caso el hecho de que ACL2 sea una lógica de funciones totales representa una ventaja, como veremos. Al definir funciones que actúen sobre términos, bastará con hacer que actúen de una manera "consistente" sobre aquellos objetos que no representan términos de primer orden. Por "consistente" entendemos que los teoremas que se obtengan sean ciertos tanto para aquellos objetos que representan términos como para aquellos que no los representan.

Un punto clave en nuestra formalización es que *todo objeto O de la lógica de ACL2 puede ser visto como la representación de un término de primer orden en sentido amplio*:

- Si `(atom` O`)`, entonces O representa a una variable en sentido amplio.
- En caso contrario, se verifica `(consp` O`)` y O representa al término, en sentido amplio, cuyo símbolo de función principal es `(car` O`)` y tal que `(cdr` O`)` representa a la lista de sus argumentos.

Por analogía con la definición de `variable-s-p`, podemos definir una función `variable-p` que reconozca cuándo un objeto ACL2 puede ser visto como una variable en sentido amplio:

```
(defun variable-p (x) (atom x))
```

Por supuesto, tal definición no es necesaria, ya que podríamos usar `atom`. Sin embargo, mejora la legibilidad de las definiciones y teoremas presentados a continuación, enfatizando el hecho de que todo objeto que no verifique `consp` se considerará como una variable en sentido amplio.

El gráfico de la figura 3.1 ilustra la situación descrita. Todos los objetos ACL2 representan a un término en sentido amplio. El predicado `variable-p` produce una partición en dos del conjunto de tales objetos: aquellos que representan variables (los que verifican `atom`) y aquellos que representan la aplicación de un símbolo de función a sus argumentos (los que verifican `consp`). Dada una signatura, los objetos que representan los términos de

primer orden en dicha signatura (los que verifican `term-s-p`) constituyen un subconjunto del total. Éstos, a su vez, se dividen en aquellos que representan a los términos variables en la signatura (los que verifican `variable-s-p`) y aquellos que representan la aplicación de una función de la signatura a sus argumentos.

Diremos que un objeto ACL2 representa un término **impropio** si no representa a un término de primer orden, sea cual sea la signatura, según la definición implementada por `term-s-p`[3]. En caso contrario, se dice que representa a un término **propio**. De manera análoga se pueden definir listas de términos propias e impropias. En la tabla de la figura 3.2, se dan varios ejemplos de objetos ACL2 y, en su caso, los términos propios que representan.

Objeto ACL2	Propio	Término representado
`x`	Sí	Una variable
`"hola"`	No	-
`12`	Sí	Una variable
`(f (g x) (h y))`	Sí	$f(g(x), h(y))$
`(f (g x x) (g x (g y) z))`	Sí	$f(g(x,x), g(x, g(y), z))$
`(f x y . z)`	No	-
`(f x (f y . 1))`	No	-
`(f x (h (g z) . 1) u)`	No	-

Figura 3.2: Algunos ejemplos de términos propios e impropios

Este punto de vista "ampliado" de nuestra representación de los términos de primer orden en ACL2 supone una importante ayuda intuitiva para comprender mejor las definiciones y teoremas que aquí se presentan. Las principales ventajas de esta generalización del concepto de término son:

- La teoría obtenida es más general y los resultados obtenidos no son ciertos sólamente para términos propios.
- Las hipótesis de la forma `(term-s-p term)` o de la forma `(term-s-p-aux flg term)` no son necesarias, en la mayoría de los casos. Esto hará que las demostraciones obtenidas no tengan una distinción de casos innecesaria (véase 3.1.3).
- El hecho de carecer de estas hipótesis innecesarias hará que las reglas de reescritura obtenidas a partir de los teoremas se apliquen más eficientemente.

Por supuesto, para que la teoría computacional que aquí se presenta refleje adecuadamente lo que se pretende formalizar, las funciones definidas han de verificar ciertas *propiedades de clausura*: deben ser cerradas respecto de los términos de una signatura. Por ejemplo, supongamos que la función `fun` formaliza en ACL2 una determinada función $F : T(\Sigma, X) \to T(\Sigma, X)$. Entonces, si T representa a un término en la signatura Σ, entonces el resultado devuelto por `(fun` T`)` debería representar a su vez a un término en la misma signatura. Estas propiedades de clausura se demostrarán como teoremas para las principales funciones definidas.

[3]Más precisamente, si para cualquier función `signat1` que defina una signatura, la correspondiente instancia funcional de `term-s-p` no lo reconoce como término en dicha signatura

Algunas funciones sobre términos

Veamos en este apartado algunas funciones definidas sobre términos. Tales definiciones, además de introducir conceptos que nos serán útiles más adelante, ilustran las cuestiones discutidas anteriormente en esta subsección.

La función `variables` obtiene la lista de las variables de un término (o lista de términos). Nótese que se trata de una definición por recursión en la estructura de los términos:

```
(defun variables (flg term)
  (if flg
      (if (variable-p term)
          (list term)
        (variables nil (cdr term)))
    (if (endp term)
        nil
      (append (variables t (car term)) (variables nil (cdr term))))))
```

Como ya se ha discutido, la función `variables` define la lista de variables tanto para términos (y listas de términos) propios como impropios, lo cual se ilustra en los siguientes ejemplos:

```
ACL2 !>(variables t '(f (g x x) (g x (g y) z)))
(X X X Y Z)
ACL2 >(variables t '(f x (h (g z) . v) u))
(X Z U)
```

Más adelante será necesario también obtener el conjunto de variables de un término (o lista de términos), sin repeticiones. La siguiente macro `variables-set` define tal concepto (la función `make-set`, omitida aquí, elimina las repeticiones de una lista). De esta manera, también podemos definir el número de variables distintas de un término (o lista de términos), dado por la macro `n-variables`:

```
(defmacro variables-set (flg term) `(make-set (variables ,flg ,term)))

(defmacro n-variables (flg term) `(len (variables-set ,flg ,term)))
```

La función `size` define el tamaño de un término o lista de términos (es decir, el número de símbolos no variables). La función `length-term` define la longitud de un término o lista de términos (es decir, el número total de símbolos de un término):

```
(defun size (flg term)
  (if flg
      (if (variable-p term)
          0
        (+ 1 (size nil (cdr term))))
    (if (endp term)
        0
      (+ (size t (car term)) (size nil (cdr term))))))
```

```
(defun length-term (flg term)
  (if flg
      (if (variable-p term)
          1
        (1+ (length-term nil (cdr term))))
    (if (endp term)
        0
      (+ (length-term t (car term)) (length-term nil (cdr term))))))
```

3.1.3 Representación de sustituciones en ACL2

Representaremos las sustituciones como una lista de asociación, que asigna un término a cada variable de su dominio. Por ejemplo, la sustitución $\delta = \{x \mapsto i(z), u \mapsto i(z), y \mapsto u * v, z \mapsto e\}$, la representaremos mediante la lista de asociación `((x . (i z)) (u . (i z)) (y . (* u v)) (z . (e)))`. En lo que sigue, formalizamos esta idea.

Sustituciones y su aplicación a los términos

Representaremos una sustitución en $T(\Sigma, X)$ como una lista de asociación que asocia términos de tipo Σ a variables, tal y como define la siguiente función `substitution-s-p`:

```
(defun substitution-s-p (l)
  (if (atom l)
      (equal l nil)
    (and (consp (car l)) (variable-s-p (caar l)) (term-s-p (cdar l))
         (substitution-s-p (cdr l)))))
```

Definamos a continuación cómo se aplican sustituciones a términos. La función `val` define el valor de una variable respecto de una sustitución. A diferencia de las listas de asociación en Common Lisp, el valor de una variable que no se encuentra dentro del dominio es la misma variable[4]:

```
(defun val (x sigma)
  (if (endp sigma)
      x
    (if (eql x (caar sigma))
        (cdar sigma)
      (val x (cdr sigma)))))
```

Existen algunas particularidades en esta representación respecto de la noción estándar de sustitución, dada en los preliminares:

- La más importante es que estamos representando una función mediante una lista de asociación. Esto hace que distintas listas de asociación representen a la misma sustitución. Por ejemplo, el orden de los pares en la lista de asociación es irrelevante

[4] Este es el motivo por el cual no usamos la función Common Lisp `assoc`.

desde el punto de vista funcional y, sin embargo, hace que dos listas de asociación no sean consideradas iguales. En otras palabras, el predicado `equal` de ACL2 no sirve como predicado para comprobar la igualdad funcional entre dos sustituciones. Más adelante discutiremos este punto.

- En nuestra representación se permite que una misma variable tenga asociados dos valores distintos. En este caso, consideraremos relevante sólo la primera asignación (exactamente igual que en Common Lisp). Por ejemplo, la sustitución representada por `'((x . (h z)) (x . (f y)))` es, desde el punto de vista funcional, la misma sustitución que `'((x . (h z))`.

- Permitiremos asociaciones de la forma `(x . x)`, que asignan una variable a sí misma. Esto no afecta al comportamiento funcional de una sustitución.

Como ya se ha discutido para el caso de los términos, aunque hemos descrito los objetos ACL2 que representan a sustituciones en una signatura dada, el hecho de que ACL2 sea una lógica de funciones totales, hace que `(val x sigma)` tenga un valor completamente especificado, aunque `sigma` no represente a una sustitución en ninguna signatura[5]. Así, como en el caso de los términos, es posible ver cualquier objeto ACL2 como la representación de una sustitución, cuyo comportamiento funcional viene determinado por la definición de `val`. Los siguientes ejemplos muestran como actúa la la función `val` cuando su segundo argumento no representa a ninguna sustitución (en el sentido definido por `substitution-s-p`):

```
ACL2 >(val 'x 3)
X
ACL2 >(val 'x '((y . (f z)) (z . (f (g u))) 6))
X
ACL2 >(val 'z '((y . (f z)) (z . (f (g u . 1)))))
(F (G U . 1))
ACL2 >(val nil '((x . (e)) 3 (nil . (h x))))
NIL
```

Como en el caso de los términos, las funciones estarán definidas de una manera "consistente" sobre aquellos objetos que no representan sustituciones, de tal manera que los teoremas que se obtengan serán ciertos tanto para aquellos objetos que representan sustituciones como para aquello que no los representan (es decir, las hipótesis de la forma `(substitution-s-p sigma)` no serán necesarias[6]).

La función `apply-subst` define la aplicación de una sustitución a un término o lista de términos, mediante recursión estructural. Como caso particular, `instance` define la aplicación de una sustitución a un término:

```
(defun apply-subst (flg sigma term)
  (if flg
      (if (variable-p term)
          (val term sigma)
```

[5] Por ejemplo, listas con elementos atómicos, o con pares cuyo primer elemento no sea una variable.
[6] También como en el caso de los términos, se demostrarán las propiedades de clausura pertinentes.

```
        (cons (car term)
              (apply-subst nil sigma (cdr term))))
    (if (endp term)
        term
      (cons (apply-subst t sigma (car term))
            (apply-subst nil sigma (cdr term))))))

(defmacro instance (term sigma) `(apply-subst t ,sigma ,term))
```

El carácter funcional de una sustitución

Como ya se ha señalado anteriormente, una misma sustitución puede ser representada por distintos objetos ACL2. Por ejemplo, las siguientes listas de asociación:

```
((x . (h y))  (y . (g z)))

((x . (h y))  (x . (k u))  (y . (g z)))

((x . (h y))  (z . z)  (y . (g z)))

(3  (x . (h y))  (y . (g z)))

(((h y) . (h y))  (y . (g z))  (x . (h y)))
```

representan a la sustitución $\sigma = \{x \mapsto h(y), y \mapsto g(z)\}$, en el sentido de que para cualquier `x` tal que `(variable-p x)`, el comportamiento de estas listas de asociación respecto de la función `val` es idéntico.

Este ejemplo muestra por qué el predicado `equal` de ACL2 no sirve para describir la igualdad entre sustituciones. Si queremos establecer la igualdad funcional entre dos sustituciones representadas por `sigma1` y `sigma2`, respectivamente, habrá que probar que para toda variable `x`, se tiene `(equal (val x sigma1) (val x sigma2))`. La ausencia de cuantificadores en la lógica de ACL2 hace que esta manera de expresar la igualdad funcional entre sustituciones no sea adecuada cuando se quiere asumir como antecedente en una implicación. En ese caso, una solución es el uso del mecanismo de encapsulado, tal y como se nuestra en la subsección 3.4.3.

A veces, sin embargo, una noción restringida de igualdad nos bastará. Esta igualdad restringida queda definida por la función `coincide` y expresa que dos sustituciones se comportan de igual manera (respecto de `val`), sobre los elementos de una lista dada:

```
(defun coincide (sigma1 sigma2 l)
  (if (atom l)
      T
    (and (equal (val (car l) sigma1)
                (val (car l) sigma2))
         (coincide sigma1 sigma2 (cdr l)))))
```

A continuación, damos las definiciones de algunas funciones relacionadas con el carácter funcional de las sustituciones. Las funciones `domain` y `co-domain` implementan los conceptos de dominio y codominio, respectivamente.

```
(defun domain (sigma)
  (if (atom sigma)
      nil
    (cons (caar sigma) (domain (cdr sigma)))))

(defun co-domain (sigma)
  (if (atom sigma)
      nil
    (cons (cdar sigma) (co-domain (cdr sigma)))))
```

La función `restriction` define la restricción de una sustitución al conjunto de los elementos de una lista dada:

```
(defun restriction (sigma l)
  (if (atom l)
      l
    (cons (cons (car l) (val (car l) sigma))
          (restriction sigma (cdr l)))))
```

Podemos definir también el concepto de extensión de una sustitución. Diremos que una sustitución *extiende* a otra si ambas se comportan de la misma manera para las variables del dominio de la segunda. Es precisamente lo que define la siguiente macro llamada `extension`:

```
(defmacro extension (sigma1 sigma)
  `(coincide ,sigma ,sigma1 (domain ,sigma)))
```

Y finalmente, la función `composition` implementa la composición de dos sustituciones:

```
(defun composition (sigma1 sigma2)
  (if (endp sigma2)
      sigma1
    (cons (cons (caar sigma2) (apply-subst t sigma1 (cdar sigma2)))
          (composition sigma1 (cdr sigma2))))))
```

Un primer teorema

En esta subsección presentamos el primer teorema dentro de nuestra formalización. Aunque no es un resultado particularmente difícil de probar, al ser el primer resultado de la teoría sobre términos de primer orden que presentamos en esta memoria, analizaremos detalladamente su formalización y prueba automática en el demostrador.

El resultado sirve para verificar formalmente la definición de la función `composition`. Nótese que `(composition sigma1 sigma2)` obtiene la lista de asociación que asocia a cada elemento del dominio de `sigma2` el término obtenido aplicando `sigma1` al término que asocia `sigma2` a tal elemento. Además, después de estas asociaciones, se incluyen las asociaciones que realiza `sigma1`. Tal definición está basada en el teorema 3.14[7]. La

[7] Nótese que no es necesario eliminar aquellas variables del dominio de `sigma1` que estén en el dominio de `sigma2`, debido a la representación escogida: cuando hay asignaciones repetidas, sólo la primera se tiene en cuenta.

propiedad fundamental que ha de verificar `(composition sigma1 sigma2)` es que sea la representación de la composición funcional de las sustituciones representadas por `sigma1` y `sigma2`. Formalmente:

```
;(defthm composition-of-substitutions-apply-v0
;        (implies (and (term-s-p term)
;                      (substitution-s-p sigma1)
;                      (substitution-s-p sigma2))
;                 (equal (instance term (composition sigma1 sigma2))
;                        (instance (instance term sigma2) sigma1))))
```

Sin embargo, esta resultado puede ser generalizado en dos aspectos. En primer lugar, el resultado resulta ser válido tanto para términos como para listas de términos. En segundo lugar, como ya hemos comentado anteriormente, el resultado resulta ser también válido si no se incluyen las hipótesis, que restringen a `term`, `sigma1` y `sigma2` a objetos que representen términos y sustituciones en una signatura. De esta manera, el resultado finalmente probado, del cual el anterior es un caso particular, se formula de la siguiente manera:

```
(defthm composition-of-substitutions-apply
  (equal (apply-subst flg (composition sigma1 sigma2) term)
         (apply-subst flg sigma1 (apply-subst flg sigma2 term))))
```

La demostración de este resultado se obtiene de manera muy similar a la dada en el teorema 3.14, mediante inducción en la estructura de los términos.

Veamos a continuación algunos detalles técnicos sobre cómo ACL2 lleva a cabo la demostración de este teorema. Observemos en primer lugar que el sistema genera un intento de prueba por inducción, usando para ello el siguiente esquema:

```
; (AND (IMPLIES (AND (NOT FLG)                                  ;;; *4*
;                    (NOT (ENDP TERM))
;                    (:P T SIGMA1 SIGMA2 (CAR TERM))
;                    (:P NIL SIGMA1 SIGMA2 (CDR TERM)))
;               (:P FLG SIGMA1 SIGMA2 TERM))
;      (IMPLIES (AND (NOT FLG) (ENDP TERM))                     ;;; *3*
;               (:P FLG SIGMA1 SIGMA2 TERM))
;      (IMPLIES (AND FLG (NOT (VARIABLE-P TERM))                ;;; *2*
;                    (:P NIL SIGMA1 SIGMA2 (CDR TERM)))
;               (:P FLG SIGMA1 SIGMA2 TERM))
;      (IMPLIES (AND FLG (VARIABLE-P TERM))                     ;;; *1*
;               (:P FLG SIGMA1 SIGMA2 TERM))).
```

Esta inducción viene dada por el esquema recursivo de la función `apply-subst` y está sugerida por el término `(apply-subst flg sigma2 term)` que aparece en la conjetura. Este esquema resulta ser el adecuado para la prueba del teorema y recuerda al *principio de inducción en la estructura de los términos*, tal y como se presenta en el teorema 3.5, excepto que nos permite probar la propiedad `:P` tanto para términos como para listas de términos. El caso `*1*` se corresponde con el caso base en la inducción estructural y el

caso *2* se corresponde con el paso de inducción. Los casos *3* y *4* se corresponden, respectivamente, con el caso base y el paso de inducción para listas de términos y vienen a ser una inducción en la longitud de la lista de argumentos. La validez de este esquema de inducción viene justificada por la prueba de la terminación de la función `apply-subst`, que a su vez se justifica mediante una medida del tamaño de los objetos que representan a los términos. Un gran número de los teoremas de la teoría que se presenta en esta memoria han sido demostrados por ACL2 usando esquemas de inducción análogos a éste, basados en la estructura de los términos.

Con este esquema de inducción (generado automáticamente), ACL2 obtiene la prueba del teorema `composition-of-substitutions-apply` sin ayuda por parte del usuario. El esquema de inducción genera cuatro subobjetivos, correspondiendo a los casos *1* a *4*. Los subobjetivos correspondientes a *4*, *3* y *2* se demuestran trivialmente, usando la definición de `apply-subst`. El subobjetivo correspondiente a *1*, queda reducido a la siguiente conjetura, que el sistema prueba automáticamente.

```
; Subgoal *1/1'
; (IMPLIES (AND FLG (VARIABLE-P TERM))
;          (EQUAL (APPLY-SUBST T SIGMA1 (VAL TERM SIGMA2))
;                 (APPLY-SUBST FLG SIGMA1 (VAL TERM SIGMA2)))).
```

Este resultado correspondiente al caso base de la inducción para términos, resulta de interés por sí mismo y puede ser usado como regla de reescritura en sucesivas pruebas. Es por esto que lo probamos como teorema independiente:

```
(defthm value-composition
  (implies (variable-p x)
           (equal (val x (composition sigma1 sigma2))
                  (apply-subst t sigma1 (val x sigma2)))))
```

Es importante destacar que el sistema ha "descubierto" el lema por sí mismo. Resulta interesante también el esquema de inducción generado por el demostrador en la prueba del teorema `value-composition`:

```
; (AND (IMPLIES (AND (NOT (ENDP SIGMA2))
;                    (NOT (EQUAL (CAAR SIGMA2) TERM))
;                    (:P FLG SIGMA1 (CDR SIGMA2) TERM))
;               (:P FLG SIGMA1 SIGMA2 TERM))
;      (IMPLIES (AND (NOT (ENDP SIGMA2))
;                    (EQUAL (CAAR SIGMA2) TERM))
;               (:P FLG SIGMA1 SIGMA2 TERM))
;      (IMPLIES (ENDP SIGMA2)
;               (:P FLG SIGMA1 SIGMA2 TERM))).
```

Nótese cómo este esquema de inducción distingue que la variable `term` esté en el dominio de `sigma2` o no, de manera que la prueba obtenida es muy similar a la prueba a mano que se lleva a cabo en el teorema 3.14 para el caso base de la inducción estructural.

Como se apuntó anteriormente, la formulación del teorema tal y como aparece en `composition-of-substitutions-apply` influye de manera notable en la obtención de la prueba automática por parte de ACL2. Discutamos esto con más detalle.

En primer lugar, obsérvese de qué manera afecta el expresar el teorema tanto para términos como para listas de términos, frente a una formulación sólo para términos. El siguiente esquema de inducción, que resulta fallido, es el generado por el sistema si en lugar del teorema `composition-of-substitutions-apply` intentamos probar el teorema `composition-of-substitutions-apply-v0`, presentado anteriormente:

```
;(AND (IMPLIES (AND T (NOT (VARIABLE-P TERM))      ;;; *2*
;                   (:P SIGMA1 SIGMA2 (CDR TERM)))
;              (:P SIGMA1 SIGMA2 TERM))
;     (IMPLIES (AND T (VARIABLE-P TERM))           ;;; *1*
;              (:P SIGMA1 SIGMA2 TERM))).
```

La razón por la cual este esquema de inducción no resulta adecuado para probar el resultado, reside en una mala elección de la hipótesis de inducción, ya que se está intentando probar una propiedad sobre *términos* y como hipótesis de inducción la propiedad se asume cierta para `(cdr term)`, que es una *lista de términos*.

Por otro lado, la inclusión de hipótesis sobre la forma de los objetos involucrados (usando `term-p-s` o `substitution-s-p`), no aporta nada a este teorema, ya que sigue siendo cierto aún sin asumir tales hipótesis. Por ejemplo, podríamos haber incluido en el teorema `composition-of-substitutions-apply` la hipótesis `(term-s-p-aux flg term)`. En este caso, se generaría un esquema de inducción adecuado, y el teorema se probaría igualmente de manera automática. Sin embargo, se generarían ocho subobjetivos, en lugar de cuatro. Los cuatro subobjetivos de más que aparecerían, se deberían a que al incluir más premisas, las hipótesis de inducción necesitan verificar más propiedades para ser aplicadas.

Otras propiedades básicas

Lo que sigue son algunas propiedades sencillas sobre sustituciones y términos, que usaremos más adelante. Las dos primeras formalizan los resultados de la proposición 3.11 (tanto para términos como para listas de términos). La función `disjointp`, cuya definición omitimos, implementa la noción de intersección vacía de listas. Nótese el uso de `coincide` para expresar la igualdad de dos sustituciones en un conjunto finito de variables:

```
(defthm coincide-in-term
   (implies (and (subsetp (variables flg term) l)
                 (coincide sigma1 sigma2 l))
            (equal (apply-subst flg sigma1 term)
                   (apply-subst flg sigma2 term))))

(defthm substitution-does-not-change-term
  (implies (disjointp (domain sigma) (variables flg term))
           (equal (apply-subst flg sigma term) term)))
```

Como consecuencia del teorema `coincide-in-term`, es posible probar un teorema que muestra que la restricción de una sustitución no afecta al valor que toma sobre términos cuyas variables están en el conjunto en el que se toma la restricción:

```
(defthm subsetp-restriction
```

```
(implies (subsetp (variables flg term) l)
         (equal (apply-subst flg (restriction sigma l) term)
                (apply-subst flg sigma term))))
```

Dado un término (o una lista de términos), existe una cantidad infinita de sustituciones que actúan de la misma manera sobre tal término. A veces, nos interesará tomar un "representante canónico" de todas esas sustituciones. Para ello tomamos la restricción de la sustitución al conjunto de variables del término. Es lo que hace la macro `normal-form--subst`:

```
(defmacro normal-form-subst (flg sigma term)
  `(restriction ,sigma (make-set (variables ,flg ,term))))
```

El siguiente teorema establece que, efectivamente, `(normal-form-subst flg sigma term)` actúa sobre `term` como lo hace `sigma`. Este teorema es consecuencia inmediata del teorema anterior, `subsetp-restriction`:

```
(defthm equal-normal-form-subst-wrt-term
  (equal (apply-subst flg (normal-form-subst flg sigma term) term)
         (apply-subst flg sigma term)))
```

Otro teorema que nos será de utilidad es el que describe el comportamiento de una sustitución sobre variables fuera del dominio.

```
(defthm x-not-in-domain-remains-the-same
  (implies (not (member x (domain sigma)))
           (equal (val x sigma) x)))
```

Por último, presentamos la primera de las *propiedades de clausura* que se muestran en esta memoria. Es la propiedad de clausura correspondiente a la aplicación de sustituciones a términos. El siguiente teorema establece que dicha operación es cerrada en el conjunto de términos de una signatura.

```
(defthm apply-subst-term-s-p-aux
  (implies (and (term-s-p-aux flg term)
                (substitution-s-p sigma))
           (term-s-p-aux flg (apply-subst flg sigma term))))
```

3.1.4 Representación de sistemas de ecuaciones en ACL2

Usaremos pares punteados para representar ecuaciones. Por ejemplo, la ecuación $g(x, y) \approx h(y, x)$ se representa mediante el par punteado `((g x y) . (h y x))`. Los sistemas de ecuaciones se representarán mediante listas de ecuaciones. De esta manera, los sistemas de ecuaciones tienen una representación análoga a las sustituciones. Formalicemos estas ideas.

Ecuaciones y sistemas de ecuaciones

Representando las ecuaciones como pares punteados, las funciones `car` y `cdr` sirven para acceder a los lados izquierdo y derecho de una ecuación. Definimos las macros `lhs` y `rhs` para expresar esta idea:

```
(defmacro lhs (equ) `(car ,equ))

(defmacro rhs (equ) `(cdr ,equ))
```

La función `system-s-p` reconoce los objetos que representan a un sistema de ecuaciones en una signatura:

```
(defun system-s-p (S)
  (if (atom S)
      (equal S nil)
    (and (consp (car S))
         (term-s-p (caar S)) (term-s-p (cdar S))
         (system-s-p (cdr S)))))
```

Análogas consideraciones que para el caso de los términos y las sustituciones, podemos hacer respecto de aquellos objetos que no representan sistemas de ecuaciones en ninguna signatura. Las funciones cuyo argumento de entrada es un sistema de ecuaciones se extienden de manera "consistente" para aquellos objetos que no representan sistemas de ecuaciones en el sentido dado por la función `system-s-p`.

Por ejemplo, lo que sigue son las definiciones de las funciones `system-var` (que obtiene la lista de variables que aparecen en un sistema de ecuaciones) y `length-system` (que obtiene el número total de símbolos en los términos que aparecen en un sistema de ecuaciones):

```
(defun system-var (S)
  (if (endp S)
      nil
    (append (variables t (caar S))
            (append (variables t (cdar S)) (system-var (cdr S))))))

(defun length-system (S)
  (if (endp S)
      0
    (+ (length-term t (caar S)) (length-term t (cdar S))
       (length-system (cdr S)))))
```

Sistemas de ecuaciones y sustituciones

Podemos definir el concepto de sistema–sustitución, definición que formaliza la dada en 3.20:

```
(defun system-substitution (S)
  (if (endp S)
```

```
    t
  (and (consp (car S))
       (variable-p (caar S))
       (not (member (caar S) (domain (cdr S))))
       (system-substitution (cdr S)))))
```

Nótese que en nuestra formalización, un sistema–sustitución S y la correspondiente sustitución asociada $\vec{S}$ están representados por el mismo objeto. Sacaremos partido de este hecho más adelante.

Emparejando argumentos: el uso de multivalores

En los algoritmos de subsunción y unificación que veremos a continuación, será a veces necesario construir sistemas de ecuaciones que emparejen dos a dos los argumentos de dos términos. En concreto, dados dos términos $t = f(t_1, \ldots, t_n)$ y $s = f(s_1, \ldots, s_m)$ necesitamos una función que construya el sistema $\{t_1 \approx s_1, \ldots, t_n \approx s_n\}$, si $n = m$ y que devuelva fallo si $n \neq m$. Para ello, usaremos la siguiente función:

```
(defun pair-args (l1 l2)
  (cond ((endp l1) (if (equal l1 l2) (mv nil t) (mv nil nil)))
        ((endp l2) (mv nil nil))
        (t (mv-let (pair-rest bool)
                   (pair-args (cdr l1) (cdr l2))
                   (if bool
                       (mv (cons (cons (car l1) (car l2)) pair-rest) t)
                     (mv nil nil))))))
```

La función `pair-args` recibe dos listas y devuelve dos valores agrupados en un multivalor. Si las listas son de la misma longitud, como primer valor devuelve la lista de pares punteados resultante de emparejar los elementos que ocupan la misma posición en las respectivas listas y como segundo el valor booleano `t`. Si las listas no son de la misma longitud, los dos valores devueltos son `nil`. Nótese que no necesitamos recorrer dos veces la lista, una para calcular su longitud y otra para realizar los emparejamientos. El uso de multivalores nos permite comprobar la igualdad de longitud de las dos listas (segundo valor) y en caso afirmativo, el primero de los valores contiene el emparejamiento resultante.

El uso de multivalores es necesario en este caso, ya que el símbolo `nil` puede representar tanto un sistema sin ecuaciones, como un indicador de fallo. Por tanto, usamos un multivalor que contenga un valor que indique el éxito o fracaso de la función y, en caso de éxito, el otro valor contenga el resultado calculado. Además de usar multivalores por razones de eficiencia (véase capítulo B) éste es un uso típico de los multivalores en nuestra formalización y solventa la posible ambigüedad que a veces puede surgir en el hecho de que la constante `nil` se use tanto para representar un objeto de la teoría como para representar fallo. Desde el punto de vista lógico, los multivalores se consideran exactamente igual que las listas.

3.2 Los términos como árboles

La estructura de un término puede ser fácilmente visualizada representándolo como un árbol etiquetado, donde las etiquetas de los nodos son los símbolos de función y variable,

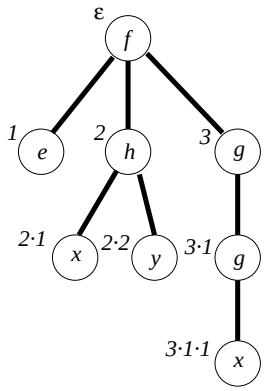

Figura 3.3: Árbol correspondiente al término $t = f(e, h(x, y), g(g(x)))$

y los hijos de cada nodo son los árboles que representan a sus argumentos. En la figura 3.3 se representa el término $t = f(e, h(x, y), g(g(x)))$ en forma de árbol. Es posible designar a los nodos del árbol correspondiente a un término mediante cadenas de números naturales. En la figura 3.3, hemos indicado cada nodo con la posición correspondiente. Por ejemplo, la cadena vacía ϵ indica la posición del símbolo f en el término t. El subtérmino de t indicado por la posición $3 \cdot 1$ es el primer argumento del tercer argumento de t (es decir, $g(x)$). En lo que sigue, definiremos estos conceptos intuitivos y veremos cómo se formalizan y se prueban en ACL2 una serie de propiedades relativas a la estructura de árbol de los términos, que serán de utilidad más adelante. Las definiciones y teoremas que se presentan en esta sección se encuentran en el libro `terms.lisp`.

3.2.1 Preliminares

Definición 3.21 *El conjunto de* **posiciones** *de un término $t \in T(\Sigma, X)$, notado $\mathcal{P}(t)$, es un conjunto de cadenas de números naturales definido inductivamente como sigue:*

- *Si $t = x \in X$, entonces $\mathcal{P}(t) = \{\epsilon\}$.*
- *Si $t = f(t_1, \ldots, t_n)$, entonces*

$$\mathcal{P}(t) = \{\epsilon\} \cup \bigcup_{i=1}^{n} \{i \cdot p \quad | \quad p \in \mathcal{P}(t_i)\}$$

Nótese las similitud de esta definición con las definiciones A.19 y A.20 del apéndice A. De hecho, un término t se puede ver como un árbol $(\Sigma \cup X)$-etiquetado, cuyo dominio es $\mathcal{P}(t)$. La posición ϵ se denomina **posición raíz** del término y el símbolo correspondiente al nodo de la posición raíz se denomina **símbolo raíz**.

Las siguientes definiciones son análogas a las dadas en A.18 para cadenas en general:

Definición 3.22 *El* **orden prefijo** *entre posiciones de un término viene definido de la siguiente manera:*

$$p \leq q \text{ si y sólo si existe } p' \text{ tal que } p \cdot p' = q$$

En ese caso, diremos que p' es la posición **diferencia** *entre p y q. Dadas dos posiciones p y q de un término, diremos que p está* **sobre** *q (y que q está* **debajo** *de p) si $p \leq q$. Asimismo, diremos que son* **disjuntas** *(y lo notaremos por $p|q$) si son incomparables respecto del orden prefijo.*

Definición 3.23 *Dados $s, t \in T(\Sigma, X)$ y $p \in \mathcal{P}(t)$, la* **ocurrencia** *de t en la posición p, notado t/p, y el* **reemplazamiento** *de dicha ocurrencia de t por s, notado $t[p \leftarrow s]$, se definen de la siguiente manera, por inducción en la longitud de p:*

- *Si $p = \epsilon$, entonces $t/p = t$ y $t[p \leftarrow s] = s$.*
- *Si $p = i \cdot q$ (y por tanto t es de la forma $f(t_1, \ldots, t_n)$), entonces $t/p = t_i/q$ y $t[p \leftarrow s] = f(t_1, \ldots, t_i[q \leftarrow s], \ldots, t_n)$.*

Diremos que $t' \in T(\Sigma, X)$ es **subtérmino** *de t, si existe $p \in \mathcal{P}(t)$ tal que $t/p = t'$. Diremos que $p \in \mathcal{P}(t)$ es una* **posición variable** *de t si $t/p \in X$.*

Ejemplo 3.24 Si $t = f(x, g(x, y))$, entonces $\mathcal{P}(t) = \{\epsilon, 1, 2, 2 \cdot 1, 2 \cdot 2\}$, $t/2 \cdot 1 = x$ (por lo que $2 \cdot 1$ es una posición variable de t) y $t[2 \leftarrow h(y)] = f(x, h(y))$. El término $g(x, y)$ es un subtérmino de t.

El siguiente teorema recoge algunas propiedades relativas a la estructura de árbol de los términos, que serán de utilidad más adelante.

Teorema 3.25 *Sean $s, t, u \in T(\Sigma, X)$ y $p \in \mathcal{P}(t)$.*

1. *Entonces $q \in \mathcal{P}(t/p)$ si y sólo si $p \cdot q \in \mathcal{P}(t)$ y en ese caso:*
   a) $t/p \cdot q = (t/p)/q$
   b) $t[p \cdot q \leftarrow s] = t[p \leftarrow (t/p)[q \leftarrow s]]$
2. *Si σ es una sustitución en $T(\Sigma, X)$, entonces:*
   a) $p \in \mathcal{P}(\sigma(t))$
   b) $\sigma(t)/p = \sigma(t/p)$
   c) $\sigma(t)[p \leftarrow \sigma(s)] = \sigma(t[p \leftarrow s])$
3. *Entonces $p \cdot q \in \mathcal{P}(t[p \leftarrow s])$ si y sólo si $q \in \mathcal{P}(s)$ y en ese caso:*
   a) $t[p \leftarrow s]/p \cdot q = s/q$
   b) $t[p \leftarrow s][p \cdot q \leftarrow u] = t[p \leftarrow s[q \leftarrow u]]$
4. *Si $q \in \mathcal{P}(t)$ y $p|q$, entonces:*
   a) $p \in t[q \leftarrow s]$
   b) $t[q \leftarrow s]/p = t/p$
   c) $t[q \leftarrow s][p \leftarrow u] = t[p \leftarrow u][q \leftarrow s]$

Demostración:
Estas propiedades se demuestran de manera bastante rutinaria y sencilla por inducción en la longitud de la posición $p \in \mathcal{P}(t)$. Como ejemplo, veamos la demostración de que $\sigma(t/p) = \sigma(t)/p$.

Si $p = \epsilon$, el resultado es trivial. Si $p = i \cdot q$, entonces necesariamente $t = f(t_1, \ldots, t_n)$. Podemos suponer, como hipótesis de inducción para t_i y $q \in \mathcal{P}(t_i)$, que $\sigma(t_i/q) = \sigma(t_i)/q$. Entonces $\sigma(t/p) = \sigma(t_i/q) = \sigma(t_i)/q = \sigma(t)/i \cdot q = \sigma(t)/p$. □

3.2.2 La estructura de árbol de los términos

Posición, ocurrencia y reemplazamiento: definiciones

Podemos definir en ACL2, de manera análoga a la definición dada en 3.21, lo que significa ser una posición de un término. La función `position-p` implementa tal concepto.

```
(defun position-p (pos term)
 (cond ((atom pos) (equal pos nil))
       ((variable-p term) nil)
       (t (and (integerp (car pos))
               (< 0 (car pos)) (<= (car pos) (len (cdr term)))
               (position-p (cdr pos) (nth (- (car pos) 1) (cdr term)))))))
```

También de manera análoga a las definiciones dadas en 3.23, podemos definir la ocurrencia en una posición dada y el reemplazamiento en una posición dada. Recuérdese que `(replace-list l i x)` devuelve la lista resultante de reemplazar en `l` el `i`-ésimo elemento por `x` (página 35).

```
(defun occurrence (term pos)
  (if (endp pos)
      term
    (occurrence (nth (- (car pos) 1) (cdr term)) (cdr pos))))

(defun replace-term (term1 pos term2)
  (if (endp pos)
      term2
    (cons (car term1)
          (replace-list (cdr term1)
                        (- (car pos) 1)
                        (replace-term (nth (- (car pos) 1) (cdr term1))
                                      (cdr pos)
                                      term2)))))
```

Propiedades

Los siguientes teoremas formalizan las propiedades listadas en el teorema 3.25:

```
(defthm position-p-append                          ;;; 1. a)
  (implies (position-p p1 term)
```

```
          (iff (position-p (append p1 p2) term)
               (position-p p2 (occurrence term p1)))))

(defthm occurrence-append                          ;;; 1. b)
  (implies (and (position-p p1 term)
                (position-p p2 (occurrence term p1)))
           (equal (occurrence term (append p1 p2))
                  (occurrence (occurrence term p1) p2))))

(defthm replace-term-append                        ;;; 1. c)
  (implies (position-p (append pos1 q) term)
           (equal (replace-term term (append pos1 q) x)
                  (replace-term term pos1
                                (replace-term (occurrence term pos1) q x)))))

(defthm position-p-instance                        ;;; 2. a)
  (implies (position-p pos term)
           (position-p pos (instance term sigma))))

(defthm occurrence-instance                        ;;; 2. b)
  (implies (position-p pos term)
           (equal (instance (occurrence term pos) sigma)
                  (occurrence (instance term sigma) pos))))

(defthm replace-term-instance                      ;;; 2. c)
  (implies (position-p pos term)
           (equal (instance (replace-term term pos t1) sigma)
                  (replace-term (instance term sigma) pos (instance t1 sigma)))))

(defthm position-p-prefix                          ;;; 3. a)
  (implies (position-p pos1 term1)
           (iff (position-p (append pos1 pos2) (replace-term term1 pos1 term2))
                (position-p pos2 term2))))

(defthm occurrence-prefix                          ;;; 3. b)
  (implies (and (position-p pos1 term1)
                (position-p pos2 term2))
           (equal (occurrence (replace-term term1 pos1 term2) (append pos1 pos2))
                  (occurrence term2 pos2))))

(defthm replace-term-prefix                        ;;; 3. c)
   (implies (and (position-p pos1 term1)
                 (position-p pos2 term2))
            (equal (replace-term (replace-term term1 pos1 term2)
                                 (append pos1 pos2)
                                 term3)
                   (replace-term term1 pos1 (replace-term term2 pos2 term3)))))

(defthm position-p-disjoint-positions              ;;; 4. a)
  (implies (and (position-p pos1 term)
                (position-p pos2 term)
                (disjoint-positions pos1 pos2))
           (position-p pos1 (replace-term term pos2 x))))
```

```
(defthm occurrence-disjoint-positions              ;;; 4. b)
  (implies (and (position-p pos1 term)
                (position-p pos2 term)
                (disjoint-positions pos1 pos2))
           (equal (occurrence (replace-term term pos1 x) pos2)
                  (occurrence term pos2))))

(defthm replace-term-disjoint-positions            ;;; 4. c)
  (implies (and (position-p pos1 term)
                (position-p pos2 term)
                (disjoint-positions pos1 pos2))
           (equal (replace-term (replace-term term pos1 x) pos2 y)
                  (replace-term (replace-term term pos2 y) pos1 x))))
```

El lector puede consultar algunos detalles relativos a la automatización de estas demostraciones en la subsección C.1.1 del apéndice C.

3.3 Equiparación y subsunción

Las sustituciones permiten definir una importante relación en el conjunto de los términos de primer orden. Entendiendo un término como un "patrón" que comparten todos aquellos términos que se obtienen de sustituir las variables de aquél por otros términos, podemos definir la relación "ser más general que", usualmente llamada relación de *subsunción*. En esta sección veremos cómo definir la relación de subsunción en ACL2. Esta definición la haremos de manera constructiva, mediante la verificación de un algoritmo que, dados dos términos, encuentra (siempre que exista) una sustitución que aplicada al primero de los términos obtiene el segundo. Este algoritmo se denomina de *equiparación*[8]. Como veremos, el estudio de tal algoritmo se simplifica si lo generalizamos a la equiparación de sistemas de ecuaciones, entendiendo como tal la búsqueda de una sustitución que aplicada a los lados izquierdos de las ecuaciones de un sistema obtiene los lados derechos respectivos. Los eventos ACL2 que se presentan en esta sección se encuentran en los libros `subsumption.lisp` y `matching.lisp`.

3.3.1 Preliminares

Definición 3.26 *Dados $s, t \in T(\Sigma, X)$ diremos que s* **subsume** *a t, o que t es una* **instancia** *de s, o que s es* **más general** *que t, o que t es* **más particular** *s y lo notaremos $s \preccurlyeq t$, si existe una substitución en $T(\Sigma, X)$, σ, tal que $\sigma(s) = t$. En ese caso decimos que σ es una sustitución* **testigo de la subsunción** *entre s y t.*

Ejemplo 3.27 Si $s_1 = i(x) * y$, $s_2 = x * i(y)$ y $s_3 = i(u * v) * i(y * z)$, entonces $s_1 \preccurlyeq s_3$, justificado por la sustitución $\{x \mapsto u * v, y \mapsto i(y * z)\}$. También se tiene que $s_2 \preccurlyeq s_3$ y que s_1 y s_2 son incomparables bajo la relación de subsunción.

Teorema 3.28 *La relación de subsunción en $T(\Sigma, X)$ es un preorden.*

[8] Traducción de la palabra inglesa *matching*.

Demostración:
Es reflexiva ya que si ϵ es la sustitución identidad (con dominio vacío), entonces $\epsilon(t) = t$ y por tanto $t \preccurlyeq t$. Para ver la transitividad, supongamos que $s \preccurlyeq t$ y $t \preccurlyeq u$. Por tanto existen σ y τ tal que $\sigma(s) = t$ y $\tau(t) = u$, lo que implica que $\tau\sigma(s) = u$ y por tanto $s \preccurlyeq u$. □

Definición 3.29 *Dado un sistema de ecuaciones S (en $T(\Sigma, X)$), una sustitución σ (en $T(\Sigma, X)$) se dirá un* **equiparador** *de S si para cualquier $s \approx t \in S$, se verifica que $\sigma(s) = t$. Un sistema de ecuaciones se dirá* **equiparable** *si tiene un equiparador.*

En particular, si $S = \{s \approx t\}$ y σ es un equiparador de S, entonces decimos que σ es un equiparador de s y t. Es claro que $s \preccurlyeq t$ si y sólo si existe un equiparador de s y t.

Existen determinados sistemas para los que la obtención de un equiparador es trivial, como refleja el siguiente lema:

Lema 3.30 *Sea S un sistema–sustitución. Entonces S es equiparable y $\vec{S}$ es un equiparador de S.*

A continuación, definiremos un *algoritmo de equiparación*. Este algoritmo, dado un sistema de ecuaciones S, devolverá un equiparador de S, si S es equiparable y fallo en caso contrario. El algoritmo, inspirado en la definición del algoritmo de unificación que daremos en 4.4.1, vendrá especificado mediante un conjunto de reglas de transformación que definen una relación $\Rightarrow_s$ entre pares de sistemas de ecuaciones. Tales reglas de transformación se presentan en la figura 3.4.

Asocia:	$\{x \approx t\} \cup R; T$	$\Rightarrow_s R; \{x \approx t\} \cup T$ si $x \in X$ y $x \notin \mathcal{D}(\vec{T})$
Borra:	$\{x \approx t\} \cup R; T$	$\Rightarrow_s R; T$ si $x \in \mathcal{D}(\vec{T})$ y $\vec{T}(x) = t$
Fallo1:	$\{x \approx t\} \cup R; T$	$\Rightarrow_s \bot$ si $x \in \mathcal{D}(\vec{T})$ y $\vec{T}(x) \neq t$
Fallo2:	$\{f(s_1, \ldots, s_n) \approx x\} \cup R; T$	$\Rightarrow_s \bot$, si $x \in X$
Descomp:	$\{f(s_1, \ldots, s_n) \approx f(t_1, \ldots, t_n)\} \cup R; T$	$\Rightarrow_s \{s_1 \approx t_1, \ldots, s_n \approx t_n\} \cup R; T$
Conflicto:	$\{f(s_1, \ldots, s_n) \approx g(t_1, \ldots, t_m)\} \cup R; T$	$\Rightarrow_s \bot$, si $n \neq m$ ó $f \neq g$

Figura 3.4: Reglas de transformación para la equiparación

Las transformaciones actúan sobre pares de sistemas de ecuaciones. Intuitivamente, el primero de estos sistemas contiene las ecuaciones pendientes de equiparar y el segundo contiene las asociaciones realizadas hasta el momento (o, lo que es lo mismo, el equiparador parcialmente calculado). Como caso especial, el símbolo $\bot$ indica fallo en la equiparación. Como veremos, este segundo sistema será siempre un sistema–sustitución y por tanto tiene sentido hablar de $\vec{T}$.

Las transformaciones están diseñadas para que, comenzando en el par de sistemas $S; \emptyset$ (donde S es el sistema que se quiere equiparar), y aplicando repetidas veces los pasos de transformación, obtengamos finalmente un par de sistemas $\emptyset; T$, siendo T un sistema sustitución (y por tanto equiparable), o bien $\bot$ (y por tanto no equiparable). Si las

transformaciones preservan el conjunto de equiparadores del par de sistemas, entonces $\vec{T}$ será un equiparador de S. Siguiendo esta idea intuitiva, podemos diseñar y verificar un algoritmo de equiparación. Antes veamos algunas propiedades de la relación $\Rightarrow_s$.

Lema 3.31 *Sean S, S', T y T' sistemas de ecuaciones en $T(\Sigma, X)$, tales que $S;T \Rightarrow_s S';T'$. Entonces:*

a) σ es equiparador de $S \cup T$ si y sólo si es equiparador de $S' \cup T'$.

b) Si T es un sistema–sustitución, entonces T' también lo es.

Demostración:
Si se han aplicado las reglas **Asocia**, **Borra** o **Descomp**, entonces las propiedades son fáciles de comprobar. Las reglas **Fallo1**, **Fallo2** y **Conflicto** no se aplican en este caso, ya que no obtienen un par de sistemas, sino $\perp$. □

Lema 3.32 *Sean S y T sistemas de ecuaciones en $T(\Sigma, X)$, tales que $S;T \Rightarrow_s \perp$. Entonces $S \cup T$ no es equiparable.*

Demostración:
Es fácil ver que los sistemas a los que se les pueden aplicar las reglas **Fallo1**, **Fallo2** y **Conflicto** no son equiparables. □

El conjunto de reglas de $\Rightarrow_s$ está diseñado de manera que existe una regla aplicable para cualquier tipo de ecuación seleccionada. Esto hace que se verifique la siguiente propiedad:

Lema 3.33 *Sean S y T sistemas de ecuaciones en $T(\Sigma, X)$, tales que el par $S;T$ está en forma normal respecto de $\Rightarrow_s$. Entonces $S = \emptyset$.*

A continuación definimos un algoritmo de equiparación de sistemas de ecuaciones, basado en el conjunto de reglas de transformación $\Rightarrow_s$:

equipara(S) =
  $T := \emptyset$
  mientras $S \neq \emptyset$ **hacer**
    si $S;T \Rightarrow_s \perp$, **parar** *y* **devolver** *FALLO*
    si no, *sea* $S';T'$ *tal que* $S;T \Rightarrow_s S';T'$ *y* **hacer** $S := S'$ *y* $T := T'$
  fin-mientras
  devolver $\vec{T}$

Nótese que el algoritmo es *no determinista*, ya que no se explicita qué regla se aplica en cada paso de transformación. Es decir, si $S;T \Rightarrow_s S';T'$ y $S;T \Rightarrow_s S'';T''$, usando distintas reglas de transformación, el algoritmo puede escoger arbitrariamente cualquiera de ellas.

Los dos teoremas siguientes muestran las propiedades de terminación, corrección y completitud del algoritmo de equiparación diseñado:

Teorema 3.34 *equipara*(S) *termina para cualquier sistema de ecuaciones S.*

Demostración:
Supongamos que S, S', T y T' son sistemas de ecuaciones en $T(\Sigma, X)$, tales que $S;T \Rightarrow_s S';T'$. Entonces es fácil ver que para cualquier regla aplicada se tiene que S' tiene menos símbolos que S. Por este motivo y por el lema 3.33, concluimos que la condición de parada del bucle en el algoritmo *equipara* siempre se llega a cumplir, a no ser que pare devolviendo $FALLO$. □

Teorema 3.35 *(**Corrección y completitud del algoritmo de equiparación**) Sea S un sistema en $T(\Sigma, X)$. Entonces S es equiparable si y sólo si equipara(S) no devuelve $FALLO$. En tal caso, equipara(S) devuelve un equiparador de S.*

Demostración:
Supongamos que *equipara*(S) no devuelve $FALLO$. Entonces existe una $\Rightarrow_s$-derivación de la forma $S;\emptyset \Rightarrow_s \ldots \Rightarrow_s \emptyset;T$ y el algoritmo devuelve $\vec{T}$. Por el lema 3.31, apartado b), se tiene que T es un sistema–sustitución; por tanto T es equiparable y $\vec{T}$ es un equiparador de T (lema 3.30). Por el lema 3.31, apartado a), S es equiparable y $\vec{T}$ un equiparador de S.

Supongamos que S es equiparable. Entonces el algoritmo no devuelve $FALLO$, porque en tal caso existiría una $\Rightarrow_s$-derivación de la forma $S;\emptyset \Rightarrow_s \ldots \Rightarrow_s \bot$. Pero esto está en contradicción con los lemas 3.31, apartado a), y 3.32. □

Como caso particular del algoritmo de equiparación de sistemas de ecuaciones, obtenemos un algoritmo de equiparación entre dos términos, permitiendo comprobar si dos términos están relacionados por la relación de subsunción. El siguiente teorema es consecuencia inmediata de 3.35.

Teorema 3.36 *Sean $t_1, t_2 \in T(\Sigma, X)$. Entonces $t_1 \preccurlyeq t_2$ si y sólo si equipara$(\{t_1 \approx t_2\})$ no devuelve $FALLO$. En ese caso, devuelve una sustitución σ tal que $\sigma(t_1) = t_2$*

3.3.2 Definición de un algoritmo de equiparación

El teorema 3.36 proporciona una definición constructiva de la relación de subsunción. En lo que sigue, definiremos (y verificaremos formalmente) una función en ACL2 que define la relación de subsunción entre términos. Al carecer ACL2 de cuantificación existencial, la definición ha de ser constructiva, inspirada en este teorema. La verificación de que tal función implementa realmente la relación de subsunción consiste en demostrar formalmente el teorema. En consecuencia, necesitamos definir en ACL2 el algoritmo *equipara* y probar formalmente su corrección y completitud (teorema 3.35).

No determinismo: función de selección de ecuaciones

Para formalizar el no determinismo que existe al aplicar las reglas de transformación en el algoritmo de equiparación que definiremos a continuación, nos basaremos en el siguiente hecho: la regla de transformación a aplicar queda determinada completamente por la forma de la ecuación a la que se le aplica. Es decir, el no determinismo reside en la posibilidad de escoger la ecuación a la cual se le aplica la regla.

Usando `encapsulate` definiremos parcialmente una función `a-pair` que seleccione una ecuación de cada conjunto de ecuaciones no vacío:

```
(encapsulate
 ((a-pair (lst) t))
 ...
 (defthm a-pair-selected
   (implies (consp l) (member (a-pair l) l))))
```

Los puntos suspensivos nos sirven para omitir (ya que es irrelevante) la definición local que justifica la introducción de tal propiedad sobre `a-pair`. El definir de esta manera la función de selección hace que el algoritmo de equiparación que definiremos a continuación no se pueda ejecutar. Sin embargo, existe una ventaja desde el punto de vista de la verificación de propiedades, ya que esto supone que estamos formalizando y verificando una familia de algoritmos de equiparación, uno por cada función de selección concreta.

Equiparación de sistemas de ecuaciones

A continuación definimos en ACL2 un algoritmo de equiparación de sistemas de ecuaciones. En primer lugar, definiremos una función `transform-subs-sel` que aplica un paso de transformación a un par de sistemas de ecuaciones, para obtener un par de sistemas transformados o fallo. Representamos los pares de sistemas como pares punteados y el fallo por la constante `nil`.

```
(defun transform-subs-sel (S-match)
  (let* ((S (car S-match)) (match (cdr S-match))
         (ecu (a-pair S))
         (t1 (car ecu)) (t2 (cdr ecu))
         (R (eliminate ecu S)))
    (cond
     ((variable-p t1)
      (let ((bound (assoc t1 match)))
        (if bound
            (if (equal (cdr bound) t2)
                (cons R match)                          ;;; *** BORRA
              nil)                                      ;;; *** FALLO1
          (cons R (cons (cons t1 t2) match)))))         ;;; *** ASOCIA
     ((variable-p t2) nil)                              ;;; *** FALLO2
     ((equal (car t1) (car t2))
      (mv-let (pair-args bool)
              (pair-args (cdr t1) (cdr t2))
              (if bool
                  (cons (append pair-args R) match)     ;;; *** DESCOMP
                nil)))                                  ;;; *** CONFLICTO
     (t nil))))                                         ;;; *** CONFLICTO
```

Esta función recibe un par de sistemas de ecuaciones, `S-match`, selecciona una ecuación del primero del par de sistemas, `(a-pair (car S-match))` y según la forma de la ecuación seleccionada, aplica una de las reglas de $\Rightarrow_s$ descritas en la figura 3.4. Téngase en cuenta el uso de la función `pair-args` (definida en la subsección 3.1.4) para implementar la acción

de las reglas **Descomp** y **Conflicto** y el uso de `assoc` (función Common Lisp) para comprobar la condición "$x \in \mathcal{D}(\vec{T})$".

Este paso de transformación que efectúa la función `transform-subs-sel`, deberá ser aplicado iterativamente hasta que se obtenga un par de sistemas en el que el primero de ellos es el sistema vacío o bien hasta que se obtenga fallo. La función `normal-form-syst` comprueba esta condición de parada (es decir, según el lema 3.33, comprueba si el par de sistemas está en *forma normal* respecto de la reducción $\Rightarrow_s$, o es $\bot$):

```
(defun normal-form-syst (S-T)
  (not (and (consp S-T) (consp (car S-T)))))
```

La función `subs-system-sel` itera la aplicación de las reglas de transformación hasta que se cumple la condición de parada. La admisión de la función `subs-system-sel` necesita la especificación de una medida que justifique su terminación (más adelante comentaremos esta cuestión con más detalle).

```
(defun subs-system-sel (S-match)
  (declare (xargs :measure (length-system (car S-match))))
  (if (normal-form-syst S-match)
      S-match
    (subs-system-sel (transform-subs-sel S-match))))
```

Finalmente, definimos la función `match-sel`, que implementa un algoritmo de equiparación de sistemas de ecuaciones. Para ello, dado un sistema de ecuaciones `S`, simplemente llamamos a la función `subs-system-sel` sobre el par de sistemas `(cons S nil)`. Si ésta llamada termina con éxito, devolvemos una lista cuyo único elemento es el equiparador obtenido. En caso contrario, devolvemos `nil`. Nótese que es necesario el uso `list` para distinguir el objeto `nil` como representación de la sustitución identidad, de `nil` como indicación de fallo[9].

```
(defun match-sel (S)
  (let ((subs-system-sel (subs-system-sel (cons S nil))))
    (if subs-system-sel (list (cdr subs-system-sel)) nil)))
```

3.3.3 Los teoremas principales

En esta subsección presentamos las propiedades principales del algoritmo de equiparación definido. Antes de eso, presentamos la definición formal del concepto de equiparador, que implementa la función `matcher`:

```
(defun matcher (sigma S)
  (if (endp S)
      t
    (and (equal (apply-subst t sigma (caar S))
                (cdar S))
         (matcher sigma (cdr S)))))
```

[9] Sin embargo, la función `subs-system-sel` no necesita de este artificio, ya que cuando acaba con éxito devuelve un par de sistemas y por tanto sólo devuelve `nil` cuando falla (diremos que `(match-sel S)` falla cuando el valor devuelto es `nil`).

Los siguientes teoremas expresan la corrección y completitud del algoritmo de equiparación de sistemas de ecuaciones definido por `match-sel` y establecen formalmente el teorema 3.35. La corrección del algoritmo significa que cuando no devuelve fallo, obtiene un equiparador del sistema de ecuaciones que recibe como entrada (y por tanto, se trata de un sistema equiparable). La completitud, establece que cuando el algoritmo recibe un sistema equiparable como entrada, no devuelve fallo.

```
(defthm match-sel-soundness
  (implies (match-sel S)
           (matcher (first (match-sel S)) S)))

(defthm match-sel-completeness
  (implies (matcher sigma S)
           (match-sel S)))
```

Además, el algoritmo de equiparación verifica la correspondiente propiedad de clausura: si recibe como entrada un sistema de ecuaciones en una signatura, el equiparador devuelto es una sustitución en la misma signatura. Es lo que establece el siguiente teorema:

```
(defthm match-sel-substitution-s-p
  (implies (system-s-p S)
           (substitution-s-p (first (match-sel S)))))
```

3.3.4 Descripción de la demostración

Describimos en esta subsección los lemas principales que guían al demostrador a obtener una prueba de los anteriores resultados en la lógica de ACL2. La demostración se basa en la prueba a mano presentada en la subsección 3.3.1.

Terminación de la relación $\Rightarrow_s$

Como ya se comentó al dar la definición de la función `subs-system-sel`, para su admisión es necesario probar previamente que, dado un par de sistemas `S-match`, la computación de `(subs-system-sel S-match)` siempre termina. Para ello, como sugiere la demostración del teorema 3.34, demostramos que la medida `(length-system (car S-match))` decrece en cada llamada recursiva. Es decir, que la aplicación de un paso de transformación sobre un par de sistemas que no esté en forma normal hace que el número de símbolos del primero de los sistemas del par, decrezca estrictamente respecto de `e0-ord-<`. Es lo que establece el siguiente teorema:

```
(defthm transform-subs-sel-decreases-length-of-first-system
  (implies (not (normal-form-syst S-match))
           (e0-ord-< (length-system (car (transform-subs-sel s-match)))
                     (length-system (car s-match)))))
```

Invariantes en las transformaciones

Análogamente al resultado de los lemas 3.31 (apartado a)) y 3.32, se demuestra que el conjunto de equiparadores del par de sistemas[10] es un invariante en las transformaciones que se llevan a cabo con $\Rightarrow_s$. Los tres teoremas siguientes establecen tal hecho:

```
(defun union-systems (S-T) (append (car S-T) (cdr S-T)))

(defthm transform-subs-sel-preserves-matchers-1
  (implies (and (not (normal-form-syst S-match))
                (matcher sigma (union-systems S-match)))
           (matcher sigma (union-systems (transform-subs-sel S-match)))))

(defthm transform-subs-sel-preserves-matchers-2
  (implies (and (not (normal-form-syst S-match))
                (transform-subs-sel S-match)
                (matcher sigma
                         (union-systems (transform-subs-sel S-match))))
           (matcher sigma (union-systems S-match))))

(defthm transform-subs-sel-fail
  (implies (and (not (normal-form-syst S-match))
                (not (transform-subs-sel S-match)))
           (not (matcher sigma (union-systems S-match)))))
```

Otro invariante en las transformaciones es la propiedad de que el segundo sistema de ecuaciones sea un sistema–sustitución (lema 3.31, apartado b)):

```
(defthm transform-subs-sel-preserves-system-substitution
  (implies (and (not (normal-form-syst S-match))
                (system-substitution (cdr S-match)))
           (system-substitution (cdr (transform-subs-sel S-match)))))
```

El último invariante en las transformaciones, que servirá para probar la propiedad de clausura del algoritmo de equiparación, es que el segundo sistema verifica la propiedad `substitution-s-p`:

```
(defthm transform-subs-sel-sel-preserves-substitution-s-p
  (implies (and (not (normal-form-syst S-match))
                (system-s-p (first S-match))
                (substitution-s-p (cdr S-match)))
           (substitution-s-p (cdr (transform-subs-sel S-match)))))
```

Es evidente que el hecho de que estas propiedades constituyan invariantes en un paso de transformación, hace que también lo sean cuando las transformaciones se aplican de manera iterativa (que es lo que hace la función `subs-system-sel`).

[10] Considerando al sistema $\perp$ como un par de sistemas sin equiparador.

```
(defthm subs-system-sel-preserves-matchers-1
  (implies (matcher sigma (union-systems S-match))
           (matcher sigma (union-systems (subs-system-sel S-match))))))

(defthm subs-system-sel-preserves-matchers-2
  (implies (and (subs-system-sel S-match)
                (matcher sigma (union-systems (subs-system-sel S-match))))
           (matcher sigma (union-systems S-match))))

(defthm subs-system-sel-fail
  (implies (and (not (subs-system-sel S-match))
                (consp S-match))
           (not (matcher sigma (union-systems S-match)))))

(defthm subs-system-sel-preserves-system-substitution
  (implies (system-substitution (cdr S-match))
           (system-substitution (cdr (subs-system-sel S-match)))))

(defthm subs-system-sel-preserves-substitution-s-p
  (implies (and (system-s-p (car S-match))
                (substitution-s-p (cdr S-match)))
           (substitution-s-p (cdr (subs-system-sel S-match)))))
```

Los teoremas anteriores establecen que las siguientes propiedades se tienen después de iterar exhaustivamente las reglas de transformación.

- El conjunto de equiparadores del sistema obtenido después de aplicar las iteraciones es el mismo que el del par de sistemas de partida (teoremas `subs-system--sel-preserves-matchers-1`, `subs-system-sel-preserves-matchers-2` y `subs--system-sel-fail`).
- Si inicialmente el segundo de los sistemas era sistema–sustitución, el que se obtiene finalmente también lo es (teorema `subs-system-sel-preserves-system-substitution`).
- Si inicialmente el primero de los sistemas era un sistema en una signatura dada y el segundo una sustitución en tal signatura, finalmente se obtiene una sustitución en dicha signatura (teorema `subs-system-sel-preserves-substitution-s-p`).

Concluyendo los teoremas principales

Los teoremas anteriores sobre `subs-system-sel` permiten deducir fácilmente las propiedades (presentadas anteriormente) que verifica el algoritmo de equiparación `match-sel`, como explicamos a continuación.

El teorema `match-sel-soundness` se tiene por el siguiente razonamiento. Supongamos que `(match-sel S)` no devuelve fallo. Entonces:

- Como caso particular del teorema `subs-system-sel-preserves-matchers-2` (para `S-match` igual a `(cons S nil)`), todo equiparador del sistema que se obtiene después de iterar las transformaciones, lo será del sistema `S`.

- Según el teorema `subs-system-sel-preserves-system-substitution`, el sistema que finalmente se obtiene es un sistema–sustitución.

- Por tanto, basta con probar un análogo al lema 3.30, que establece que todo sistema–sustitución es equiparador de sí mismo, como hace el siguiente teorema (recuérdese que si `T` representa a un sistema–sustitución T, entonces `T` también representa a la sustitución $\vec{T}$):

```
(defthm system-substitutions-main-property
  (implies (system-substitution T) (matcher T T)))))
```

El teorema `match-sel-completeness` es un caso particular del teorema `subs-system--sel-fail`, instanciando `S-match` por `(cons S nil)`. Instanciando de la misma manera, el teorema `match-sel-substitution-s-p` es un caso particular del teorema `subs--system-sel-substitution-s-p`.

En la sección C.1.2 comentamos algunos detalles adicionales sobre la demostración automática de estos resultados.

3.3.5 Un algoritmo de equiparación ejecutable

El algoritmo de equiparación de sistemas definido por la función `match-sel` no puede ser ejecutado, ya que la función de selección `a-pair` que usa para escoger la sucesión de reglas de transformación que se van a aplicar está parcialmente definida mediante un encapsulado. Vamos a definir ahora un algoritmo de equiparación ejecutable mediante la definición de una función de selección completamente especificada. Las propiedades de este algoritmo ejecutable se obtendrán de manera inmediata mediante instanciación funcional de las propiedades del algoritmo genérico.

La definición del algoritmo

La función de selección que vamos a definir busca una ecuación que implique de manera inmediata que el sistema no es equiparable: ecuaciones de la forma $f(t_1, \ldots, t_n) = x$ (con $x \in X$) o de la forma $f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m)$ (con $f \neq g$). La razón por la que escogemos esta función de selección es que el algoritmo de equiparación se usará como función auxiliar en procedimientos donde en la mayoría de los casos la equiparación falla (por ejemplo, en el cálculo de formas normales respecto de un sistema de reescritura de términos, capítulo 7). Es, pues, deseable que el fallo en la equiparación se detecte cuanto antes. La función `find-inconsistency` implementa esta función de selección:

```
(defun find-inconsistency (S)
  (if (endp (cdr S))
      (car S)
    (let* ((equ (car S)) (t1 (car equ)) (t2 (cdr equ)))
        (cond ((variable-p t1)  (find-inconsistency (cdr S)))
              ((variable-p t2) equ)
              ((eql (car t1) (car t2)) (find-inconsistency (cdr S)))
              (t equ)))))
```

En la figura 3.5, presentamos la definición de un algoritmo de equiparación de sistemas de ecuaciones `match-mv`, que aplica las reglas de transformación de $\Rightarrow_s$ y usa como función de selección `find-inconsistency`.

```
(defun transform-subs (S match)
  (let* ((ecu (find-inconsistency S))
         (t1 (car ecu)) (t2 (cdr ecu))
         (R (eliminate ecu S)))
    (cond
     ((variable-p t1)
      (let ((bound (assoc t1 match)))
         (if bound
             (if (equal (cdr bound) t2)
                 (mv R match t)
               (mv nil nil nil))
           (mv R (cons (cons t1 t2) match) t))))
     ((variable-p t2) (mv nil nil nil))
     ((eql (car t1) (car t2))
      (mv-let (empareja bool)
              (pair-args (cdr t1) (cdr t2))
              (if bool
                  (mv (append empareja R) match t)
                (mv nil nil nil))))
     (t (mv nil nil nil)))))

(defun subs-system (S match bool)
  (if (or (not bool) (not (consp S)))
      (mv S match bool)
    (mv-let (S1 match1 bool1)
            (transform-subs S match)
            (subs-system S1 match1 bool1))))

(defun match-mv (S)
  (mv-let (S1 sol1 bool1)
          (subs-system S nil t)
          (mv sol1 bool1)))
```

Figura 3.5: Un algoritmo de equiparación ejecutable

El algoritmo `match-mv` es, esencialmente, el algoritmo `match-sel`, en el que la función de selección se especifica completamente, definida por `find-inconsistency`. Existen, sin embargo, ciertas diferencias entre ambos:

- La función `transform-subs` recibe el par de sistemas como dos argumentos separados. Asimismo devuelve un multivalor triple: los dos primeros valores representan al par de sistemas transformados y el tercero de los valores es un indicador de fallo

(`nil`) o éxito (`t`). El uso de multivalores mejora considerablemente la eficiencia en ACL2, respecto a la versión en la que se usa un par punteado que contiene al par de sistemas, ya que `mv` evita el coste de la construcción del par punteado. Recordemos que desde el punto de vista de la lógica, los multivalores se consideran listas.

- Este cambio en `transform-subs` afecta también a la definición de `subs-system` y a la de `match-mv`, que en esta versión también usan multivalores. Así, `match-mv` se define como una función que recibe un sistema de ecuaciones como entrada y devuelve dos valores mediante un multivalor: el segundo de estos valores es un booleano indicando el éxito o fallo de la equiparación y el primero de los valores es, en caso de éxito, el equiparador calculado.

- Nótese el uso del predicado de igualdad `eql` a la hora de comparar símbolos. Esto mejora la eficiencia y está permitido si suponemos que los términos de entrada van a ser términos en una signatura (apéndice B). Desde el punto de vista lógico, es equivalente a usar `equal`.

Lo que sigue son un par de ejemplos mostrando la ejecución del algoritmo `match-mv` para calcular un equiparador de un sistema de ecuaciones[11]:

```
ACL2 !>(match-mv '(((f x) . (f (h y)))
                   ((g z) . (g (a)))
                   ((k x) . (k (h y)))))
(((Z . (A)) (X . (H Y))) T)
ACL2 !>(match-mv '(((f x) . (f (h y)))
                   ((g z) . (g (a)))
                   ((k x) . (k (h (a))))))
(NIL NIL)
```

El lector puede consultar más detalles relativos a la ejecución de funciones ACL2, su eficiencia y la verificación de protecciones en el apéndice B.

Propiedades del algoritmo `match-mv`

Lo que sigue son las propiedades fundamentales del algoritmo de equiparación definido por `match-mv`, que se tienen como consecuencia de las propiedades análogas para la función `match-sel` (vistas anteriormente), y se obtienen aplicando la regla de instanciación funcional. Estos teoremas establecen que `(match-mv S)` no falla si y sólo si `S` es equiparable y en ese caso devuelve un equiparador de `S`. Además, se establece la propiedad de clausura para `match-mv`.

```
(defthm match-mv-soundness
  (implies (second (match-mv S))
           (matcher (first (match-mv S)) S)))

(defthm match-mv-completeness
  (implies (matcher sigma S)
```

[11] Se ha mantenido la notación de par punteado para una mayor legibilidad.

```
          (second (match-mv S))))

(defthm match-mv-substitution-s-p
  (implies (system-s-p S)
           (substitution-s-p (first (match-mv S)))))
```

El uso de instanciación funcional

Como se ha comentado, las propiedades del algoritmo `match-mv` se obtienen por instanciación funcional de las propiedades análogas de `match-mv-sel`. Por ejemplo la corrección del algoritmo, dada por el teorema `match-mv-soundness`, se obtiene mediante la siguiente instancia funcional del teorema `match-mv-sel-soundness`:

```
;  :hints (("Goal" :use ((:functional-instance
;                         match-sel-soundness
;                         (match-sel match-mv-bridge)
;                         (subs-system-sel subs-system-bridge)
;                         (transform-subs-sel transform-subs-bridge)
;                         (a-pair find-inconsistency))))))
```

Nótese que la función genérica `a-pair` queda instanciada por la función de selección particular `find-inconsistency`.

Es interesante destacar que la función `transform-subs-sel` no se instancia directamente por la función `transform-subs`, como podría esperarse. La razón es que ésta última devuelve multivalores y aquélla devuelve pares de sistemas (o `nil`). Por este motivo definimos una función "puente" entre ambas: la función `transform-subs-bridge`, que implementa, a partir de `transform-subs`, la función análoga a `transform-subs-sel`. Un caso análogo es el de `subs-system-sel`, `subs-system` y la función puente `subs-system--bridge`. Ocurre también lo mismo con `match-sel`, `match-mv` y la función puente `match--mv-bridge`:

```
(defun transform-subs-bridge (S-match)
  (mv-let (S1 match1 bool1)
          (transform-subs (car S-match) (cdr S-match))
          (if bool1 (cons S1 match1) nil)))

(defun subs-system-bridge (S-match)
  (if (normal-form-syst S-match)
      S-match
    (mv-let (S1 match1 bool1)
            (subs-system (car S-match) (cdr S-match) t)
            (if bool1 (cons S1 match1) nil))))

(defun match-mv-bridge (S)
  (let ((subs-system-bridge (subs-system-bridge (cons S nil))))
    (if subs-system-bridge (list (cdr subs-system-bridge)) nil)))
```

Además de ilustrar el uso de la instanciación funcional como regla de inferencia, la verificación de `match-mv` es un ejemplo (a pequeña escala) del llamado *razonamiento compuesto*. El mayor esfuerzo de verificación se produce al verificar la función `match-sel`, en la que no se han tenido en cuenta determinados aspectos que mejorarían su eficiencia y que podrían "estorbar" en la demostración de sus propiedades fundamentales. Posteriormente, se define la función `match-mv` que refina en algunos aspectos el código de la anterior (por ejemplo, con la introducción de multivalores, que mejoran la eficiencia) y se comprueban teoremas de equivalencia con la función anterior, permitiéndose así el trasladar las propiedades fundamentales obtenidas a la función mejorada. En la subsección 4.3.4 comentamos más sobre este particular.

3.3.6 El preorden de subsunción entre términos

Terminamos esta sección usando el algoritmo de equiparación `match-mv` previamente definido y verificado, para definir en ACL2 de una manera constructiva la relación de subsunción entre términos y demostrar formalmente sus propiedades fundamentales. Tal definición será ejecutable, ya que `match-mv` lo es.

Definiciones

Como caso particular del algoritmo de equiparación de sistemas de ecuaciones, podemos definir un algoritmo de equiparación entre términos. De esta manera, definimos el algoritmo `subs-mv`, tal que dados dos términos `t1` y `t2`, llama a `match-mv` sobre el sistema `(list (cons t1 t2))`, para obtener dos valores: un indicador de fallo (`nil`) o éxito (`t`), y un equiparador de `t1` y `t2` (en caso de éxito).

```
(defun subs-mv (t1 t2)
  (match-mv (list (cons t1 t2))))
```

Para ver si dos términos verifican la relación de subsunción, bastará con observar el éxito o fallo del algoritmo de equiparación anterior al actuar sobre los mismos. Por tanto, la siguiente función `subs` define en ACL2 la relación de subsunción:

```
(defun subs (t1 t2)
  (mv-let (matching subs)
          (subs-mv t1 t2)
          subs))
```

La relación de subsunción, tal y como se ha dado en 3.26, viene definida por una fórmula existencial, ya que decimos que s subsume a t si *existe* una sustitución tal que aplicada a s obtiene t. Para verificar que la función `subs` define realmente la relación de subsunción debemos dar explícitamente una sustitución con tal propiedad (un *testigo*), ya que la lógica de ACL2 carece de cuantificador existencial. En este caso es fácil: tal sustitución la calcula el algoritmo de equiparación y esta definida por la función `matching`:

```
(defun matching (t1 t2)
  (mv-let (matching subs)
          (subs-mv t1 t2)
          matching))
```

Corrección y completitud

Como consecuencia de las propiedades de corrección y completitud del algoritmo `match-mv`, se tiene de manera inmediata (instanciando `S` por `(list (cons t1 t2))`), que la relación `subs` define la relación de subsunción entre términos, demostrando así formalmente el teorema 3.36. Nótese el papel de la función `matching` como función que suple la falta de cuantificador existencial:

```
(defthm subs-soundness
  (implies (subs t1 t2)
           (equal (instance t1 (matching t1 t2)) t2)))

(defthm subs-completeness
  (implies (equal (instance t1 sigma) t2)
           (subs t1 t2)))
```

Además, la función `matching` verifica la correspondiente propiedad de clausura:

```
(defthm matching-substitution-s-p
  (implies (and (term-s-p t1) (term-s-p t2))
           (substitution-s-p (matching t1 t2))))
```

Es importante destacar que en todo el desarrollo posterior de la teoría que se presenta en esta memoria, estos tres teoremas son *las únicas propiedades* que usaremos sobre la función `subs`[12]. Para ello deshabilitamos las definiciones lógicas y las contrapartidas ejecutables[13] de las funciones `match-mv`, `subs-mv`, `subs` y `matching`:

```
(in-theory
 (disable
  match-mv (match-mv) subs-mv (subs-mv) subs (subs) matching (matching)))
```

De esta manera nos aseguramos de que la teoría que se presenta en esta memoria no usa otras propiedades de la función `subs` distintas de su corrección, completitud y su clausura. Esto hace que sea *independiente* de una u otra definición particular del algoritmo de equiparación. Un primer ejemplo de esto lo constituye la prueba de la propiedades de preorden de la relación de subsunción, que presentamos a continuación.

Reflexividad y transitividad de la relación de subsunción

Los siguientes teoremas establecen la reflexividad y transitividad de la relación de subsunción entre términos, demostrando así que se trata de un preorden:

```
(defthm subsumption-reflexive
   (subs t1 t1))

(defthm subsumption-transitive
   (implies (and (subs t1 t2) (subs t2 t3))
            (subs t1 t3)))
```

[12] Por cuestiones técnicas también asumiremos dos reglas que expresan la relación entre `subs-mv`, y `subs` y `matching`. Consulte el lector el fichero `subsumption.lisp`.

[13] Lo cual no afecta a la posibilidad de ejecutar tales funciones. Véase `:executable-counterpart`.

El teorema de reflexividad se tiene usando el teorema `subs-completeness`, ya que la sustitución identidad (representada por `nil`, por ejemplo) es testigo de la subsunción.

La transitividad también se tiene por el teorema de completitud, usando como testigo la sustitución que resulta de componer las sustituciones `(matching t1 t2)` y `(matching t2 t3)`. Para deducir que aplicando al término `t1` la sustitución `(composition (matching t2 t3) (matching t1 t2))` se obtiene `t3`, hay que recurrir dos veces al teorema `subs--soundness`.

Uso de los teoremas de corrección y completitud

Existe una diferencia entre el uso que el demostrador puede hacer del teorema `subs--completeness` y el uso que puede hacer del teorema `subs-soundness`. En el caso del teorema de corrección, su uso es como regla de reescritura: toda instancia de la expresión `(instance t1 (matching t1 t2))` es sustituida por la correspondiente instancia de `t2`.

Sin embargo, el uso del teorema de completitud como regla de reescritura estaría muy limitado, debido a la variable libre `sigma` en sus hipótesis. Si se usara como regla de reescritura, significaría que el demostrador debería "adivinar" la sustitución que aplicada a un término obtiene otro. Esto sólo sería posible si en el momento de producirse la reescritura, tal condición estuviera explícita entre las asumidas como ciertas, lo que rara vez ocurre. Es por esto que el teorema `subs-completeness` está declarado con `:rule-classes nil`. El uso habitual del teorema de completitud es como sigue:

- Supongamos que queremos probar que dos términos T_1 y T_2 están relacionados mediante la relación de subsunción, `subs`, bajo ciertas condiciones.
- Probamos los lemas necesarios que sirven para demostrar que cierta sustitución δ (posiblemente dependiendo de T_1 y T_2), aplicada a T_1, obtiene T_2, bajo tales condiciones.
- Se tiene entonces inmediatamente `(subs` T_1 T_2`)`, mediante un consejo `:use` que instancia el teorema `subs-completeness`, asignando a `sigma` la sustitución δ y a `t1` y `t2` los términos T_1 y T_2, respectivamente.

La demostración del teorema de transitividad de la subsunción es un ejemplo ilustrativo de tal uso de la regla de completitud:

- Queremos probar que `(subs t1 t3)`, bajo las condiciones `(subs t1 t2)` y `(subs t2 t3)`.
- Tenemos probados los teoremas `composition-of-substitutions-apply` (véase la página 78), que expresa cómo calcular la aplicación de una composición de sustituciones y `subs-soundness`, que expresa como calcular la acción de `(matching t1 t2)` sobre `t1` y de `(matching t2 t3)` sobre `t2`. Estos lemas sirven para probar que la sustitución `(composition (matching t2 t3) (matching t1 t2))` aplicada a `t1` obtiene `t3`.
- Por tanto, tenemos inmediatamente `(subs t1 t3)` sin más que usar el teorema de completitud de la subsunción mediante el siguiente consejo:

```
;   :hints (("Goal" :use (:instance
;                         subs-completeness
;                         (sigma (composition (matching t2 t3)
;                                             (matching t1 t2)))
;                         (t2 t3)))))
```

Esta manera de demostrar subsunción entre dos términos mediante el teorema de completitud se usa repetidas veces en el desarrollo de esta teoría.

La relación de subsunción entre listas de términos

A partir de la función `match-mv` podemos definir también la relación de subsunción entre listas de términos:

```
(defun subs-list-mv (l1 l2)
  (mv-let (pair-lists bool)
          (pair-args l1 l2)
          (if bool (match-mv pair-lists) (mv nil nil))))
```

La función `subs-list-mv` define un algoritmo de equiparación de listas de términos, usando `pair-args` para construir el sistema de ecuaciones que recibe `match-mv`. De manera análoga a como se obtiene `subs` a partir de `subs-mv`, definimos `subs-list` a partir de `subs-list-mv`:

```
(defun subs-list (l1 l2)
  (mv-let (matching subs-list)
          (subs-list-mv l1 l2)
          subs-list))
```

A partir de las propiedades de corrección y completitud de `match-mv`, es muy fácil obtener las propiedades correspondiente para `subs-list` (como se ha hecho en el caso de `subs`). Remitimos al lector a la sección 2 del libro `subsumption.lisp`.

3.4 La relación de subsunción entre sustituciones

Es posible definir una relación de subsunción entre sustituciones, de una manera parecida a la definida entre términos: una sustitución subsume a otra si existe una tercera tal que compuesta con la primera es igual a la segunda. En esta sección definiremos y verificaremos formalmente en ACL2 la relación de subsunción entre sustituciones. Como en el caso de los términos, la subsunción entre sustituciones la vamos a definir de manera constructiva, mediante un algoritmo que, dados dos sustituciones, encuentra (si existe) una sustitución que compuesta con la primera es igual a la segunda. Los eventos ACL2 que se presentan en esta sección se encuentran en el libro `subsumption-subst.lisp`.

3.4.1 Preliminares

Definición 3.37 *Dadas dos sustituciones σ y δ en $T(\Sigma, X)$, diremos que σ* **subsume** *a δ, o que σ es* **menos específica** *(o* **más general***) que δ, y lo notamos por $\sigma \preccurlyeq \delta$, si existe una sustitución γ tal que $\delta = \gamma\sigma$. En tal caso diremos que γ es una sustitución* **testigo de la subsunción** *entre σ y δ.*

El siguiente teorema expresa la conexión existente entre la relación de subsunción entre términos y la relación de subsunción entre sustituciones.

Teorema 3.38 *Sean σ y δ dos sustituciones en $T(\Sigma, X)$.*

a) Si $\sigma \preccurlyeq \delta$, entonces $\sigma(t) \preccurlyeq \delta(t)$, para todo $t \in T(\Sigma, X)$.

b) Si en Σ existe un símbolo de función binario y $\sigma(t) \preccurlyeq \delta(t)$ para todo $t \in T(\Sigma, X)$, entonces $\sigma \preccurlyeq \delta$.

Demostración:

a) Como $\sigma \preccurlyeq \delta$, existe una sustitución γ tal que $\delta = \gamma\sigma$. Por tanto, para cualquier $t \in T(\Sigma, X)$, $\delta(t) = \gamma\sigma(t)$ y por tanto $\sigma(t) \preccurlyeq \delta(t)$.

b) Sea $u \in T(\Sigma, X)$ conteniendo a las variables de V, donde $V = \mathcal{D}(\sigma) \cup \mathcal{D}(\delta) \cup \mathcal{V}(\mathcal{C}(\sigma))$. Existe un término tal porque en Σ al menos hay un símbolo de función binario. Entonces $\sigma(u) \preccurlyeq \delta(u)$, por hipótesis. En consecuencia, existe γ' sustitución tal que $\delta(u) = \gamma'\sigma(u)$. Esto significa que para todo $x \in \mathcal{V}(u)$, $\delta(x) = \gamma'\sigma(x)$, y en particular para todo $x \in V$. Sea $\gamma = \gamma'_{|V}$ y veamos que $\delta = \gamma\sigma$; es decir, $\delta(x) = \gamma\sigma(x)$, para todo $x \in X$. Distinguimos tres casos:

- Si $x \in \mathcal{D}(\sigma)$, entonces $\mathcal{V}(\sigma(x)) \subseteq \mathcal{C}(\sigma) \subseteq V$ y por tanto $\gamma\sigma(x) = \gamma'\sigma(x) = \delta(x)$.
- Si $x \in V \setminus \mathcal{D}(\sigma)$, entonces $\sigma(x) = x \in V$ e igualmente $\gamma\sigma(x) = \gamma'\sigma(x) = \delta(x)$.
- Si $x \notin V$, entonces $\gamma\sigma(x) = \gamma(x) = x = \delta(x)$.

□

La condición de que Σ tenga al menos un símbolo de función binario es necesaria, como muestra el siguiente ejemplo:

Ejemplo 3.39 Sea Σ una signatura tal que $\Sigma_0 = \{0\}$, $\Sigma_1 = \{s\}$ y $\Sigma_n = \emptyset$ para $n \geq 2$. Sea $\sigma = \{x \mapsto s(y)\}$ y $\delta = \{x \mapsto s(s(u)), y \mapsto u\}$. Entonces se comprueba que $\sigma \not\preccurlyeq \delta$ y sin embargo, al tener los términos de esta signatura a lo más una variable, se verifica que $\sigma(t) \preccurlyeq \delta(t)$ para todo $t \in T(\Sigma, X)$.

La demostración del teorema 3.38 nos proporciona la base para definir la relación de subsunción entre sustituciones de una manera constructiva, a partir de un algoritmo de equiparación de sistemas de ecuaciones. La idea es que para comprobar la relación de subsunción entre dos sustituciones nos podemos ceñir a la acción de las sustituciones sobre un conjunto finito de variables.

Teorema 3.40 *Sean σ y δ un par de sustituciones en $T(\Sigma, X)$, $V = \mathcal{D}(\sigma) \cup \mathcal{D}(\delta) \cup \mathcal{V}(\mathcal{C}(\sigma)) = \{v_1, \ldots, v_n\}$ y $S = \{\sigma(v_1) \approx \delta(v_1), \ldots, \sigma(v_n) \approx \delta(v_n)\}$. Entonces $\sigma \preccurlyeq \delta$ si y sólo si S es equiparable.*

Demostración:
En primer lugar, obsérvese que de la demostración del teorema 3.38, apartado b), se deduce que si γ es un equiparador de S cuyo dominio está contenido en V, entonces $\delta = \gamma\sigma$.

Supongamos que S es equiparable y sea γ' es equiparador de S y γ la restricción de γ' a las variables de V. Es evidente que γ es equiparador de S y que su dominio está contenido en V. Por tanto, según la observación anterior, $\delta = \gamma\sigma$ y en consecuencia $\sigma \preccurlyeq \delta$.

Si $\sigma \preccurlyeq \delta$, existe ρ tal que $\rho\sigma = \delta$ y esto significa que ρ es equiparador de S. □

El teorema 3.40 proporciona una definición alternativa, constructiva, de la relación de subsunción entre sustituciones. En lo que sigue, definiremos (y verificaremos formalmente) una función en ACL2 que, basada en este teorema, define esta relación.

3.4.2 Definición de la relación de subsunción entre sustituciones

En primer lugar definimos el conjunto $V = \mathcal{D}(\sigma) \cup \mathcal{D}(\delta) \cup \mathcal{V}(\mathcal{C}(\sigma))$ de variables. La función `domain-var` calcula las variables del dominio de una sustitución. En este caso no podemos usar la función `domain` ya que hemos de eliminar aquellos elementos "extraños" del dominio que no sean variables (ya que la sustitución podría ser impropia). La función `co-domain-var` calcula los elementos del codominio de una sustitución. Finalmente, la función `important-variables` calcula V:

```
(defun domain-var (sigma)
  (if (endp sigma)
      nil
    (if (variable-p (caar sigma))
        (cons (caar sigma) (domain-var (cdr sigma)))
      (domain-var (cdr sigma)))))

(defun co-domain-var (sigma)
  (if (endp sigma)
      nil
    (if (variable-p (caar sigma))
        (cons (cdar sigma) (co-domain-var (cdr sigma)))
      (co-domain-var (cdr sigma)))))

(defun important-variables (sigma delta)
  (append (domain-var sigma)
          (append (domain-var delta)
                  (variables nil (co-domain-var sigma))))))
```

Tal y como está definido, `(important-variables sigma delta)` es una lista que sólo contiene variables (en sentido amplio), para cualesquiera que sean `sigma` y `delta`.

La función `subs-subst` define la relación de subsunción entre sustituciones, usando para ello el segundo de los valores (el que indica éxito o fallo) del multivalor devuelto

por el algoritmo de equiparación `match-mv`, al actuar sobre el sistema $S = \{\sigma(v_1) \approx \delta(v_1), \ldots, \sigma(v_n) \approx \delta(v_n)\}$ (donde $V = \{v_1, \ldots, v_n\}$).

```
(defun subs-subst-mv (sigma delta)
  (let ((V (important-variables sigma delta)))
    (mv-let (system bool)
            (pair-args (apply-subst nil sigma V)
                       (apply-subst nil delta V))
            (match-mv system))))

(defun subs-subst (sigma delta)
  (mv-let (match bool)
          (subs-subst-mv sigma delta)
          bool))
```

Para verificar que `subs-sust` define efectivamente la relación de subsunción entre sustituciones nos hará falta la sustitución testigo de tal relación, que remediará la ausencia de cuantificación existencial (de manera análoga a lo que ocurría con la subsunción entre términos). La demostración del teorema 3.40 nos proporciona la pista para definir esta sustitución: basta tomar un equiparador de S, dado por la función `matching-subst` (es el primero de los valores que devuelve `subs-subst-mv`) y restringirlo a las variables de V, que es lo que hace la función `matching-subst-r`:

```
(defun matching-subst (sigma delta)
  (mv-let (match bool)
          (subs-subst-mv sigma delta)
          match))

(defun matching-subst-r (sigma delta)
  (restriction (matching-subst sigma delta)
               (important-variables sigma delta))))
```

3.4.3 Propiedades de la subsunción entre sustituciones

A continuación presentamos las propiedades principales de la función `subs-subst`, que sirven para verificar formalmente que la función implementa la relación de subsunción entre sustituciones.

Corrección

El siguiente teorema expresa que si `(subs-subst sigma delta)` no falla, entonces existe una sustitución (dada por `(matching-subst-r sigma delta)`) tal que compuesta con `sigma` es igual (como función) a la sustitución `delta` (es decir, `sigma` subsume a `delta` en el sentido de la definición 3.37).

```
(defthm subs-subst-soundness
  (implies (subs-subst sigma delta)
           (equal (instance term
```

```
                    (composition
                      (matching-subst-r sigma delta)
                      sigma))
            (instance term delta))))
```

Completitud

Para formalizar el teorema de completitud de la función `subs-subst`, debemos expresar en la lógica de ACL2 la siguiente propiedad: si σ y δ son dos sustituciones tales que $\sigma \preccurlyeq \delta$, entonces la función `subs-subst` actuando sobre la representación de σ y δ no devuelve fallo. Asumir $\sigma \preccurlyeq \delta$ significa asumir la existencia de una sustitución γ tal que $\delta = \gamma\sigma$. Como ya hemos comentado en la subsección 3.1.3, esta igualdad entre sustituciones la expresamos afirmando que $\delta(t) = \gamma\sigma(t)$, para todo término t (como en la conclusión del teorema anterior). La cuantificación de t hace que no se pueda expresar en ACL2 tal asunción como hipótesis de una implicación. Para suplir esta carencia, usamos el mecanismo de encapsulado.

El siguiente encapsulado expresa la existencia de dos sustituciones `(sigma-w)` y `(delta-w)` que verifican la relación de subsunción entre sustituciones tal y como se define en 3.37. La sustitución `(gamma-w)` representa la sustitución testigo de tal subsunción. Los puntos suspensivos sustituyen a las definiciones locales (que son irrelevantes).

```
(encapsulate
 (((sigma-w) t)
  ((delta-w) t)
  ((gamma-w) t))
  ....
 (defthm sigma-w-delta-w-subsumption-hypothesis
   (equal (instance term (composition (gamma-w) (sigma-w)))
          (instance term (delta-w)))
   :rule-classes nil))
```

Con estas hipótesis, el teorema de completitud se reduce a probar que el valor de `(subs-subst (sigma-w) (delta-w))` es distinto de `nil`:

```
(defthm subs-subst-completeness
  (subs-subst (sigma-w) (delta-w)))
```

Como veremos más adelante este resultado de completitud se puede usar en otras demostraciones mediante el uso de instanciación funcional.

Propiedades de clausura

Podemos demostrar una propiedad de clausura relativa al teorema de corrección, estableciendo que si las dos sustituciones relacionadas por subsunción están definidas en una signatura determinada, la sustitución testigo usada en ese teorema es una sustitución en la misma signatura:

```
(defthm matching-subst-r-substitution-s-p
  (implies (and (substitution-s-p sigma)
```

```
            (substitution-s-p delta))
          (substitution-s-p (matching-subst-r sigma delta)))))
```

Análogamente, podemos establecer una propiedad de clausura para el teorema de completitud. Supongamos que tenemos dos sustituciones σ y δ, en $T(\Sigma, X)$, tales que $\sigma \preccurlyeq \delta$. En otras palabras, existe γ en $T(\Sigma, X)$ tal que $\delta(t) = \gamma(\sigma(t))$, *para todo* $t \in T(\Sigma, X)$. Entonces hemos de probar que `subs-subst` actuando sobre la representación de σ y δ no devuelve fallo. Nótese la diferencia de esta propiedad con la completitud de `subs-subst` expresada anteriormente: estamos suponiendo ahora que las sustituciones involucradas están en una determinada signatura y que la propiedad de subsunción sólo se tiene para los términos de la misma signatura.

Siguiendo una formalización similar a la del teorema `subs-subst-completeness`, podemos formular ese resultado como sigue. En primer lugar, usamos `encapsulate` para establecer las hipótesis:

```
(encapsulate
 (((sigma-w-s) => *)
  ((delta-w-s) => *)
  ((gamma-w-s) => *))
 ...
 (defthm sigma-w-s-substitution-s-p (substitution-s-p (sigma-w-s)))
 (defthm delta-w-s-substitution-s-p (substitution-s-p (delta-w-s)))
 (defthm gamma-w-s-substitution-s-p (substitution-s-p (gamma-w-s)))

 (defthm sigma-w-s-delta-w-s-subsumption-hypothesis
   (implies (term-s-p term)
            (equal (instance term (composition (gamma-w-s) (sigma-w-s)))
                   (instance term (delta-w-s))))))
```

La conclusión buscada la establece el siguiente teorema:

```
(defthm subs-subst-completeness-closure
  (subs-subst (sigma-w-s) (delta-w-s))
```

3.4.4 Descripción de la demostración

El teorema `subs-subst-completeness` se demuestra directamente como consecuencia inmediata del teorema de completitud de `match-mv`. Igualmente, las propiedades de clausura de `subs-subst` son consecuencia directa de las correspondientes propiedades de clausura para `apply-subst` y `match-mv`. Remitimos al lector al libro `subsumption-subst.lisp`.

La demostración de `subs-subst-soundness` es más elaborada y está inspirada en la demostración a mano dada en el apartado *b*) del teorema 3.38. La describimos a continuación con más detalle.

En primer lugar, el siguiente teorema expresa la propiedad fundamental de la función `matching-subst`. Recuérdese que esta función devuelve el equiparador calculado por `match-mv` al actuar sobre el sistema de ecuaciones $S = \{\sigma(v_1) \approx \delta(v_1), \ldots, \sigma(v_n) \approx \delta(v_n)\}$ (donde $V = \{v_1, \ldots, v_n\}$ denota a `(important-variables sigma delta)`) y que

`matching-subst-r` obtiene precisamente la restricción de tal sustitución a las variables de V. Por el teorema de corrección de `match-mv` y por la propia definición de `matching-subst`, se tiene que cuando `(subs-subst sigma delta)` es verdadero, `delta` coincide, en el conjunto V, con la composición de `sigma` y `(matching-subst sigma delta)`:

```
(defthm matching-subst-composed-sigma-coincide-with-delta-in-V
  (let ((V (important-variables sigma delta)))
    (implies (and (variable-p x)
                  (member x V)
                  (subs-subst sigma delta))
             (equal (instance (val x sigma) (matching-subst sigma delta))
                    (val x delta)))))
```

Veamos ahora cómo este resultado sirve para probar el teorema `subs-subst-soundness`. Según la definición de `apply-subst`, para demostrar `subs-subst-soundness` basta demostrar previamente la siguiente versión con `val`:

```
(defthm equal-composition-matching-subst-with-sigma-to-delta-variable
 (implies (and (variable-p x)
               (subs-subst sigma delta))
          (equal (val x (composition (matching-subst-r sigma delta)
                                     sigma))
                 (val x delta))))
```

Para probar este teorema, distinguimos tres casos, según dónde se encuentre `x`.
Caso 1: Supongamos que `x` es una variable en `(domain-var sigma)` (y por tanto en V). Entonces el siguiente teorema afirma que las variables del término `(val x sigma)` están todas dentro de `(co-domain-var sigma)`:

```
(defthm variables-co-domain-var
 (implies (member x (domain-var sigma))
          (subsetp (variables t (val x sigma))
                   (variables nil (co-domain-var sigma)))))
```

Por tanto, según el teorema `subsetp-restriction` (dado en 3.1.3), las sustituciones dadas por `matching-subst` y `matching-subst-r` actúan de la misma manera sobre `(val x sigma)` y en consecuencia podemos usar el lema previo sobre `matching-subst` para concluir este primer caso:

```
(defthm subs-subst-main-property-variable-x-in-domain-var-sigma
  (implies (and (variable-p x)
                (member x (domain-var sigma))
                (subs-subst sigma delta))
           (equal (instance (val x sigma) (matching-subst-r sigma delta))
                  (val x delta)))))
```

Caso 2: Supongamos ahora que `x` está en V pero no en `(domain-var sigma)`. En ese caso, `(val x sigma)` es igual a `x` (según el lema `x-not-in-domain-remains-the-same` de 3.1.3) y por un razonamiento análogo al anterior, `matching-subst` y `matching-subst-r` actúan de la misma manera sobre `(val x sigma)`. Así que nuevamente por el lema anterior sobre `matching-subst` tenemos:

```
(defthm subs-subst-main-property-variable-x-in-V-not-in-domain-var-sigma
  (let ((V (important-variables sigma delta)))
    (implies (and (variable-p x)
                  (not (member x (domain-var sigma)))
                  (member x V)
                  (subs-subst sigma delta))
             (equal (apply-subst t (matching-subst-r sigma delta)
                                 (val x sigma))
                    (val x delta))))))
```

Caso 3: Para el caso en que la variable no se encuentre en V, tenemos que usando el lema `x-not-in-domain-remains-the-same` (en este caso las sustituciones se aplican sobre variables que no están en el dominio) se tiene fácilmente:

```
(defthm subs-subst-main-property-variable-x-not-in-V
  (let ((V (important-variables sigma delta)))
    (implies (and (variable-p x)
                  (not (member x V))
                  (subs-subst sigma delta))
             (equal (instance (val x sigma)
                              (matching-subst-r sigma delta))
                    (val x delta)))))
```

La conjunción de los tres casos discutidos, junto con el teorema que expresa cómo actúa una composición de sustituciones (teorema `composition-of-substitutions-apply` en 3.1.3) implica trivialmente el teorema `equal-composition-subs-subst-with-sigma--to-delta-variable`.

Por último, destacar el uso del consejo `:cases` para hacer que se usen los resultados en una demostración que está basada en una distinción de casos que el demostrador no encuentra por sí mismo. El siguiente consejo basta para que el sistema use los teoremas probados previamente para cada uno de los tres casos descritos en la subsección anterior:

```
;   :hints
;   (("Goal"
;     :cases ((not (member x (important-variables sigma delta)))
;             (member x (domain-var sigma)))))))
```

Sumario

En este capítulo:

- Hemos visto cuestiones relativas a la representación en ACL2 de los términos, las sustituciones y los sistemas de ecuaciones. Hemos explicado que, debido a la estructura recursiva de los términos, gran parte de la teoría que se presenta se desarrolla tanto para términos como para listas de términos. Esto tiene un impacto positivo sobre los esquemas de inducción generados por el demostrador.
- Hemos descrito las nociones de término propio e impropio y cómo intuitivamente podemos ver cualquier objeto ACL2 como la representación de un término, en sentido amplio.
- Hemos definido y verificado un algoritmo de equiparación de sistemas de ecuaciones, basado en reglas de transformación. A partir de este algoritmo, hemos definido la relación de subsunción entre términos.
- Finalmente, hemos definido y verificado la relación de subsunción entre sustituciones.

Capítulo 4

El retículo de los términos de primer orden

El conjunto de los términos de primer orden (en una signatura) posee una estructura de retículo completo respecto del preorden de subsunción. La relación de subsunción y sus propiedades reticulares tienen un interés práctico además del puramente teórico. Por ejemplo, la existencia de la instancia más general de dos términos, su supremo, queda demostrada mediante la definición de un algoritmo de unificación. Este algoritmo es esencial en deducción automática; por ejemplo, para el cálculo de pares críticos de sistemas de reescritura, en el algoritmo de SLD-resolución en programación lógica o en las reglas de resolución y paramodulación. Por otro lado, la existencia de la generalización más particular de dos términos, su ínfimo, se demuestra mediante la definición de un algoritmo de anti–unificación. El proceso de anti–unificación juega un importante papel en muchos algoritmos de aprendizaje automático.

En este capítulo demostraremos formalmente, en la lógica de ACL2, que la relación de subsunción entre términos definida en el capítulo anterior, dota de una estructura de retículo bien fundamentado al conjunto de los términos de primer orden en una signatura. Como consecuencia de esta formalización, se han verificado una serie de funciones de manipulación de términos, ejecutables en ACL2. El resultado más importante de este capítulo es la verificación automática del algoritmo de unificación de Martelli y Montanari.

Este capítulo se estructura de la siguiente manera. En la primera sección definimos la equivalencia de términos respecto de la relación de subsunción y demostramos sus principales propiedades. Además presentamos un algoritmo de renombrado de variables y lo verificamos formalmente. En la segunda sección, demostramos una interesante propiedad del preorden de subsunción: su buena fundamentación. En la siguiente sección, probamos que todo par de términos tiene un ínfimo respecto del preorden de subsunción. Para ello, definimos y verificamos un algoritmo de anti–unificación de pares de términos. La cuarta sección presenta un algoritmo de unificación de términos, basado en reglas de transformación. Una vez verificado este algoritmo, podemos demostrar en la sección quinta que cualquier par de términos que tengan una instancia común tiene un supremo respecto del preorden de subsunción. Por último, recopilamos todos los resultados anteriores para concluir que el conjunto de los términos de primer orden en una signatura tiene estructura de retículo bien fundamentado.

4.1 Renombrados

En esta sección presentamos la formalización en ACL2 de las sustituciones que renombran variables, la equivalencia entre términos y la relación entre ambos conceptos. Además, definimos una clase particular de renombrados y verificamos sus principales propiedades. Los eventos ACL2 que conducen a los resultados presentados en esta sección se encuentran en el libro `renamings.lisp`.

4.1.1 Preliminares

Definición 4.1 *Si $s \preccurlyeq t$ y $t \preccurlyeq s$, decimos que s y t son equivalentes respecto de la relación de subsunción, o simplemente* **equivalentes**, *y lo notamos por $s \equiv t$. Si $s \preccurlyeq t$ pero no es cierto que $s \equiv t$, entonces decimos que s subsume estrictamente a t y lo notamos por $s \prec t$.*

Ejemplo 4.2 $f(x, h(y, x)) \equiv f(y, h(x, y))$ ya que si $\xi = \{x \mapsto y, y \mapsto x\}$ entonces $\xi(f(x, h(y, x)) = f(y, h(x, y))$ y $\xi(f(y, h(x, y)) = f(x, h(y, x))$.

Teorema 4.3 *La relación $\equiv$ en el conjunto $T(\Sigma, X)$ es una relación de equivalencia.*

El hecho de que $\preccurlyeq$ sea un preorden nos permite concluir este teorema. Podemos entonces hablar del conjunto cociente $T(\Sigma, X)/\equiv$. Dado un término $t \in T(\Sigma, X)$, a la clase de equivalencia de t respecto de $\equiv$ la notaremos por $[t]$. Si $t_1, t_2, s_1, s_2 \in T(\Sigma, X)$, $t_1 \equiv t_2$, $s_1 \equiv s_2$ y $t_1 \preccurlyeq s_1$, entonces $t_2 \preccurlyeq s_2$. Por tanto, podemos definir la relación $\preccurlyeq$ en $T(\Sigma, X)/\equiv$ como $[t] \preccurlyeq [s]$ si y sólo si $t \preccurlyeq s$.

Es posible caracterizar la equivalencia entre términos como un cambio de nombre de las variables que aparecen en el término.

Definición 4.4 *Sea $V \subseteq X$. Decimos que una sustitución ξ es una* **sustitución variable** *en V si $\mathcal{C}(\xi_{|V}) \subseteq X$. Si además se tiene que para cualesquiera variables $x, y \in V$, $\xi(x) = \xi(y)$ implica que $x = y$, entonces decimos que ξ es un* **renombrado** *de las variables de V.*

Ejemplo 4.5 La sustitución $\delta = \{x \mapsto y, y \mapsto v, z \mapsto u\}$ es un renombrado de las variables de $\{x, y, z\}$ y sin embargo no es un renombrado de las variables de $\{x, y, z, u\}$. La sustitución ξ del ejemplo 4.2 es un renombrado de las variables de cualquier $V \subseteq X$.

El siguiente teorema expresa la relación existente entre los conceptos de equivalencia de términos y de renombrado de variables.

Teorema 4.6 *Sean $s, t \in T(\Sigma, X)$. Entonces $s \equiv t$ si y sólo si existe una sustitución ξ tal que ξ es un renombrado de las variables de s y $\xi(s) = t$.*

Demostración:
$\boxed{\Longrightarrow}$ Supongamos que $s \equiv t$. Por tanto, existen sustituciones σ y δ tales que $\sigma(s) = t$ y $\delta(t) = s$ y en consecuencia, $\delta(\sigma(s)) = s$. Esto implica que para toda $x \in \mathcal{V}(s)$ se tiene $\delta(\sigma(x)) = x$. Sea $\xi = \sigma_{|\mathcal{V}(s)}$. Es claro que $\xi(s) = \sigma(s) = t$. Veamos que ξ es un renombrado de $\mathcal{V}(s)$. Si $x \in \mathcal{V}(s)$, entonces $\delta(\xi(x)) = \delta(\sigma(x)) = x$. Lo que implica necesariamente que $\xi(x) \in X$ y por tanto ξ es una sustitución variable en $\mathcal{V}(s)$. Por otro lado, si $x, y \in \mathcal{V}(s)$ y $\xi(x) = \xi(y)$, entonces $x = \delta(\sigma(x)) = \delta(\xi(x)) = \delta(\xi(y)) = \delta(\sigma(y)) = y$. Esto demuestra que ξ es un renombrado de las variables de s.

$\boxed{\Longleftarrow}$ Supongamos que ξ es un renombrado de las variables de $\mathcal{V}(s)$ tal que $\xi(s) = t$. Por tanto, $s \preccurlyeq t$ y quedará probar que $t \preccurlyeq s$. Si $\mathcal{V}(s) = \{x_1, \ldots, x_n\}$, entonces $\xi(x_1), \ldots, \xi(x_n)$ son n variables distintas y podemos hablar de la sustitución $\gamma = \{\xi(x_1) \mapsto x_1, \ldots, \xi(x_n) \mapsto x_n\}$. Es claro que si $x \in \mathcal{V}(s)$, $\gamma(\xi(x)) = x$. Por tanto, $\gamma(t) = \gamma(\xi(s)) = s$, lo que demuestra que $t \preccurlyeq s$. □

4.1.2 Renombrados y términos equivalentes

Esta subsección se dedica a la definición y formalización en la lógica de ACL2 del concepto de equivalencia entre términos y de sustituciones de renombrado. Para verificar las definiciones escogidas, hemos demostrado formalmente el teorema 4.6.

Formalización

La equivalencia entre términos es sencilla de definir a partir de la relación de subsunción. La función `renamed` implementa tal concepto[1]:

```
(defun renamed (t1 t2)
  (if (subs t1 t2)
      (if (subs t2 t1) t nil)
    nil))
```

Veamos ahora cómo definir los renombrados de variables, para expresar formalmente en ACL2 la definición 4.4. Realizaremos una pequeña modificación en estas definiciones: en lugar de definir renombrados y sustituciones variables como conceptos relativos a un conjunto de variables V, los definiremos respecto al dominio de la sustitución, evitando así la introducción de un parámetro extra. Esto no afectará a los resultados principales, como veremos.

En primer lugar, definimos el concepto de sustitución variable, implementado por la función `variable-substitution`:

```
(defun variable-substitution (sigma)
  (if (atom sigma)
      t
    (and (variable-p (cdar sigma))
         (variable-substitution (cdr sigma)))))
```

La función `renaming` define el concepto de renombrado. La inyectividad de la sustitución se traduce aquí por el hecho de que no haya elementos repetidos en el codominio de la sustitución (la función `setp`, cuya definición omitimos, comprueba que no existen elementos repetidos en una lista):

```
(defun renaming (sigma)
  (and (variable-substitution sigma)
       (setp (co-domain sigma))))
```

[1] Usamos esta definición en lugar de `(and (subs t1 t2) (subs t2 t1))` para conseguir que la función `renamed` devuelva un valor booleano. Esto será necesario para definirla como relación de equivalencia mediante `defequiv`. Recuérdese que las únicas propiedades que estamos asumiendo sobre `subs` son su corrección, completitud y clausura.

A continuación presentamos los dos teoremas principales de esta sección, que demuestran formalmente el teorema 4.6 y que sirven para verificar que las definiciones anteriores formalizan el concepto deseado.

```
(defthm renaming-implies-renamed
  (implies (and (renaming sigma)
                (subsetp (variables t term) (domain sigma)))
           (renamed term (instance term sigma))))

(defthm renamed-implies-renaming
  (let ((ren (normal-form-subst t (matching t1 t2) t1)))
    (implies (renamed t1 t2)
             (and (renaming ren)
                  (equal (instance t1 ren) t2)))))
```

El teorema `renaming-implies-renamed` establece que toda sustitución de renombrado cuyo dominio contiene a las variables de un término, aplicada sobre dicho término obtiene otro término equivalente. El teorema `renamed-implies-renaming` muestra que si dos términos son equivalentes, existe un renombrado que transforma un término en otro. Este renombrado se obtiene mediante la expresión `(normal-form-subst t (matching t1 t2) t1)`.

La demostración en ACL2 del teorema `renaming-implies-renamed`

La demostración formal del teorema está basada en la prueba a mano del teorema 4.6. En primer lugar, definimos el concepto de sustitución inversa, simplemente cambiando el orden de los pares de la lista de asociación que representa a la sustitución:

```
(defun inverse (sigma)
  (if (atom sigma)
      nil
    (cons (cons (cdar sigma) (caar sigma))
          (inverse (cdr sigma)))))
```

En el caso de los renombrados, y restringiéndonos a las variables del dominio, la inversa, tal y como se ha definido, se comporta como una función inversa:

```
(defthm val-val-inverse-renaming
  (implies (and (renaming sigma)
                (member x (domain sigma)))
           (equal (val (val x sigma) (inverse sigma)) x)))
```

Esto hace que cuando `sigma` sea un renombrado y `term` un término (o lista de términos) cuyo conjunto de variables esté contenido en el dominio de `sigma`, entonces la sustitución `(inverse sigma)` sirve para obtener el término `term` a partir de `(instance term sigma)`:

```
(defthm renaming-inverse
  (implies (and (renaming sigma)
                (subsetp (variables flg term) (domain sigma)))
           (equal (apply-subst flg (inverse sigma)
                               (apply-subst flg sigma term))
                  term)))))
```

Por tanto, en este caso la sustitución `(inverse sigma)` sirve como testigo de que `(instance term sigma)` y `term` están relacionados respecto de la relación de subsunción. Usando la completitud de `subs`, podemos entonces probar el teorema `renaming-implies--renamed`.

La demostración en ACL2 del teorema `renamed-implies-renaming`

Nuevamente la prueba formal de este teorema está basada en la prueba presentada en el teorema 4.6. En este caso, estamos asumiendo `(renamed t1 t2)` y debemos encontrar un renombrado tal que aplicado a `t1` obtenga `t2`. Por el teorema de corrección de la relación de subsunción, podemos transformar las hipótesis a la siguiente situación: tenemos dos sustituciones `sigma` y `delta`, y un término `term`[2], tales que `delta` aplicado al resultado de aplicar `sigma` a `term` obtiene nuevamente el término `term`.

En tal situación[3], la composición de `delta` con `sigma` se comporta como la identidad en el conjunto de las variables de `term`:

```
(defthm identity-on-term-identity-val
  (implies (and (equal (apply-subst flg delta
                                     (apply-subst flg sigma term))
                       term)
                (member x (variables flg term)))
           (equal (instance (val x sigma) delta) x)))
```

Por tanto, `sigma` es inyectiva en el conjunto de variables de `term`:

```
(defthm renamed-implies-injective-val
  (implies (and (equal (apply-subst flg delta
                                     (apply-subst flg sigma term))
                       term)
                (member x (variables flg term))
                (member y (variables flg term))
                (not (equal x y)))
           (not (equal (val x sigma) (val y sigma)))))
```

Nótese que de lo anterior no podemos deducir que no haya elementos repetidos en el codominio de `sigma`, ya que nada se puede decir de lo que ocurra fuera de las variables de `term`. Sin embargo, podemos tomar la restricción de `sigma` a las variables de `term`, lo cual no afecta a la acción de la sustitución sobre el término `term` (teorema `subsetp-restriction` del capítulo anterior). Esto es lo que hace justamente la macro

[2] Que *a posteriori* serán respectivamente `(matching t1 t2)`, `(matching t2 t1)` y `t1`.
[3] Generalizando a términos y a listas de términos.

`normal-form-subst` definida en el capítulo anterior: restringir el dominio de una sustitución al conjunto[4] de las variables de un término dado. Con esta transformación, el teorema `renamed-implies-injective-val` nos sirve para obtener el siguiente resultado, que expresa que no existen elementos repetidos en el codominio de la restricción de `sigma` a las variables de `term`:

```
(defthm renamed-implies-setp-codomain
  (implies (equal (apply-subst flg delta
                               (apply-subst flg sigma term)) term)
           (setp (co-domain (normal-form-subst flg sigma term)))))
```

Por otro lado, bajo las mismas hipótesis, se tiene necesariamente que `sigma` actuando sobre una variable de `term` ha de devolver una variable:

```
(defthm renamed-implies-variable-val
  (implies (and (equal (apply-subst flg delta
                                    (apply-subst flg sigma term))
                       term)
                (member x (variables flg term)))
           (variable-p (val x sigma))))
```

Y por tanto, la restricción de `sigma` a las variables de `term` es una sustitución variable:

```
(defthm renamed-implies-variable-substitution
  (implies (equal (apply-subst flg delta
                               (apply-subst flg sigma term))
                  term)
           (variable-substitution (normal-form-subst flg sigma term))))
```

A partir de los teoremas `renamed-implies-variable-substitution` y `renamed-implies-setp-codomain` podemos deducir que la restricción de `sigma` a las variables de `term` es un renombrado:

```
(defthm renamed-implies-renaming-main-lemma
  (implies (equal (apply-subst flg delta
                               (apply-subst flg sigma term))
                  term)
           (renaming (normal-form-subst flg sigma term)))
```

Como caso particular de este teorema (para términos), sustituyendo `sigma` por `(matching t1 t2)`, `delta` por `(matching t2 t1)` y `term` por `t1`, podemos ahora concluir (usando también el teorema de corrección de `subs`) que la sustitución definida por la expresión `(normal-form-subst t (matching t1 t2) t1)` es un renombrado que actuando sobre `t1` obtiene `t2`. Es decir: el teorema `renamed-implies-renaming`.

En el primer apartado de la subsección C.2.1, detallamos algunas cuestiones adicionales sobre la demostración automática de todos estos resultados.

[4] Recuérdese que se hacía `make-set` sobre el conjunto de variables del término.

4.1.3 Renombrados numéricos

En esta sección vamos a dar un procedimiento particular de obtener un término equivalente a uno dado. Se trata de sustituir las variables del término por números, secuencialmente[5]. Llamaremos a este tipo de operación un *renombrado numérico*. Además de proporcionar un interesante ejemplo de verificación de un procedimiento definido por recursión en la estructura de los términos, este tipo de renombrados nos servirá para separar las variables de dos términos (cuando sea necesario) y para mejorar la presentación en la salida de algunos procedimientos (secciones 4.5 y 7.4.2).

Definiciones y propiedades principales

En la figura 4.1 presentamos la definición de la función `number-rename` que implementa el renombrado numérico de términos.

```
(defun number-rename-aux (flg term sigma x y)
  (if flg
     (if (variable-p term)
         (let ((find-term (assoc term sigma)))
            (if find-term
                (mv (cdr find-term) sigma)
              (let ((y (if (endp sigma) x (+ y (cdar sigma)))))
                (mv y (cons (cons term y) sigma)))))
      (mv-let (renamed-args renaming-args)
              (number-rename-aux nil (cdr term) sigma x y)
              (mv (cons (car term) renamed-args) renaming-args)))
    (if (endp term)
        (mv term sigma)
      (mv-let (renamed-car renaming-car)
              (number-rename-aux t (car term) sigma x y)
              (mv-let (renamed-cdr renaming-cdr)
                      (number-rename-aux
                         nil (cdr term) renaming-car x y)
                      (mv (cons renamed-car renamed-cdr)
                          renaming-cdr))))))

(defun number-rename (term x y)
  (mv-let (renamed renaming)
          (number-rename-aux t term nil x y)
          renamed))
```

Figura 4.1: Renombrado numérico

La función `number-rename` se ayuda de una función auxiliar `number-rename-aux`, que recibe cinco argumentos como entrada:

[5] Recuérdese que estamos considerando que los números, como cualquier objeto atómico, representan variables.

- Un indicador `flg`, que representa si el segundo argumento debe ser interpretado como una lista de términos (`flg`=`nil`) o como un término (`flg`≠`nil`). Se trata, pues, de una definición recursiva en la estructura de los términos.

- El término o lista de términos `term`.

- Una sustitución `sigma`, que durante el proceso va a representar a la sustitución de renombrado parcialmente calculada hasta el momento.

- Un número `x`, que indica el valor inicial que toman las variables del término renombrado.

- Un número `y`, que especifica cómo asignar un nuevo número a una nueva variable encontrada en el proceso de renombrado: se incrementa en `y` el número correspondiente a la última asignación realizada hasta el momento.

La función `number-rename-aux` devuelve dos valores mediante un multivalor. Como primer valor, el término equivalente a `term` obtenido renombrando las variables de `term` de manera que cada variable se sustituye por un número, suponiendo que ya se tienen una serie de asignaciones en `sigma`. Las variables de `term` que ya tengan asignado número en `sigma` se sustituyen por ese número. Si una variable no tiene asignado número en `sigma`, se le asigna el numero resultante de incrementar en `y` el último número asignado en `sigma` (si `sigma` es vacía, se le asigna el número `x`). Como segundo valor, la función `number-rename-aux` devuelve la sustitución de renombrado que transforma `term` en el término equivalente que se devuelve como primer valor.

La función `number-rename` recibe un término `term` y dos números `x` e `y` y llama a `(number-rename-aux t term nil x y)`, para obtener un término equivalente a `term`, tal y como se acaba de describir. He aquí algunos ejemplos:

```
ACL2 !>(number-rename '(f x (g y z) (h z y) x (k u (a))) 1 1)
(F 1 (G 2 3) (H 3 2) 1 (K 4 (A)))
ACL2 !>(number-rename '(f x (g y z) (h z y) x (k u (a))) 5 12)
(F 5 (G 17 29) (H 29 17) 5 (K 41 (A)))
ACL2 !>(number-rename '(f 1 (h 2 x) (k 1)) 0 -1)
(F 0 (H -1 -2) (K 0))
ACL2 !>(number-rename '(k (h x y) (k (f x) (f y) (f z))) 2/11 -4/5)
(K (H 2/11 -34/55) (K (F 2/11) (F -34/55) (F -78/55)))
```

Veamos a continuación las propiedades de `number-rename` que se han verificado. La principal es que obtiene un término equivalente al que recibe como entrada, siempre que el tercer argumento (el correspondiente al incremento) sea distinto de cero:

```
(defthm number-renamed-term-renamed-term
  (implies (and (acl2-numberp x) (acl2-numberp y) (not (= y 0)))
           (renamed (number-rename term x y) term)))
```

Lo que hace útil al renombrado numérico es que podemos controlar el rango numérico en el que están las variables de un término, una vez renombrado. Son todas números mayores o iguales que el segundo argumento, si el tercer argumento es positivo, o menores

o iguales si el tercer argumento es negativo. Como consecuencia, se puede demostrar una importante propiedad sobre la función `number-rename`, que la hace muy útil para separar las variables de dos términos. Por *separación de variables*[6] de un par de términos, entendemos obtener un par de términos equivalentes a los originales, pero renombrados de tal manera que no tengan variables en común. El siguiente teorema sugiere un método para separar variables de dos términos:

```
(defthm number-rename-standardization-apart
  (implies (and (acl2-numberp x1) (acl2-numberp x2)
                (< x1 x2) (< y1 0) (< 0 y2))
           (disjointp (variables t (number-rename t1 x1 y1))
                      (variables t (number-rename t2 x2 y2))))
```

Según este teorema, para separar las variables de dos términos `t1` y `t2`, basta con renombrar `t1` con incremento negativo y `t2` con incremento positivo, siempre que el número de partida para `t1` sea menor que el número de partida para `t2`. En lo sucesivo, siempre que necesitemos separar las variables de dos términos `t1` y `t2`, los renombraremos a `(number-rename t1 0 -1)` y `(number-rename t2 1 1)`, respectivamente.

Por último, `number-rename` es una operación cerrada para los términos de una signatura. Lo que sigue es la correspondiente propiedad de clausura:

```
(defthm number-rename-term-s-p
  (implies (and (acl2-numberp x) (term-s-p term))
           (term-s-p (number-rename term x y))))
```

Demostración de las propiedades en ACL2

A continuación, describimos esquemáticamente los principales lemas que guían la prueba ACL2 de estas propiedades sobre `number-rename`. En líneas generales, la demostración de una propiedad sobre `number-rename` se tiene como caso particular de la propiedad análoga para `number-rename-aux`, lo que significa que hemos de probar la propiedad para términos y listas de términos a la misma vez (como se ha descrito en la subsección 3.1.3). La definición recursiva de `number-rename-aux` hace que estas pruebas se realicen por inducción en la estructura de los términos.

Centrémonos en primer lugar en el resultado sobre la equivalencia del término renombrado, teorema `number-renamed-term-renamed-term`. En primer lugar, nótese que tal y como se ha diseñado el algoritmo, se tiene que `number-rename-aux` verifica la siguiente propiedad:

```
(defthm term-subsumes-number-renamed-aux-term
  (implies (alistp sigma)
           (equal (apply-subst
                    flg
                    (second (number-rename-aux flg term sigma x y))
                    term)
                  (first (number-rename-aux flg term sigma x y)))))
```

[6] En inglés, *standardization apart*.

Es decir, la sustitución calculada por `number-rename-aux` aplicada al término (o lista de términos) de entrada, es igual al término calculado por `number-rename-aux`. Como consecuencia inmediata (por el teorema de completitud de `subs`) resulta que `term` subsume a `(number-rename term x y)`, como afirma el siguiente teorema:

```
(defthm term-subsumes-number-renamed-term
  (subs term (number-rename term x y)))
```

Para probar la equivalencia, resta concluir que se tiene la subsunción en el sentido contrario. Bastará ver que la sustitución que devuelve `number-rename-aux` es un renombrado. Previamente definimos el siguiente concepto intermedio:

```
(defun acl2-numberp-list-increment (l y)
  (cond ((endp l) t)
        ((endp (cdr l)) (acl2-numberp (first l)))
        (t (and (acl2-numberp (first l))
                (= y (- (first l) (second l)))
                (acl2-numberp-list-increment (cdr l) y)))))
```

La función `acl2-numberp-list-increment` define las listas numéricas tales que cada uno de sus elementos se obtiene a partir del anterior incrementando una cantidad constante. La propiedad que nos interesa sobre este concepto es que aquellas sustituciones cuyo codominio verifica tal propiedad son renombrados (siempre que el incremento sea un número no nulo):

```
(defthm acl2-numberp-list-increment-implies-renaming
  (implies (and (acl2-numberp-list-increment (co-domain sigma) y)
                (acl2-numberp y)
                (not (= y 0)))
           (renaming sigma))))
```

El codominio de la sustitución que construye `number-rename-aux` tiene esa propiedad:

```
(defthm number-rename-co-domain-acl2-numberp-list-increment
  (implies (and (acl2-numberp-list-increment (co-domain sigma) y)
                (acl2-numberp x)
                (acl2-numberp y))
           (acl2-numberp-list-increment
            (co-domain
             (second (number-rename-aux flg term sigma x y))) y))))
```

Podemos ahora aplicar el teorema `renaming-inverse` (página 116) para concluir que la sustitución inversa de la obtenida por `number-rename-aux`, aplicada al término devuelto por la misma función, es igual al término que se recibe como entrada (siempre que el incremento sea no nulo):

```
(defthm number-renamed-aux-term-subsumes-term
  (implies (and (acl2-numberp x) (acl2-numberp y) (not (= y 0)))
           (equal (apply-subst
```

```
                  flg
                  (inverse
                     (second (number-rename-aux flg term nil x y)))
               (first (number-rename-aux flg term nil x y)))
         term)))
```

Por tanto, usando el teorema de completitud de `subs`, podemos concluir que `(number--rename term x y)` subsume a `term`, como afirma el siguiente teorema:

```
(defthm number-renamed-term-subsumes-term
  (implies (and (acl2-numberp x) (acl2-numberp y) (not (= y 0)))
           (subs (number-rename term x y) term)))
```

Esto completa la demostración de que `number-rename` obtiene términos equivalentes bajo subsunción.

Comentemos ahora la demostración del teorema que nos permite separar variables de dos términos. Para ello, se prueba que las variables de un término renombrado son números mayores (o menores, dependiendo del incremento) que el número de partida. Nótese que las variables numéricas que se van necesitando se toman a partir del último número asignado, sumando el incremento (o el número de partida, si es la primera vez). Por tanto, la propiedad clave es que durante el proceso de renombrado, las variables del codominio de la sustitución que se construye cumplen esa restricción. Por ejemplo, para el caso en que el incremento sea positivo, esta propiedad queda formalizada por el siguiente teorema:

```
(defthm number-renamed-aux-variables->=-x
  (implies (and (acl2-numberp-list-bigger-than (co-domain sigma) x)
                (acl2-numberp x) (> y 0))
           (and (acl2-numberp-list-bigger-than
                  (variables
                     flg (first (number-rename-aux flg term sigma x y)))
                  x)
                (acl2-numberp-list-bigger-than
                 (co-domain
                   (second (number-rename-aux flg term sigma x y)))
                 x))))
```

Es destacable que este lema se pruebe a la misma vez para la sustitución y para el término. Un teorema análogo se tiene en el caso de que el incremento sea negativo. Las funciones `acl2-numberp-list-bigger-than` y `acl2-numberp-list-smaller-than` (cuyas sencillas definiciones omitimos aquí), definen la propiedad de ser una lista de números mayor o menor, respectivamente, que uno dado. Con estos resultados y el siguiente lema previo:

```
(defthm smaller-bigger-disjointp
  (implies (and (< x1 x2)
                (acl2-numberp-list-smaller-than l1 x1)
                (acl2-numberp-list-bigger-than l2 x2))
           (disjointp l1 l2))))
```

podemos obtener inmediatamente el teorema que nos permite separar variables (teorema `number-rename-standardization-apart`).

Por último, la propiedad de clausura de `number-rename` (teorema `number-rename--term-s-p`) se tiene probando una propiedad análoga para la función `number-rename-aux`, mediante una sencilla inducción en la estructura de los términos

En el segundo apartado de la subsección C.2.1, explicamos con detalle algunos aspectos relacionados con la demostración automática de estos resultados.

4.1.4 La relación de equivalencia `renamed` y sus congruencias

El hecho de que la relación de subsunción entre términos sea un preorden (sección 3.3.6) hace evidente que la relación `renamed` define una relación de equivalencia. Además es fácil probar que, por lo que respecta a la relación de subsunción, los términos equivalentes (según la definición dada por `renamed`) se comportan de manera equivalente, como expresan los siguientes teoremas:

```
(defthm renamed-implies-iff-subs-1
  (implies (renamed t1 t1-equiv)
           (iff (subs t1 t2) (subs t1-equiv t2))))

(defthm renamed-implies-iff-subs-2
  (implies (renamed t2 t2-equiv)
           (iff (subs t1 t2) (subs t1 t2-equiv))))
```

Desde el punto de vista de las demostraciones, esto significa que para probar subsunción entre dos términos podemos, si fuera conveniente, probar la subsunción entre dos términos que sean equivalentes a los originales.

Estos resultados hacen posible usar el mecanismo de reescritura congruente para que, durante los intentos de prueba que genera el demostrador, se reescriban términos por otros equivalentes respecto de la relación de subsunción. Para ello usamos los eventos `defequiv` y `defcong`. Recuérdese que un evento `defequiv` (página 51) sirve para marcar una relación de equivalencia. Para que sea aceptado, se ha de probar que la relación cumple las propiedades reflexiva, simétrica y transitiva. En el caso de `renamed`, esto es bien simple usando la reflexividad y la transitividad de la relación de subsunción, establecida en la subsección 3.3.6:

```
(defequiv renamed)
```

La definición de las dos siguientes congruencias permitirá al sistema sustituir un término por otro equivalente, cuando el término figure como argumento de la relación de subsunción (estos eventos `defcong` generan la demostración de los dos teoremas `renamed--implies-iff-subs-1` y `renamed-implies-iff-subs-2` presentados anteriormente):

```
(defcong renamed iff (subs t1 t2) 1)

(defcong renamed iff (subs t1 t2) 2)
```

Una vez definida la equivalencia y los lugares donde se pueden sustituir términos por otros equivalentes, una tercera pieza para que el mecanismo de reescritura en ACL2 pueda actuar, es una serie de reglas de reescritura, que indiquen al demostrador que determinadas expresiones representan términos equivalentes a otras expresiones. En este caso, por ejemplo, un teorema de la forma `(implies` *HIP* `(renamed` *A B*`))`, permitirá, durante un intento de prueba, sustituir cualquier instancia de *A* por la correspondiente instancia de *B*, siempre que la correspondiente instancia de *HIP* sea cierta y siempre que la instancia de *A* figure como argumento de la relación `subs`.

Por ejemplo, el teorema `renaming-implies-renamed` actúa como una regla de reescritura, permitiendo reemplazar expresiones que sean instancias de `(instance term sigma)` por la correspondiente instancia de `term`, siempre que se pueda establecer que la correspondiente instancia de `sigma` es un renombrado y que este reemplazamiento se produzca en uno de los dos argumentos de `subs`.

De la misma manera, la regla de reescritura definida por el teorema `number-renamed--term-renamed-term` permite sustituir expresiones cuyo símbolo principal de función sea `number-rename`, por su primer argumento. Es decir, el demostrador sustituirá cualquier expresión que represente a un término renombrado mediante `number-rename`, por el correspondiente término sin renombrar, cuando la expresión aparezca como argumento de `subs`. Esto nos permite automatizar de manera simple y eficaz el razonamiento con los renombrados numéricos, al menos en lo que respecta a sus propiedades relativas a la relación de subsunción.

4.1.5 Renombrado de listas de términos

En la formulación del concepto de par crítico (subsección 7.4.2) necesitaremos renombrar pares de términos en lugar de renombrar términos. Es decir, todas las ocurrencias de una variable en el par se deben sustituir por la misma variable. Por ello (y generalizando el renombrado a listas de términos), definimos la función `number-rename-list`:

```
(defun number-rename-list (l x y)
  (mv-let (renamed renaming)
          (number-rename-aux nil l nil x y)
          renamed))
```

Las propiedades de `number-rename-list` son análogas a las de `number-rename`. En particular, obtiene una lista de términos equivalente a la original (respecto de subsunción entre listas, véase la definición de `subs-list` en la página 104):

```
(defthm list-subsumes-number-renamed-list-
  (subs-list l (number-rename-list l x y)))

(defthm number-renamed-list-subsumes-list
  (implies (and (acl2-numberp x) (acl2-numberp y) (not (= y 0)))
           (subs-list (number-rename-list l x y) l)))
```

Los teoremas sobre separación de variables de `number-rename-list` son análogos a los de `number-rename`. Remitimos al lector a la sección `3.5` del libro `renamings.lisp`.

4.2 Buena fundamentación de la relación de subsunción

En esta sección presentamos una prueba formal en ACL2 de que la relación de subsunción en el conjunto de los términos de primer orden está bien fundamentada. Los eventos descritos en esta sección se encuentran en el libro `subsumption-well-founded.lisp`.

Preliminares

Para ver que el orden $\prec$ de subsunción estricta es un orden bien fundamentado en $T(\Sigma, X)$, la idea intuitiva es la siguiente: si $s \prec t$, entonces o bien el tamaño de s es más pequeño que el de t, o en caso de igualdad de tamaños, el número de variables distintas de s es mayor que el de t y este número es a lo sumo el número de posiciones variables de s. Esto hace que no pueda existir una secuencia infinita de términos cada más generales. La demostración que ahora describimos es la que presenta Gerard Huet en [29].

Lema 4.7 *Sea $t_1, t_2 \in T(\Sigma, X)$ tal que $t_1 \equiv t_2$. Entonces $|t_1| = |t_2|$ y $v(t_1) = v(t_2)$. Por tanto, el tamaño y la apertura están bien definidas sobre el conjunto $T(\Sigma, X)/\equiv$.*

Demostración:
En primer lugar, obsérvese que si $t_1 \equiv t_2$, existen sustituciones ξ_1 y ξ_2 tales que $t_2 = \xi_1(t_1)$ y $t_1 = \xi_2(t_2)$ y tales que son renombrados de las variables de t_1 y t_2, respectivamente. Para demostrar que $|t_1| = |t_2|$, basta probar por inducción en la estructura de t_1 que $|t_1| = |\xi_1(t_1)|$, teniendo en cuenta que ξ_1 es una sustitución variable. Para ver que $v(t_1) = v(t_2)$, nótese que si $x \in \mathcal{V}(t_1)$, entonces $\xi_1(x) \in \mathcal{V}(t_2)$. Análogamente, si $x \in \mathcal{V}(t_2)$, $\xi_2(x) \in \mathcal{V}(t_1)$. Puesto que ξ_1 y ξ_2 son inyectivas en $\mathcal{V}(t_1)$ y $\mathcal{V}(t_2)$, respectivamente, esto implica que $v(t_1) \leq v(t_2)$ y que $v(t_1) \geq v(t_2)$. □

Los dos siguientes lemas se tienen por una sencilla inducción en la estructura de los términos.

Lema 4.8 *Si $s, t \in T(\Sigma, X)$ y $s \preccurlyeq t$, entonces $|s| \leq |t|$.*

Lema 4.9 *Dada una sustitución σ y un término t, $|t| = |\sigma(t)|$ si y sólo si σ es una sustitución variable en $\mathcal{V}(t)$*

Lema 4.10 *Sean $s, t \in T(\Sigma, X)$ tal que $s \preccurlyeq t$ y $|s| = |t|$. Entonces $v(s) \geq v(t)$. Además, bajo esas condiciones, $v(s) = v(t)$ si y sólo si $s \equiv t$*

Demostración:
Supongamos $s \preccurlyeq t$ y $|s| = |t|$, y sea σ una sustitución tal que $\sigma(s) = t$. Por el lema 4.9 se tiene $\sigma(x) \in X$, para $x \in \mathcal{V}(s)$. Esto implica (por inducción en la estructura de s) que $\mathcal{V}(t) = \{\sigma(x) : x \in \mathcal{V}(s)\}$. Por tanto, $v(t) \leq v(s)$. Además la igualdad se obtiene si y sólo si σ es inyectiva de $\mathcal{V}(s)$ en $\mathcal{V}(t)$, es decir, en el caso en el que σ sea un renombrado de las variables de $\mathcal{V}(s)$, lo que implicaría, por el teorema 4.6, que $t = \sigma(s) \equiv s$. □

Definición 4.11 *Sean $t_1, \ldots, t_n \in T(\Sigma, X)$ y $\xi_1, \ldots, \xi_n$ sustituciones. Decimos que las sustituciones ξ_i* **separan** *a los términos t_i ($1 \leq i \leq n$), si para todo $1 \leq i \leq n$, ξ_i es un renombrado de las variables de t_i y para todo $1 \leq i < j \leq n$, se tiene que $\mathcal{V}(\xi_i(t_i)) \cap \mathcal{V}(\xi_j(t_j)) = \emptyset$.*

Definición 4.12 *La* **proyección** *de un término t, $p(t)$, es el resultado de aplicar a t la función $p : T(\Sigma, X) \longrightarrow T(\Sigma, X)/\equiv$ definida de la siguiente manera:*

$$p(t) = \begin{cases} X & \text{si } t \in X \\ [f(\xi_1(p_1), \ldots, \xi_n(p_n))] & \text{si } t = f(t_1, \ldots, t_n) \end{cases}$$

donde $p_i \in p(t_i)$, y ξ_i son sustituciones que separan a los p_i, $(1 \leq i \leq n)$.

Intuitivamente, p distingue todas las ocurrencias de las variables de un término. Es fácil ver que p está bien definida. Es decir, el resultado no depende de los $p_i \in p(t_i)$ y los ξ_i escogidos.

Ejemplo 4.13 Calculemos $p(f(x, g(x, h(y, y), y)))$.

$$p(x) = p(y) = X$$

$$p(h(y, y)) = [h(u, v)]$$

$$p(g(x, h(y, y))) = [g(x, h(u, v))]$$

$$p(f(x, g(x, h(y, y)), y)) = [f(x, g(w, h(u, v)), y)]$$

El siguiente lema enuncia una serie de propiedades sencillas sobre la función de proyección p. La demostración de estas propiedades, que omitimos, es fácil mediante inducción en la estructura de los términos.

Lema 4.14 *Sea p la función de proyección y $t, s \in T(\Sigma, X)$. Entonces se verifican las siguientes propiedades:*

1. *Si $t \equiv s$, entonces $p(t) = p(s)$.*
2. *$p(t) \preccurlyeq [t]$.*
3. *Si $s \in p(t)$, entonces $p(s) = p(t)$.*
4. *Si $t = f(t_1, \ldots, t_n)$, $s = f(s_1, \ldots, s_n)$ y $p(t_i) = p(s_i)$, para $1 \leq i \leq n$, entonces $p(t) = p(s)$.*

El siguiente lema relaciona la función de proyección con el tamaño y la apertura de los términos.

Lema 4.15 *Sea p la función de proyección y $t, s \in T(\Sigma, X)$. Entonces se verifican las siguientes propiedades:*

1. *Si $s \preccurlyeq t$ y $|s| = |t|$, entonces $p(s) = p(t)$.*
2. *$|p(t)| = |t|$.*
3. *Si $s \in p(t)$, entonces $s \preccurlyeq t$.*
4. *$\upsilon(t) \leq \upsilon(p(t))$.*
5. *Si $s \prec t$ y $|s| = |t|$, entonces $\upsilon(t) < \upsilon(s) \leq \upsilon(p(t))$*

Demostración:

1) Por inducción en la estructura de s:

i) Si $s = x \in X$, entonces $0 = |s| = |t|$, luego $t \in X$ y por tanto $p(s) = p(t)$.

ii) Si $s = f(s_1, \ldots, s_n)$, entonces $t = f(\sigma(s_1), \ldots, \sigma(s_n))$ para cierta sustitución σ. Por el lema 4.9 se tiene que $\sigma(x) \in X$ para toda $x \in \mathcal{V}(s)$ y por tanto $|\sigma(s_i)| = |s_i|$. Aplicando la hipótesis de inducción, $p(s_i) = p(\sigma(s_i))$ y por el lema 4.14 (apartado 4)) $p(s) = p(t)$.

2) Por inducción en la estructura de t y teniendo en cuenta que el lema 4.7 nos permite calcular el tamaño de una clase con cualquiera de sus representantes,

i) Si $t = x \in X$, entonces $|p(t)| = |t| = 0$

ii) Si $t = f(t_1, \ldots, t_n)$, entonces $p(t) = [f(\xi_1(p_1), \ldots, \xi_n(p_n))]$ con ξ_i separando a los p_i, $p_i \in p(t_i)$. Se tiene que

$$|p(t)| = 1 + \sum_{i=1}^{n} |\xi_i(p_i)| = 1 + \sum_{i=1}^{n} |p_i|$$

Aplicando la hipótesis de inducción,

$$|p(t)| = 1 + \sum_{i=1}^{n} |t_i| = |t|.$$

3) Por el lema 4.14 (apartado 2)) $p(t) \preccurlyeq [t]$ y como $s \in p(t)$ y $t \in [t]$, entonces $s \preccurlyeq t$.

4) Por el lema 4.7 basta probar que si $s \in p(t)$, entonces $\upsilon(t) \leq \upsilon(s)$. Por 3) $s \preccurlyeq t$ y por 2) $|s| = |t|$. Podemos aplicar entonces el lema 4.10 para concluir que $\upsilon(t) \leq \upsilon(s)$.

5) Por el lema 4.10 $\upsilon(s) > \upsilon(t)$. Por 1), $p(s) = p(t)$. Por 4) $\upsilon(s) \leq \upsilon(p(s)) = \upsilon(p(t))$. Luego $\upsilon(t) < \upsilon(s) \leq \upsilon(p(t))$.

□

Teorema 4.16 *El orden de subsunción estricta en $T(\Sigma, X)$ está bien fundamentado.*

Demostración:
Supongamos que existiera una secuencia infinita:

$$t_1 \succ t_2 \succ \cdots \succ t_i \succ t_{i+1} \succ \cdots$$

Entonces, por el lema 4.8 se tiene que $|t_i| \geq |t_{i+1}|$. Luego existe $k \in \mathbb{N}$ tal que $\forall i \geq k, |t_i| = |t_{i+1}|$ Por el lema 4.15(5) se tiene que

$$\upsilon(p(t_k)) \geq \cdots \upsilon(t_{k+1}) > \upsilon(t_k).$$

Esto implicaría la existencia de una sucesión infinita de números naturales acotada superiormente y estrictamente creciente, lo cual es imposible. □

Formalización en ACL2

Como ya se comentó en el capítulo 2 (página 54), las relaciones bien fundamentadas en ACL2 se definen mediante la demostración del correspondiente teorema de buena fundamentación asociado a la relación. En este teorema se debe especificar un determinado conjunto de medida y una función monótona (una inmersión) en el conjunto de los ordinales menores que ε_0. Es lo que haremos a continuación. En primer lugar, definimos la relación de subsunción estricta entre términos:

```
(defun strict-subs (t1 t2) (and (subs t1 t2) (not (subs t2 t1))))
```

A continuación definimos la función de inmersión `subsumption-measure`:

```
(defun subsumption-measure (term)
  (cons (1+ (size t term))
        (- (len (variables t term))
           (len (make-set (variables t term))))))
```

Esta función devuelve un ordinal para cada término que recibe como entrada. Tal y como está definida, se consigue obtener el orden lexicográfico en $\mathbb{N} \times \mathbb{N}$, a partir del orden entre ordinales (véase la sección 2.8 de [7]). Es decir, el comparar los ordinales `(subsumption-measure t1)` y `(subsumption-measure t2)` mediante `e0-ord-<`, tiene el efecto de comparar lexicográficamente pares de números naturales. En primer lugar se compara el tamaño de los términos. En caso de igualdad de tamaños, se compara la diferencia entre el número de posiciones correspondientes a variables y el número de variables distintas. Nótese que el número de posiciones variables de un término viene determinado por la longitud de su lista de variables y el número de variables distintas se puede obtener eliminado repeticiones (con la función `make-set`) de esa lista de variables y calculando la longitud de la lista resultante.

Lo que sigue es el teorema de buena fundamentación correspondiente a la relación de subsunción entre términos de primer orden:

```
(defthm subsumption-well-founded
  (and (e0-ordinalp (subsumption-measure t1))
       (implies (strict-subs t1 t2)
                (e0-ord-< (subsumption-measure t1)
                          (subsumption-measure t2))))
  :rule-classes :well-founded-relation))
```

Como se observa, el conjunto de medida en este caso viene definido por el predicado `t`. Es decir, se trata de una relación bien fundamentada sobre todos los objetos ACL2 (tanto términos propios como términos impropios). Como caso particular, el teorema es cierto para los términos propios de una signatura (no es necesario en este caso probar propiedades de clausura).

La demostración en ACL2

Presentamos aquí los lemas que guían a ACL2 a la prueba del teorema `subsumption-well-founded` presentado anteriormente. Como queda reflejado en la definición de la

función `subsumption-measure`, la idea intuitiva que inspira la prueba ACL2 es la misma que en la prueba de Huet presentada en la subsección anterior. Sin embargo, al no poder manejar el concepto de clase de equivalencia como un objeto individual en la lógica ACL2, la prueba a mano y la presentada aquí difieren bastante desde el punto de vista técnico. Por un lado, la prueba ACL2 está completamente influenciada por la representación de las sustituciones y de los conjuntos de variables mediante listas. Por otro lado, la prueba ACL2 resulta más simple que la prueba a mano.

Para describir la prueba del teorema de buena fundamentación de `strict-subs`, nos centraremos en la demostración de que la función `subsumption-measure` es una inmersión (la prueba de que dicha función siempre devuelve un ordinal es muy sencilla).

Supongamos por tanto que tenemos `(subs t1 t2)` y `(not (subs t2 t1))`. Por los teoremas de corrección y completitud de la relación de subsunción, podemos transformar las hipótesis a la siguiente situación: tenemos una sustitución `sigma`[7] y dos términos, `t1` e `(instance t1 sigma)`. Además, *ninguna* sustitución aplicada a `(instance t1 sigma)` obtiene `t1`. Partiendo de estas suposiciones, hemos de probar que `(subsumption-measure t1)` es mayor, respecto al orden `e0-ord-<`, que `(subsumption-measure (instance t1 sigma))`. Gran parte de la demostración se tiene tanto para el caso de que `t1` sea un término como para el caso en el que `t1` sea una lista de términos, de manera simultánea.

En primer lugar, el tamaño de una instancia de un término, nunca es menor que el del propio término, como establece el siguiente teorema:

```
(defthm size-instance-geq
  (>= (size flg (apply-subst flg sigma t1)) (size flg t1)))
```

Si el tamaño es estrictamente menor, ya se tiene el resultado buscado. En caso de que se dé la igualdad de tamaños entre un término y una de sus instancias, la sustitución que realiza la instanciación debe necesariamente asignar variables a las variables del término. Nótese que esto no nos permite concluir directamente que tal sustitución sea una sustitución variable, ya que nada podemos asegurar de lo que ocurra para variables que no están en el término. Sin embargo, esto es cierto si nos restringimos a las variables del término. Como se ha visto en la subsección 3.1.3, esta restricción viene definida por `normal-form-subst`. El siguiente teorema establece este resultado:

```
(defthm size-equal-variable-substitution
  (implies (equal (size flg t1) (size flg (apply-subst flg sigma t1)))
           (variable-substitution (normal-form-subst flg sigma t1)))))
```

Aplicando el teorema `renaming-inverse` (subsección 4.1.2), la sustitución `(normal-form-subst flg sigma t1)` no puede ser un renombrado, ya que entonces su inversa proporcionaría un sustitución testigo de la subsunción entre `(apply-subst flg sigma t1)` y `t1`, lo que estamos suponiendo que no es cierto. Por tanto, al ser una sustitución variable, necesariamente tiene que ocurrir que el codominio de la sustitución `(normal-form-subst flg sigma t1)` no sea un conjunto (es decir, que tenga elementos repetidos):

```
(defthm equal-size-and-not-inverse-subsumption-implies-not-renaming
```

[7] Que *a posteriori* será `(matching t1 t2)`

```
  (let ((sigmar (normal-form-subst flg sigma t1)))
    (implies (and (equal (size flg t1)
                         (size flg (apply-subst flg sigma t1)))
                  (not (equal (apply-subst flg (inverse sigmar)
                                           (apply-subst flg sigma t1))
                              t1)))
             (not (setp (co-domain sigmar))))))
```

Si el codominio de una sustitución no es un conjunto, se verifica que el número de elementos distintos de su codominio es siempre estrictamente menor que el número de elementos de su dominio, como expresa el siguiente teorema:

```
(defthm not-injective-implies-co-domain-lessp-than-domain
  (implies (not (setp (co-domain delta)))
           (< (len (make-set (co-domain delta)))
              (len (domain delta)))))
```

Las siguientes igualdades sirven para conectar el resultado anterior con la desigualdad que se pretende demostrar, ya que permiten expresar el codominio y el dominio de `(normal-form-subst flg sigma t1)` en función de los conjuntos de variables de `t1` y de `(apply-subst flg sigma t1)`, respectivamente:

```
(defthm n-variables-decreases-lemma-1
  (equal (domain (normal-form-subst flg sigma t1))
         (make-set (variables flg t1)))))

(defthm n-variables-decreases-lemma-2
  (implies (equal (size flg t1)
                  (size flg (apply-subst flg sigma t1)))
           (equal (make-set
                    (co-domain (normal-form-subst flg sigma t1)))
                  (make-set
                    (variables flg (apply-subst flg sigma t1))))))
```

Por tanto, tomando `delta` igual a `(normal-form-subst flg sigma t1)` en el teorema `not-injective-implies-co-domain-lessp-than-domain` y usando los lemas anteriores, podemos deducir el siguiente teorema:

```
(defthm n-variables-decreases
  (let ((sigmar (normal-form-subst flg sigma t1)))
    (implies (and (equal (size flg t1)
                         (size flg (apply-subst flg sigma t1)))
                  (not (equal (apply-subst flg (inverse sigmar)
                                           (apply-subst flg sigma t1))
                              t1)))
     (< (len (make-set (variables flg (apply-subst flg sigma t1))))
        (len (make-set (variables flg t1)))))))
```

Es decir, hemos probado que en caso de igualdad de tamaños entre `t1` y `(apply-subst flg sigma t1)`, el número de variables distintas de `t1` es estrictamente mayor que el de `(apply-subst flg sigma t1)`, siempre que ambos términos no sean equivalentes.

Por otro lado, al ser `(normal-form-subst flg sigma t1)` una sustitución variable, podemos deducir que el numero de posiciones en las que ocurre una variable de `t1` es igual al número de posiciones variables de `(apply-subst flg sigma t1)`, como expresa el siguiente teorema:

```
(defthm n-variables-bounded-substitution-variable
  (implies (variable-substitution (normal-form-subst flg sigma t1))
           (equal (len (variables flg (apply-subst flg sigma t1)))
                  (len (variables flg t1)))))
```

Por tanto, si `(normal-form-subst flg sigma t1)` es una sustitución variable (y eso ocurre cuando `t1` y `(apply-subst flg sigma t1)` tienen tamaños iguales), el número de posiciones variables de `t1` constituye una cota superior para el número de variables distintas de `(apply-subst flg sigma t1)`. De los dos últimos teoremas se concluye fácilmente que si `t1` y `(apply-subst flg sigma t1)` tienen tamaños iguales y no son equivalentes, la segunda componente de la medida lexicográfica definida por `subsumption--measure` decrece estrictamente.

Usando propiedades aritméticas básicas de la relación `<` y el teorema de completitud de la relación `subs` (que permite asegurar la segunda hipótesis del teorema `n-variables--decreases`), las anteriores consideraciones nos llevan al siguiente teorema (particularizando para `flg`=`t`):

```
(defthm subsumption-well-founded-instance-version
  (implies (not (subs (instance t1 sigma) t1))
           (e0-ord-< (subsumption-measure t1)
                     (subsumption-measure (instance t1 sigma)))))
```

Finalmente, el teorema de corrección de la relación `subs` (que nos permite reescribir `t2` como instancia de `t1`) hace posible deducir el teorema `subsumption-well-founded` tal y como se ha presentado anteriormente.

4.3 Ínfimo de dos términos. Anti–unificación

En esta sección presentamos la formalización en ACL2 de la existencia de un ínfimo (respecto del preorden de subsunción) para cualquier par de términos. Para ello, definimos y verificamos formalmente un algoritmo de anti–unificación entre términos. Los eventos que conducen hacia los resultados de esta sección se encuentran en el libro `anti-unification-.lisp`.

4.3.1 Preliminares

Definición 4.17 *Sea ϕ una función inyectiva entre $T(\Sigma, X) \times T(\Sigma, X)$ y X (existe pues ambos conjuntos son numerables). Definimos la operación binaria $\wedge_\phi$ en $T(\Sigma, X)$ de la siguiente manera:*

1. $f(s_1, \ldots, s_n) \wedge_\phi f(t_1, \ldots, t_n) = f(s_1 \wedge_\phi t_1, \ldots, s_n \wedge_\phi t_n)$.
2. $s \wedge_\phi t = \phi(s, t)$, *en otro caso*

Esta operación se puede trasladar a las sustituciones mediante la siguiente definición: $(\sigma_1 \wedge_\phi \sigma_2)(x) = \sigma_1(x) \wedge_\phi \sigma_2(x)$.

Ejemplo 4.18 Supongamos que ϕ es tal que $\phi(h(y), g(z)) = u$ y que $\phi(x, g(z)) = v$ entonces:

$$f(h(y), x, h(y)) \wedge_\phi f(g(z), g(z), g(z)) =$$
$$= f(h(y) \wedge_\phi g(z), x \wedge_\phi g(z), h(y) \wedge_\phi g(z)) = f(u, v, u)$$

Como se demostrará, la operación $\wedge_\phi$ implementa un método para encontrar un ínfimo de dos términos. Es por esto que en contraposición con el algoritmo de unificación (que más adelante veremos y que nos servirá para definir el supremo de dos términos), lo llamaremos algoritmo de **anti–unificación**.

El siguiente lema se tiene por una sencilla inducción en la estructura del término t.

Lema 4.19 *Para cualesquiera sustituciones* σ_1 *y* σ_2, *y* $t \in T(\Sigma, X)$, *se tiene que* $(\sigma_1 \wedge_\phi \sigma_2)(t) = \sigma_1(t) \wedge_\phi \sigma_2(t)$.

Teorema 4.20 *Dados* $t_1, t_2 \in T(\Sigma, X)$, *el término* $t_1 \wedge_\phi t_2$ *es* un *ínfimo de* t_1 *y* t_2 *bajo subsunción.*

Demostración:
Sea $t = t_1 \wedge_\phi t_2$. Veamos que t es una cota inferior de t_1 y t_2. Sean $\phi_1, \phi_2 : X \longrightarrow T(\Sigma, X)$ tales que $\phi^{-1}(x) = (\phi_1(x), \phi_2(x))$. Sean $\sigma_1 = \phi_{1|\mathcal{V}(t)}$ y $\sigma_2 = \phi_{2|\mathcal{V}(t)}$. Probemos que $\sigma_1(t) = t_1$ (análogamente $\sigma_2(t) = t_2$). Hagámoslo por inducción en la estructura de t_1:

i) Si $t_1 = x \in X$, entonces

$$\sigma_1(t) = \phi_1(x \wedge_\phi t_2) = \phi_1(\phi(x, t_2)) = x = t_1.$$

ii) Si $t_1 = f(t'_1, \ldots, t'_n)$, pueden ocurrir dos casos:

Caso 1: $t_2 = x \in X$ ó $t_2 = g(t''_1, \ldots, t''_m)$, con $n \neq m$ ó $f \neq g$. Entonces

$$\sigma_1(t) = \phi_1(t_1 \wedge_\phi t_2) = \phi_1(\phi(t_1, t_2)) = t_1.$$

Caso 2: $t_2 = f(t''_1, \ldots, t''_n)$. Entonces

$$\sigma_1(t) = \sigma_1(t_1 \wedge_\phi t_2) = \sigma_1(f(t'_1 \wedge_\phi t''_1, \ldots, t'_n \wedge_\phi t''_n)) = f(\sigma_1(t'_1 \wedge_\phi t''_1), \ldots, \sigma_1(t'_n \wedge_\phi t''_n))$$

Aplicando la hipótesis de inducción, tenemos que $\sigma_1(t) = f(t'_1, \ldots, t'_n) = t_1$.

Veamos ahora que t es mayor que cualquier otra cota inferior. Si s es una cota inferior de t_1 y t_2, sean σ_1 y σ_2 tales que $\sigma_1(s) = t_1$ y $\sigma_2(s) = t_2$. Sea la sustitución $\sigma = (\sigma_1 \wedge_\phi \sigma_2)_{|\mathcal{V}(s)}$. Entonces, $\sigma(s) = (\sigma_1 \wedge_\phi \sigma_2)(s)$. Por el lema 4.19 se tiene que $\sigma(s) = \sigma_1(s) \wedge_\phi \sigma_2(s) = t_1 \wedge_\phi t_2 = t$. Luego $s \preccurlyeq t$. □

```
(defun anti-unify-injection (p phi)
  (let ((found (assoc-equal p phi)))
    (if found
        (mv (cdr found) phi t)
      (let ((y (if (endp phi) 1 (+ 1 (cdar phi)))))
        (mv y (cons (cons p y) phi) t)))))

(defun anti-unify-aux (flg t1 t2 phi)
  (if flg
      (cond ((or (variable-p t1) (variable-p t2))
             (anti-unify-injection (cons t1 t2) phi))
            ((eql (car t1) (car t2))
             (mv-let (a-u-args phi1 bool)
                     (anti-unify-aux nil (cdr t1) (cdr t2) phi)
                     (if bool
                         (mv (cons (car t1) a-u-args) phi1 t)
                       (anti-unify-injection (cons t1 t2) phi))))
            (t (anti-unify-injection (cons t1 t2) phi)))
    (cond  ((endp t1)
            (if (eql t1 t2) (mv t1 phi t) (mv nil nil nil)))
           ((endp t2) (mv nil nil nil))
           (t (mv-let (a-u-cdr phi1 bool1)
                      (anti-unify-aux nil (cdr t1) (cdr t2) phi)
                      (if bool1
                          (mv-let (a-u-car phi2 bool2)
                                  (anti-unify-aux
                                       t (car t1) (car t2) phi1)
                            (if bool2
                                (mv (cons a-u-car a-u-cdr) phi2 t)
                              (mv nil nil nil)))
                        (mv nil nil nil)))))))

(defun anti-unify (t1 t2)
  (mv-let (anti-unify phi bool)
          (anti-unify-aux t t1 t2 nil)
          anti-unify))
```

Figura 4.2: Algoritmo de anti–unificación

4.3.2 Anti–unificación y sus propiedades principales

En la figura 4.2 aparece la definición en ACL2 de la función `anti-unify`, que implementa un algoritmo de anti–unificación entre términos de primer orden.

Esta implementación está basada en la definición dada en 4.17. Aunque pueda parecer difícil definir en ACL2 una función ϕ inyectiva entre $T(\Sigma, X) \times T(\Sigma, X)$ y X, en realidad

para obtener la anti–unificación de dos términos, sólo es necesario disponer de tal función inyectiva definida en un conjunto finito de pares de términos: aquellos pares que surgen durante el proceso de anti–unificación. Por tanto, podemos representar la función inyectiva como una lista de asociación, que asigna pares de términos con variables. Para que el proceso sea más eficiente, esta función inyectiva ha de ser construida incrementalmente, a medida que se necesite.

En concreto, la función `anti-unify` se define a partir de una función auxiliar `anti--unify-aux`, que implementa la anti–unificación de manera recursiva en la estructura de los términos y por tanto lo hace tanto para términos como para listas de términos. Recibe como entrada dos términos o listas de términos `t1` y `t2`, un indicador `flg` y una lista de asociación `phi` que asocia pares punteados de términos con variables numéricas. Intuitivamente, este último argumento va a servir para construir la función inyectiva que se necesita para asignar variables a parejas de términos. Como es usual, si `flg`=`nil`, se interpreta que `t1` y `t2` son listas de términos; en caso contrario, se interpreta que son términos. La función `anti-unify-aux` devuelve tres valores a través de un multivalor. El primero de ellos es el término (o lista de términos) resultante de la anti–unificación, el segundo es la lista de asociación que representa a la función inyectiva que se ha necesitado (suponiendo que ya se parte de las asociaciones que aparecen en `phi`) y el tercero es `t` o `nil`, indicando si la anti–unificación ha tenido éxito o no. La función `anti-unify` recibe dos términos `t1` y `t2` como entrada y llama a `(anti-unify-aux t t1 t2 nil)`, para obtener, como veremos, un ínfimo de `t1` y `t2`.

La anti–unificación de dos términos siempre tiene éxito. Sin embargo, la anti–unificación de dos listas de términos puede fallar: por ejemplo, porque sean de distinta longitud. En ese caso existe ambigüedad entre la lista vacía de términos `nil` y el objeto `nil` como señal de fallo en la anti–unificación. Por este motivo, usamos multivalores en `anti-unify--aux` para devolver a la misma vez el indicador de éxito o de fallo y el término resultante.

Lo que sigue son algunos ejemplos de llamadas a la función `anti-unify` para calcular el ínfimo de dos términos:

```
ACL2 !>(anti-unify '(f (h y) x (h y)) '(f (g z) (g z) (g z)))
(F 1 2 1)
ACL2 !>(anti-unify '(f (h y) x (h y)) '(g (g z) (g z) (g z)))
1
ACL2 !>(anti-unify '(f (h (k u)) x (h y)) '(f (h u) (g z) (h z)))
(F (H 3) 2 (H 1))
```

La función `anti-unify-aux` sigue de manera aproximada la idea de la operación $\wedge_\phi$ de la definición 4.17. Si los dos términos tienen el mismo símbolo raíz y el mismo número de argumentos, entonces se toma ese símbolo común y se sigue el proceso recursivamente para los argumentos. En otro caso, se toma una variable correspondiente al par de términos que se anti–unifican. Esa variable asignada sólo se volverá a asignar si vuelve a surgir la misma pareja de términos durante el proceso de anti–unificación. La función auxiliar `anti-unify-injection` implementa la manera de asignar variables a pares de términos durante el proceso de anti–unificación, además de extender la lista de asociación que representa a la función inyectiva, cuando sea necesario. Esta función recibe un par de términos `p` una lista de asociación `phi`. Si el par de términos `p` se encuentra en el dominio de `phi`, entonces devuelve la variable que `p` tiene asignado en `phi`. Si no, le asigna una nueva

variable al par de términos: el último valor asignado más 1 (ó 1 si `phi` es vacía). Nótese que `anti-unify-injection` devuelve un multivalor consistente en la variable asignada, la lista de asociación (posiblemente extendida con una nueva asociación) y el valor booleano `t`.

A continuación presentamos las propiedades principales de la función `anti-unify`. Los dos teoremas siguientes formalizan el teorema 4.20, demostrando que la función calcula el ínfimo de dos términos. El primero de ellos muestra que obtiene una cota inferior, respecto de la relación de subsunción, de los dos términos que recibe como entrada. El segundo teorema establece que es cota superior de cualquier otra cota inferior de los dos términos que se reciben como entrada:

```
(defthm anti-unify-lower-bound
  (and (subs (anti-unify t1 t2) t1)
       (subs (anti-unify t1 t2) t2)))

(defthm anti-unify-greatest-lower-bound
   (implies (and (subs term t1)
                 (subs term t2))
            (subs term (anti-unify t1 t2))))
```

Además, el cálculo del ínfimo es una operación cerrada en el conjunto de los términos de una signatura dada:

```
(defthm anti-unify-term-s-p
  (implies (and (term-s-p t1) (term-s-p t2))
           (term-s-p (anti-unify t1 t2))))
```

4.3.3 Descripción de la demostración

A continuación describimos los principales lemas que permiten obtener una prueba en ACL2 de las propiedades de `anti-unify` presentadas anteriormente. Aunque la idea principal de la prueba está inspirada en la demostración del teorema 4.20, el hecho de que el algoritmo de anti–unificación implementado construya la función inyectiva ϕ durante el proceso de anti–unificación, hace que debamos abordar la prueba con una estrategia diferente.

Para ello, usaremos una técnica de *razonamiento compuesto*: definiremos previamente una versión simple, aunque menos eficiente, de `anti-unify-aux`, que llamaremos `pre-anti-unify-aux`. A continuación veremos que esta versión simplificada obtiene (bajo ciertas condiciones) el ínfimo de dos términos. Por último probaremos un teorema de equivalencia entre `anti-unify-aux` y `pre-anti-unify-aux`, lo cual nos permitirá concluir las propiedades principales de la función `anti-unify`. En lo que sigue detallamos todo el proceso.

La función `pre-anti-unify-aux`

La diferencia fundamental del algoritmo implementado por `anti-unify`, respecto de la definición dada en 4.17, es que en ésta se suponía la existencia de una función inyectiva ϕ

fijada de antemano. Por contra, la función `anti-unify-aux` combina el proceso de anti–unificación con la construcción incremental de la función inyectiva, a medida que se va necesitando. Esto complica considerablemente el razonamiento.

Podemos, sin embargo, definir una versión alternativa a `anti-unify-aux`, actuando *como si dispusiéramos* de la función inyectiva construida. La función `pre-anti-unify--aux` implementa esta idea:

```
(defun pre-anti-unify-aux (flg t1 t2 phi)
  (if flg
      (cond ((or (variable-p t1) (variable-p t2))
             (mv (cdr (assoc (cons t1 t2) phi)) t))
            ((eql (car t1) (car t2))
             (mv-let (anti-unify-args bool)
                     (pre-anti-unify-aux nil (cdr t1) (cdr t2) phi)
                     (if bool
                         (mv (cons (car t1) anti-unify-args) t)
                       (mv (cdr (assoc (cons t1 t2) phi)) t))))
            (t (mv (cdr (assoc (cons t1 t2) phi)) t)))
    (cond  ((endp t1) (if (eql t1 t2) (mv t1 t) (mv nil nil)))
           ((endp t2) (mv nil nil))
           (t (mv-let (anti-unify-cdr bool1)
                      (pre-anti-unify-aux nil (cdr t1) (cdr t2) phi)
                      (if bool1
                          (mv-let (anti-unify-car bool2)
                                  (pre-anti-unify-aux
                                     t (car t1) (car t2) phi)
                                  (if bool2
                                      (mv (cons anti-unify-car
                                                anti-unify-cdr)
                                          t)
                                    (mv nil nil)))
                        (mv nil nil)))))))
```

Como se observa, la función `pre-anti-unify-aux` es similar a la definición de `anti--unify-aux`, excepto en el tratamiento de su cuarto argumento `phi`. En este caso, la lista de asociación `phi` se supone constante a lo largo de todo el proceso. Por tanto, `pre-anti--unify-aux` devuelve un multivalor que sólo consta de dos valores: el término resultante del proceso y un valor booleano, indicando éxito o fallo.

Como veremos, `(pre-anti-unify-aux flg t1 t2 phi)` calcula un ínfimo de los términos (o lista de términos) `t1` y `t2`, siempre que `phi` verifique unas determinadas condiciones. Esta condiciones expresan que `phi` debe asignar, de manera unívoca, una variable a cada par de términos que aparezcan en el proceso de anti–unificación de `t1` y `t2`. La función `injection-p` formaliza estas condiciones:

```
(defun injection-p (phi flg t1 t2)
  (and (alistp phi)
       (setp (co-domain phi))
       (list-of-variables-p (co-domain phi))
```

```
      (mv-let (term phi1 bool)
              (anti-unify-aux flg t1 t2 nil)
              (subsetp (domain phi1) (domain phi))))))
```

Esta función comprueba que `phi` es una lista de asociación que representa a una función cuyo codominio es una lista de variables, que es inyectiva en su dominio (usando `setp` para comprobar que no existen repeticiones en su co-dominio) y cuyo dominio contiene a todos los pares de términos a los cuales hay que asignar una variable durante el proceso. Es interesante destacar que la lista de asociación construida por `anti-unify-aux` sirve en este caso para expresar los pares que deben estar incluidos en el dominio de `phi`.

A continuación comprobaremos que `pre-anti-unify-aux` calcula el ínfimo de dos términos, siempre que la lista de asociación que recibe como entrada verifique la propiedad `injection-p`.

Propiedades de `pre-anti-unify-aux`

Dada su estructura recursiva, resulta más sencillo (véase la subsección 3.1.3) establecer las propiedades de la función `pre-anti-unify-aux` tanto para términos como para listas de términos.

En primer lugar, `(pre-anti-unify-aux flg t1 t2)` obtiene una cota inferior de los términos (o lista de términos) que recibe como entrada. Centrémonos, por ejemplo, en la demostración de que el término obtenido subsume a `t1` (para `t2` el razonamiento es totalmente análogo). Siguiendo la demostración a mano dada en el teorema 4.20, definimos la función `subst-anti-unify-1`, que nos servirá para obtener la sustitución testigo de la subsunción que se pretende demostrar:

```
(defun subst-anti-unify-1 (phi)
  (if (endp phi)
      nil
    (cons (let* ((p (car phi))
                 (pair-of-terms (car p))
                 (variable (cdr p))
                 (s1 (car pair-of-terms)))
            (cons variable s1))
          (subst-anti-unify-1 (cdr phi))))))
```

Es fácil probar que, bajo ciertas condiciones sobre `phi`, la sustitución calculada por `(subst-anti-unify-1 phi)` representa a la primera proyección de la función inversa de `phi`, como establece el siguiente lema:

```
(defthm inverse-projection-1
  (implies (and (alistp phi)
                (setp (co-domain phi))
                (list-of-variables-p (co-domain phi))
                (member (cons t1 t2) (domain phi)))
           (equal (val (cdr (assoc (cons t1 t2) phi))
                       (subst-anti-unify-1 phi))
                  t1)))
```

Aplicando el esquema de inducción sugerido por `pre-anti-unify-aux`, podemos demostrar que `subst-anti-unify-1` obtiene la sustitución testigo que justifica la relación de subsunción entre el término calculado por `pre-anti-unify-aux` y `t1`. El siguiente teorema establece el resultado finalmente demostrado:

```
(defthm subsumes-pre-anti-unify-aux-1
  (let* ((glb (pre-anti-unify-aux flg t1 t2 phi))
         (sigma (subst-anti-unify-1 phi)))
    (implies (and (injection-p phi flg t1 t2)
                  (second glb))
             (equal (apply-subst flg sigma (first glb)) t1))))
```

Para la prueba por inducción de este teorema, como en la prueba a mano dada en 4.20, el lema `inverse-projection-1` presentado anteriormente es clave en la prueba del caso base. Para manejar el caso de las listas de términos, hemos de incluir entre las hipótesis del teorema que `pre-anti-unify-aux` termina con éxito.

De manera totalmente análoga, se define `subst-anti-unify-2` y se demuestra el siguiente teorema, que expresa que el término calculado subsume a `t2`:

```
(defthm subsumes-pre-anti-unify-aux-2
 (let* ((glb (pre-anti-unify-aux flg t1 t2 phi))
        (sigma (subst-anti-unify-2 phi)))
   (implies (and (injection-p phi flg t1 t2)
                 (second glb))
            (equal (apply-subst flg sigma (first glb)) t2)))))
```

Veamos ahora cómo demostramos que el término calculado por `pre-anti-unify-aux` es la mayor de las cotas inferiores de `t1` y `t2`. Para ello, supongamos que `term` es una cota inferior de `t1` y `t2`. Según el teorema de corrección de la relación de subsunción, estas relaciones vendrán justificada por dos sustituciones `sigma1` y `sigma2`[8], tales que aplicadas a `term` obtienen `t1` y `t2` respectivamente. Debemos probar que, en ese caso, `term` subsume al término obtenido por `pre-anti-unify-aux`. Para ello, hemos de construir una sustitución testigo de tal subsunción.

Podemos seguir la demostración a mano del teorema 4.20 y definir formalmente la operación $\wedge_\phi$ que, aplicada a `sigma1` y `sigma2`, va a servir para obtener la sustitución testigo de la subsunción entre `term` y `(first (pre-anti-unify-aux flg t1 t2 phi))`. La función `anti-unify-substitutions` implementa formalmente esta idea. Esta función construye una sustitución tal que a cada variable `x` de su dominio le asocia el resultado de la anti–unificación entre `(val x sigma1)` y `(val x sigma2)`. Existe un detalle técnico que hemos de solventar, ya que nuestra representación de una sustitución es mediante una lista de asociación y por tanto, con un dominio finito. Introducimos un argumento extra `l`, que representa el dominio de la sustitución construida. Nótese además que la función inyectiva `phi` respecto de la cual se realizan las anti–unificaciones debe ser incluida también como argumento de entrada:

```
(defun anti-unify-substitutions (sigma1 sigma2 l phi)
```

[8] Que a posteriori serán, respectivamente, `(matching term t1)` y `(matching term t2)`.

```
  (if (endp l)
     nil
    (cons
     (cons (car l)
           (mv-let (anti-unify bool)
                   (pre-anti-unify-aux
                    t (val (car l) sigma1) (val (car l) sigma2) phi)
                   anti-unify))
     (anti-unify-substitutions sigma1 sigma2 (cdr l) phi)))))
```

Podemos probar por inducción estructural (sugerida por `pre-anti-unify-aux`) que siempre que la lista `l` contenga a las variables de `term`, entonces la sustitución que calcula `anti-unify-substitutions` es testigo de la subsunción entre la cota inferior `term` y el término calculado por `pre-anti-unify-aux`. Es lo que establece el siguiente teorema:

```
(defthm pre-anti-unify-aux-greatest-lower-bound-main-lemma
  (let ((anti-unif-sigma
           (anti-unify-substitutions sigma1 sigma2 l phi))
        (anti-unif-term (first (pre-anti-unify-aux
                                flg
                                (apply-subst flg sigma1 term)
                                (apply-subst flg sigma2 term)
                                phi))))
    (implies (subsetp (variables flg term) l)
             (equal (apply-subst flg anti-unif-sigma term)
                    anti-unif-term)))))
```

Varios aspectos en la formulación de este teorema merecen ser destacados. En primer lugar, nótese que los términos `t1` y `t2` están aquí expresados como respectivas instancias de `term`. En segundo lugar, nótese que no es necesario incluir entre las hipótesis que la anti–unificación de `(apply-subst flg sigma1 term)` y `(apply-subst flg sigma2 term)` tiene éxito: en este caso el algoritmo nunca devuelve fallo, ya que existe al menos una cota inferior, que es `term`. Y por último, hemos de destacar el papel decisivo que juega en la demostración el hecho de haber usado `subsetp` en las hipótesis. De esta manera, se generaliza convenientemente el resultado, permitiendo una demostración más simple: no se necesita ningún lema previo específico para completar la prueba.

Por último, es posible demostrar la correspondiente propiedad de clausura de `pre--anti-unify-aux`. Si actúa sobre términos de una signatura determinada, devuelve un término en la misma signatura:

```
(defthm pre-anti-unify-aux-term-s-p-aux
  (let* ((glb (pre-anti-unify-aux flg t1 t2 phi)))
    (implies (and (injection-p phi flg t1 t2)
                  (alistp-acl2-numberp phi)
                  (second glb)
                  (term-s-p-aux flg t1)
                  (term-s-p-aux flg t2))
             (term-s-p-aux flg (first glb)))))
```

En el teorema debemos asumir las siguientes hipótesis sobre `phi`: que verifique la condición `injection-p` y que su codominio sea una lista de números (es lo que comprueba la función `alistp-acl2-numberp`). Esta última condición es necesaria: recuérdese que las variables de los términos en una signatura deben verificar el predicado `eqlablep`.

El puente entre `anti-unify-aux` y `pre-anti-unify-aux`

Los resultados que se acaban de presentar sobre `pre-anti-unify-aux` son justamente los que se pretenden demostrar sobre `anti-unify-aux`. Para poder trasladar tales propiedades, debemos demostrar que:

- La función `pre-anti-unify-aux` calcula el mismo término que `anti-unify-aux`, siempre que en el proceso se use una función inyectiva concreta (que a continuación definiremos).
- Tal función inyectiva verifica la propiedad `injection-p`, lo cual nos permitirá hacer uso de los teoremas anteriores.

Para demostrar la primera de las afirmaciones, pensemos primero en qué propiedades debemos exigir a la lista de asociación que recibe como argumento la función `pre-anti--unify-aux`, para poder concluir que el término que obtiene es el mismo que el obtenido por `anti-unify-aux`. Dado `(anti-unify-aux flg t1 t2 phi)`, queremos encontrar una lista de asociación `phi1`, tal que `(pre-anti-unify-aux flg t1 t2 phi1)` obtenga el mismo término. Como veremos, la lista de asociación `phi1` que buscamos es precisamente la que finalmente construye `(anti-unify-aux flg t1 t2 phi)` y que devuelve como segundo valor. Para demostrarlo, nuevamente será más sencillo generalizar el resultado, como veremos a continuación.

El predicado `(extension-assoc phi1 phi)` se verifica si `phi1` es una lista de asociación que puede expresarse como `(append phi2 phi)` y los elementos del dominio de `phi2` no están en el dominio de `phi`:

```
(defun extension-assoc (phi1 phi)
  (cond ((equal phi1 phi) t)
        ((endp phi1) nil)
        (t (and (not (assoc (caar phi1) phi))
                (extension-assoc (cdr phi1) phi)))))
```

Usando este concepto de extensión, podemos formular una generalización conveniente de la propiedad que establece la equivalencia entre `anti-unify-aux` y `pre-anti-unify--aux`:

```
(defthm anti-unify-aux-pre-anti-unify-aux-main-lemma
  (implies (extension-assoc phi1 (second (anti-unify-aux flg t1 t2 phi)))
           (equal (first (anti-unify-aux flg t1 t2 phi))
                  (first (pre-anti-unify-aux flg t1 t2 phi1)))))
```

Es decir si `phi1` es una lista de asociación que extiende a la calculada por `(anti--unify-aux flg t1 t2 phi)`, entonces el término que calcula esta llamada, es igual que el calculado por `(pre-anti-unify-aux flg t1 t2 phi1)`. La prueba de este resultado

se tiene por inducción, mediante un esquema sugerido por `anti-unify-aux`. Hemos de enfatizar nuevamente el hecho de que esta generalización usando `extension-assoc` es fundamental para obtener la prueba, de manera análoga a lo que ocurre con `subsetp` (véase la subsección C.2.1 del apéndice C). Como caso particular de esta generalización se tiene el resultado buscado:

```
(defthm anti-unify-aux-pre-anti-unify-aux
  (equal (first (anti-unify-aux flg t1 t2 phi))
         (first (pre-anti-unify-aux
                 flg t1 t2
                 (second (anti-unify-aux flg t1 t2 phi))))))
```

Como hemos dicho al principio, para poder usar las propiedades ya probadas (sobre `pre-anti-unify-aux`), debemos probar ahora que la lista de asociación `(second (anti--unify-aux flg t1 t2 phi))` verifica la propiedad `injection-p`:

```
(defthm anti-unify-aux-injection-injection-p
  (implies (and (alistp phi)
                (acl2-numberp-increasing-list (co-domain phi)))
           (injection-p
             (second (anti-unify-aux flg t1 t2 phi)) flg t1 t2))))
```

Nótese que hemos de exigir que inicialmente la lista de asociación de partida sea una lista cuyo codominio es una lista decreciente de números (esto no es problema, ya que para la definición de `anti-unify` la lista de asociación de partida será `nil`).

Nos referiremos a los dos teoremas anteriores como los *teoremas puente*. Finalmente, para poder usar el teorema de clausura sobre `pre-anti-unify-aux` se prueba la propiedad `alistp-acl2-numberp` para la lista de asociación construida:

```
(defthm anti-unify-aux-injection-alistp-acl2-numberp
  (implies (alistp-acl2-numberp phi)
           (alistp-acl2-numberp
            (second (anti-unify-aux flg t1 t2 phi)))))
```

Las propiedades principales de `anti-unify`

Una vez probadas las propiedades sobre `pre-anti-unify-aux` y los teoremas puente entre `anti-unify-aux` y `pre-anti-unify-aux`, es posible obtener de manera sencilla las propiedades sobre `anti-unify`:

- Los teoremas `subsumes-pre-anti-unify-aux-1`, `subsumes-pre-anti-unify-aux--2`, junto con el teorema de completitud de la subsunción, permiten demostrar que `pre-anti-aux` calcula una cota inferior de `t1` y `t2`. Los teoremas puente sirven para poder probar lo mismo sobre `anti-unify-aux` y por tanto sobre `anti-unify` (para `flg=t` y `phi=nil`): eso es justamente el teorema `anti-unify-lower-bound`.
- Instanciando el teorema `pre-anti-unify-aux-greatest-lower-bound-main-lemma` (para `l=(variables flg term)`) y a partir de los teoremas de corrección y completitud de la relación de subsunción, podemos establecer que el término que

calcula `pre-anti-unify-aux` es subsumido por cualquier generalización común de `t1` y `t2`. Nuevamente, los teoremas puente sirven para poder probar lo mismo sobre `anti-unify-aux` y por tanto sobre `anti-unify`, permitiendo concluir el teorema `anti-unify-greatest-lower-bound`.

- Por último, y análogamente, el teorema `pre-anti-unify-aux-term-s-p-aux`, los teoremas puente y el teorema `anti-unify-aux-injection-alistp-acl2-numberp`, nos llevan a concluir la propiedad de clausura de `anti-unify`, teorema `anti-unify--term-s-p`.

4.3.4 La estrategia del razonamiento compuesto

Una vez explicada la demostración de las propiedades del algoritmo de anti–unificación, no hay nada más que reseñar sobre la demostración llevada a cabo usando el demostrador ACL2: se trata de una interacción típica, mediante la cual el usuario suministra los lemas necesarios para llegar a los resultados buscados. Además de destacar la importancia que tiene el generalizar adecuadamente la formulación de los teoremas (como ya se ha comentado), el aspecto más relevante en las pruebas presentadas es el uso de una estrategia de razonamiento compuesto.

En un principio, habíamos intentado la demostración de las propiedades razonando directamente sobre la definición de `anti-unify-aux`. Sin embargo, la demostración automática resultó compleja. El problema estriba en razonar sobre la función inyectiva, ya que ésta se construye de manera incremental. En concreto, razonando directamente sobre `anti-unify-aux` no fuimos capaces de probar el teorema `anti-unify-greatest--lower-bound`. Es por esto que adoptamos una estrategia de refinamiento sucesivo del razonamiento.

Ésta es una estrategia general (véase [47, 70]), que a veces resulta de utilidad en la verificación formal, como ha demostrado este ejemplo:

- Dada una función sobre la cual queremos razonar, definimos una versión más simple de la misma. Probablemente, esta simplificación será a costa de sacrificar algunas cualidades de la función original, como por ejemplo la eficiencia. Sin embargo, esta versión más simple conserva las propiedades esenciales de la original. Es el caso de la función `pre-anti-unify-aux`, como versión simplificada de `anti-unify-aux`.

- Se prueban las propiedades principales de la versión simplificada de la función. Es de esperar que estas propiedades sean más sencillas de probar que para la función original. En nuestro caso, se demuestra que bajo ciertas condiciones `pre-anti--unify-aux` calcula un ínfimo, respecto de la relación de subsunción, de los términos que recibe como entrada.

- Se demuestran una serie de teoremas puente que permiten establecer las condiciones bajo las cuales la versión simplificada es equivalente a la original. En este caso, hemos encontrado una lista de asociación concreta para la cual `pre-anti-unify--aux` y `anti-unify-aux` obtienen el mismo valor. Además hemos probado que tal lista de asociación verifica las propiedades necesarias para poder concluir que `pre--anti-unify-aux` calcula un ínfimo.

- Los teoremas puente nos permiten trasladar de manera inmediata las propiedades que se han probado, desde la versión simplificada a la versión original.

Este estrategia se puede aplicar para refinar progresivamente la función que se quiere verificar. Es lo que se conoce como *razonamiento compuesto*. Como hemos visto, la verificación de `anti-unify` es un buen ejemplo de tal método.

Nótese que la verificación de la función `number-rename` de la sección 4.1.3 podría haberse hecho usando una técnica parecida. Una versión simplificada de la función podría renombrar el término suponiendo que se dispone de una sustitución de renombrado previamente calculada. Hemos preferido dejar la demostración tal y como se ha presentado: razonando directamente sobre la definición original. Además del interés en sí mismo de las técnicas empleadas en ella, supone un buen ejemplo para contrastar ambas aproximaciones.

4.4 Un algoritmo de unificación basado en reglas de transformación

Antes de probar la existencia de supremo para cada par de términos que tengan instancias comunes, será necesario definir y verificar formalmente un algoritmo de unificación. Al igual que con el algoritmo de equiparación de términos, este algoritmo estará especificado a partir un conjunto de reglas de transformación. Por ese motivo, la presentación de esta sección sigue la misma línea argumental que la de la sección 3.3. Los eventos ACL2 que conducen a las definiciones y teoremas aquí presentados se encuentran en el libro `unification-pattern.lisp`. Además, los resultados de la subsección 4.4.5 se encuentran en el libro `unification.lisp`.

4.4.1 Preliminares

Definición 4.21 *Sea S un sistema de ecuaciones (en $T(\Sigma, X)$) y σ una sustitución (en $T(\Sigma, X)$). Decimos que σ es un* **unificador** *(o una* **solución***) de S, si para cualquier $s \approx t \in S$, se verifica que $\sigma(s) = \sigma(t)$. Decimos que σ es un* **unificador** *(o una* **solución***)* **de máxima generalidad** *de S, si σ es un unificador de S y para todo σ' que sea unificador de S se tiene que $\sigma \preccurlyeq \sigma'$. Un sistema de ecuaciones se dirá* **unificable** *si tiene un unificador.*

Podemos particularizar estas definiciones para el caso de pares de términos.

Definición 4.22 *Sean $s, t \in T(\Sigma, X)$ y σ sustitución en $T(\Sigma, X)$. Decimos que σ es un* **unificador (de máxima generalidad)** *de s y t, si es un unificador (de máxima generalidad) del sistema $\{s \approx t\}$. Decimos que s y t son* **unificables** *si lo es el sistema $\{s \approx t\}$.*

Existen determinados sistemas de ecuaciones para los cuales es muy fácil encontrar un unificador de máxima generalidad, como veremos más adelante. La siguiente definición identifica tales sistemas.

Definición 4.23 *Sea $S = \{x_1 \approx t_1, \ldots, x_n \approx t_n\}$ un sistema de ecuaciones en $T(\Sigma, X)$. Decimos que S es un* **sistema en forma resuelta** *si es un sistema–sustitución y $x_i \notin \mathcal{V}(t_j)$ para $i, j \in \{1, \ldots, n\}$*

Los sistemas en forma resuelta van a dar lugar a un tipo específico de sustituciones, que definimos a continuación:

Definición 4.24 *Sea σ una sustitución en $T(\Sigma, X)$. Decimos que σ es una sustitución* **idempotente** *si $\sigma\sigma = \sigma$.*

Existe una manera de caracterizar sintácticamente a las sustituciones idempotentes, como expresa la siguiente proposición.

Proposición 4.25 *Una sustitución σ es idempotente si y sólo si ninguna variable del dominio de σ es una variable de algún término del codominio de σ.*

Demostración:
Sea σ una sustitución idempotente. Supongamos que existen $x, y \in \mathcal{D}(\sigma)$ tal que $y \in \mathcal{V}(\sigma(x))$. Entonces está claro que $\sigma(\sigma(x)) \neq \sigma(x)$, ya que $\sigma(y) \neq y$.

Para la implicación contraria, supongamos que ninguna variable del dominio de σ es una variable de algún término del codominio de σ. Si $x \in \mathcal{D}(\sigma)$, entonces $\mathcal{V}(\sigma(x)) \cap \mathcal{D}(\sigma) = \emptyset$ y por tanto $\sigma(\sigma(x)) = \sigma(x)$. Si $x \notin \mathcal{D}(\sigma)$, entonces claramente $\sigma(\sigma(x)) = \sigma(x)$. □

Los siguientes lema y teorema expresan la relación existente entre los sistemas en forma resuelta y las sustituciones idempotentes. Además establecen cómo obtener un unificador de máxima generalidad para sistemas en forma resuelta.

Lema 4.26 *Sea S un sistema de ecuaciones en forma resuelta y σ un unificador de S. Entonces $\sigma = \sigma\vec{S}$.*

Demostración:
Sea $S = \{x_1 \approx t_1, \ldots, x_n \approx t_n\}$ y σ unificador de S. Probemos que $\sigma(x) = \sigma\vec{S}(x)$, para cualquier $x \in X$. Si $x = x_k \in \{x_1, \ldots, x_n\}$, entonces $\sigma(x) = \sigma(x_k) = \sigma(t_k) = \sigma(\vec{S}(x_k)) = \sigma(\vec{S}(x))$. Si $x \notin \{x_1, \ldots, x_n\}$, entonces $\sigma(x) = \sigma(\vec{S}(x))$, ya que que $\vec{S}(x) = x$. □

Teorema 4.27 *Sea S un sistema de ecuaciones en forma resuelta. Entonces $\vec{S}$ es un unificador de máxima generalidad e idempotente de S.*

Demostración:
Sea $S = \{x_1 \approx t_1, \ldots, x_n \approx t_n\}$ y sea $i \in \{1, \ldots, n\}$. Es claro que $\vec{S}(x_i) = t_i$, ya que S es un sistema sustitución. Por otro lado, como $\mathcal{D}(\vec{S}) \cap \mathcal{V}(t_i) = \emptyset$, entonces $\vec{S}(t_i) = t_i$. Por tanto, $\vec{S}$ es un unificador de S. Según el lema 4.26, si σ es un unificador de S, tenemos que $\sigma\vec{S} = \sigma$, luego $\vec{S} \preccurlyeq \sigma$ y por tanto $\vec{S}$ es unificador de máxima generalidad de S. La idempotencia de $\vec{S}$ se tiene nuevamente por el lema 4.26, ya que al ser $\vec{S}$ unificador de S, en particular se tiene que $\vec{S}\vec{S} = \vec{S}$. □

Es posible diseñar un algoritmo de unificación basado en este último teorema, análogamente a lo que se hizo para el algoritmo de equiparación en la sección 3.3.1. La idea principal es que se pueden transformar los sistemas de ecuaciones, de manera que se conserve el conjunto de unificadores y de tal manera que las transformaciones conduzcan o bien hacia sistemas en forma resuelta (de los cuales es inmediato extraer un unificador de máxima generalidad), o bien se detecte la no unificabilidad.

En la figura 4.3 se presenta un conjunto de reglas de transformación que definen una relación $\Rightarrow_u$. Este conjunto de transformaciones es esencialmente el dado por Martelli y Montanari en [48]. Las transformaciones actúan sobre pares de sistemas de ecuaciones.

Intuitivamente, el primero de estos sistemas contiene las ecuaciones pendientes de unificar y el segundo contiene el unificador parcialmente calculado. El símbolo $\bot$ indica que se ha detectado fallo en la unificación. Las transformaciones están diseñadas para que comenzando en el par de sistemas $S;\emptyset$ y aplicando repetidas veces los pasos de transformación, obtengamos finalmente un par de sistemas $\emptyset;T$, siendo T un sistema en forma resuelta (y por tanto unificable) o bien $\bot$ (y por tanto no unificable). Usando estas reglas de transformación, podemos fácilmente diseñar y verificar un algoritmo de unificación, como veremos.

Borra:	$\{t \approx t\} \cup R; T$	$\Rightarrow_u R; T$
Chequea:	$\{x \approx t\} \cup R; T$	$\Rightarrow_u \bot$ si $x \in \mathcal{V}(t)$ y $x \neq t$
Elimina:	$\{x \approx t\} \cup R; T$	$\Rightarrow_u \theta(R); \{x \approx t\} \cup \theta(T)$ si $x \in X$, $x \notin \mathcal{V}(t)$ y $\theta = \{x \mapsto t\}$
Descomp:	$\{f(s_1, \ldots, s_n) \approx f(t_1, \ldots, t_n)\} \cup R; T$	$\Rightarrow_u \{s_1 \approx t_1, \ldots, s_n \approx t_n\} \cup R; T$
Conflicto:	$\{f(s_1, \ldots, s_n) \approx g(t_1, \ldots, t_m)\} \cup R; T$	$\Rightarrow_s \bot$, si $n \neq m$ ó $f \neq g$
Orienta:	$\{t \approx x\} \cup R; T$	$\Rightarrow_u \{x \approx t\} \cup R; T$ si $x \in X$, $t \notin X$

Figura 4.3: Reglas de transformación para la unificación

Los siguientes ejemplos muestran cómo una aplicación exhaustiva de las reglas de $\Rightarrow_u$ nos permite decidir si un sistema es unificable o no y en caso afirmativo, encontrar un unificador de máxima generalidad.

Ejemplo 4.28 La siguiente secuencia de transformaciones obtiene el unificador de máxima generalidad $\{x \mapsto h(g(a), z), y \mapsto g(a), v \mapsto a\}$ para el par de términos $f(x, g(v), a)$ y $f(h(y, z), y, v)$.

$$
\begin{array}{ll}
 & \{f(x, g(v), a) \approx f(h(y, z), y, v)\}; \emptyset \\
\Rightarrow_u & \{x \approx h(y, z), y \approx g(v), v \approx a\}; \emptyset \qquad (\textbf{Descomp}) \\
\Rightarrow_u & \{x \approx h(g(v), z), v \approx a\}; \{y \approx g(v)\} \qquad (\textbf{Elimina}) \\
\Rightarrow_u & \{x \approx h(g(a), z)\}; \{y \approx g(a), v \approx a\} \qquad (\textbf{Elimina}) \\
\Rightarrow_u & \emptyset; \{x \approx h(g(a), z), y \approx g(a), v \approx a\} \qquad (\textbf{Elimina})
\end{array}
$$

La siguiente secuencia nos sirve para detectar que los términos $f(x, y)$ y $f(y, h(x))$ no son unificables.

$$
\begin{array}{ll}
 & \{f(x, y) \approx f(y, h(x))\}; \emptyset \\
\Rightarrow_u & \{x \approx y, y \approx h(x)\}; \emptyset \qquad (\textbf{Descomp}) \\
\Rightarrow_u & \{y \approx h(y)\}; \{x \approx y\} \qquad (\textbf{Elimina}) \\
\Rightarrow_u & \bot \qquad (\textbf{Chequea})
\end{array}
$$

Los siguientes lemas establecen algunas propiedades de la relación $\Rightarrow_u$.

Lema 4.29 *Sean S, S', T y T' sistemas de ecuaciones en $T(\Sigma, X)$, tales que $S;T \Rightarrow_u S';T'$. Entonces:*

a) *σ es unificador de $S \cup T$ si y sólo si es unificador de $S' \cup T'$.*

b) *Si T es un sistema en forma resuelta y $\mathcal{D}(\vec{T}) \cap \mathcal{V}(S) = \emptyset$, entonces T' está en forma resuelta y $\mathcal{D}(\vec{T'}) \cap \mathcal{V}(S') = \emptyset$.*

Demostración:

Para las reglas **Borra**, **Descomp** y **Orienta**, ambos apartados son triviales. Veamos con más detalle la regla **Elimina** (el resto de reglas devuelven fallo). Por tanto, supongamos que tenemos un par de sistemas $S;T$, tal que $S = \{x \approx t\} \cup R$ (donde $x \in X$ y $x \notin \mathcal{V}(t)$), que se transforman en un par de sistemas $S';T'$, donde $S' = \theta(R)$ y $T' = \{x \approx t\} \cup \theta(T)$, donde $\theta = \{x \mapsto t\}$.

Para probar el apartado a), nótese que por el lema 4.26 aplicado al sistema $\{x \approx t\}$, se tiene que $\sigma\theta = \sigma$ si $\sigma(x) = \sigma(t)$. Además, se tiene que para cualquier sistema U, σ es unificador de $\theta(U)$ si y sólo si $\sigma\theta$ es unificador de U.

De todo lo cual se deduce que σ es unificador de $S \cup T = \{x \approx t\} \cup R \cup T$ si y sólo si σ es unificador de $S' \cup T' = \theta(R) \cup \{x \approx t\} \cup \theta(T)$.

Para la prueba del apartado b) (regla **Elimina**), supongamos que T está en forma resuelta y que $\mathcal{D}(\vec{T}) \cap (\mathcal{V}(R) \cup \{x\} \cup \mathcal{V}(t)) = \emptyset$. Nótese que como $x \notin \mathcal{V}(t)$, la variable x no aparece en los sistemas $\theta(R)$ y $\theta(T)$. Esto y el hecho de que $\mathcal{V}(t)$ y $\mathcal{D}(\vec{T})$ son conjuntos disjuntos, hace que T' esté en forma resuelta. Además, como $\mathcal{V}(R)$ y $\mathcal{D}(\vec{T})$ son disjuntos, $\mathcal{V}(\theta(R))$ y $\{x\} \cup \mathcal{D}(\vec{T}) = \mathcal{D}(\vec{T'})$ son disjuntos. □

Lema 4.30 *Sean S y T sistemas de ecuaciones en $T(\Sigma, X)$, tales que $S;T \Rightarrow_u \bot$. Entonces $S \cup T$ no es unificable.*

Demostración:
Para las regla **Conflicto**, el resultado es evidente. Veamos con más detalle el resultado para la regla **Chequea** (el resto de reglas no devuelven fallo). Para ello, basta ver que los sistemas de la forma $\{x \approx t\}$, con $x \in \mathcal{V}(t)$ y $x \neq t$, no son unificables. Y esto se tiene porque en ese caso, para cualquier sustitución σ, $|\sigma(x)| < |\sigma(t)|$ (lo cual se puede probar por una sencilla inducción en la estructura de t). □

El conjunto de reglas de $\Rightarrow_u$ está diseñado para que exista una regla aplicable para cualquier tipo de ecuación seleccionada en el sistema S. Por tanto, se tiene trivialmente la siguiente propiedad sobre los pares de sistemas que están en forma normal respecto de $\Rightarrow_u$.

Lema 4.31 *Sean S y T sistemas de ecuaciones en $T(\Sigma, X)$, tales que el par $S;T$ está en forma normal respecto de $\Rightarrow_u$. Entonces $S = \emptyset$.*

A continuación definimos un algoritmo de unificación de sistemas de ecuaciones, basado en el conjunto de reglas de transformación $\Rightarrow_u$:

$unifica(S) =$
  $T := \emptyset$
  mientras $S \neq \emptyset$ **hacer**
    si $S;T \Rightarrow_u \bot$, **parar** *y* **devolver** *FALLO*
    si no, *sea* $S';T'$ *tal que* $S;T \Rightarrow_u S';T'$ *y* **hacer** $S := S'$ *y* $T := T'$
  fin-mientras
  devolver $\vec{T}$

Nótese que el algoritmo es *no determinista* ya que no se determina qué regla se aplica en los sucesivos pasos de transformación. Es decir, si $S;T \Rightarrow_u S';T'$ y $S;T \Rightarrow_u S'';T''$, usando distintas reglas de transformación, el algoritmo puede escoger arbitrariamente cualquiera de ellas.

Los siguientes teoremas establecen la terminación del algoritmo $unifica$ y la corrección y completitud del mismo.

Teorema 4.32 *$unifica(S)$ termina para cualquier sistema de ecuaciones S.*

Demostración:
Para probar la terminación de $unifica$, bastará ver que no es posible obtener una secuencia infinita de transformaciones con $\Rightarrow_u$. Para ello, asignamos a cada sistema S de un par de sistemas $S;T$, una medida $M(S)$ en $\mathbb{N}\times\mathbb{N}\times\mathbb{N}$ y probaremos que si $S;T \Rightarrow_u S';T'$, entonces $M(S) >_{lex} M(S')$, donde $>_{lex}$ es el orden lexicográfico en $\mathbb{N}\times\mathbb{N}\times\mathbb{N}$ inducido por el orden usual en $\mathbb{N}$. Nótese por tanto que nos podemos restringir a las reglas que no devuelven $\bot$. La medida M se define de la siguiente manera: $M(S) = (M_1(S), M_2(S), M_3(S))$, donde $M_1(S) = |\mathcal{V}(S)|$ (el número de variables distintas de S), $M_2(S) = t(S)$ (el número de símbolos de función de S) y $M_3(S)$ es número de ecuaciones de S cuyo lado derecho es una variable. De la buena fundamentación de $>_{lex}$ (teorema A.15), se seguirá entonces la terminación de $unifica$.

Si aplicamos la regla **Borra**, se tiene que $M_1(S) \geq M_1(S')$ y $M_2(S) > M_2(S')$. Si aplicamos la regla **Descomp**, entonces $M_1(S) = M_1(S')$ y $M_2(S) > M_2(S')$. Para la regla **Orienta**, se cumple que $M_1(S) = M_1(S')$, $M_2(S) = M_2(S')$ y $M_3(S) > M_3(S')$. Finalmente, nótese que la regla **Elimina** efectivamente elimina de S una variable, por lo que $M_1(S) > M_1(S')$. Por la definición de $>_{lex}$, se tiene que $M(S) >_{lex} M(S')$ en cualquier caso. □

Teorema 4.33 *(**Corrección y completitud del algoritmo de unificación**) Sea S un sistema en $T(\Sigma, X)$. Entonces S es unificable si y sólo si $unifica(S)$ no devuelve FALLO. En tal caso, $unifica(S)$ devuelve un unificador de máxima generalidad e idempotente de S.*

Demostración:
Para probar la corrección, supongamos que $unifica(S)$ no devuelve fallo. Entonces existe una $\Rightarrow_u$-derivación de la forma $S;\emptyset \Rightarrow_u \ldots \Rightarrow_u \emptyset;T$ y el algoritmo devuelve $\vec{T}$. Por el lema 4.29, apartado b), se tiene que T es un sistema en forma resuelta y por tanto T es equiparable y $\vec{T}$ es un unificador de máxima generalidad e idempotente de T (lema 4.27). Por el lema 4.29, apartado a), S es unificable y $\vec{T}$ un unificador de máxima generalidad e idempotente de S.

Para probar la completitud, supongamos ahora que S es unificable. Entonces el algoritmo no devuelve $FALLO$, porque en tal caso existiría una $\Rightarrow_u$-derivación de la forma $S;\emptyset \Rightarrow_u \ldots \Rightarrow_u \bot$. Pero esto está en contradicción con los lemas 4.29, apartado a) y 4.30. □

Como caso particular del algoritmo de unificación de sistemas de ecuaciones, podemos definir un algoritmo de unificación de pares de términos. Los siguientes teorema y corolario son consecuencia inmediata de 4.33.

Teorema 4.34 *Dos términos s y t son unificables si y sólo si $unifica(\{s \approx t\})$ no devuelve FALLO. En ese caso, devuelve un unificador de máxima generalidad e idempotente de s y t.*

Corolario 4.35 *Sean $s, t \in T(\Sigma, X)$. Entonces s y t son unificables si y sólo si existe un unificador de máxima generalidad de s y t.*

4.4.2 Definición de un algoritmo de unificación no determinista

En esta subsección presentamos la definición formal del algoritmo de unificación de Martelli y Montanari, basado en el conjunto de reglas de transformación que define la relación $\Rightarrow_u$ de la figura 4.3.

Función de selección y no determinismo

Como en el caso del algoritmo de equiparación, debemos formalizar el no determinismo que existe en la aplicación de las reglas de $\Rightarrow_u$. La regla de transformación que se aplica viene completamente determinada por la ecuación a la cual se le aplica. Por tanto, el no determinismo proviene de la posibilidad de elegir cualquier ecuación del sistema de ecuaciones. Para formalizar este comportamiento definimos parcialmente, usando un encapsulado, una función `sel`, asumiendo únicamente que tiene la propiedad de seleccionar una ecuación de cada conjunto de ecuaciones no vacío:

```
(encapsulate

 ((sel (lst) t))
 ...
 (defthm sel-selects
   (implies (consp l) (member (sel l) l))))
```

Esta definición parcial de la función de selección permite una mayor generalidad en el razonamiento, ya que no se asume ninguna estrategia concreta de aplicación de las reglas de transformación. Sin embargo, esto hace que el algoritmo de unificación que definiremos a continuación no sea ejecutable. Más adelante (subsección 4.4.5) usaremos una instancia concreta de la función de selección para definir (y verificar) un algoritmo de unificación ejecutable.

Definición de las reglas de transformación: unificación de sistemas

Dado un par de sistemas de ecuaciones, la función `transform-mm-sel` aplica un paso de transformación correspondiente a la relación $\Rightarrow_u$. Los pares de sistemas se representan mediante pares punteados y el sistema correspondiente al fallo, $\perp$, se representa por `nil`.

```
(defun transform-mm-sel (S-sol)
  (let* ((S (first S-sol)) (sol (cdr S-sol))
         (ecu (sel S))
         (t1 (car ecu)) (t2 (cdr ecu)) (R (eliminate ecu S)))
    (cond ((equal t1 t2) (cons R sol))                  ;;; *BORRA*
          ((variable-p t1)
           (if (member t1 (variables t t2))
               nil                                      ;;; *CHEQUEA*
             (cons                                      ;;; *ELIMINA*
              (apply-syst (list ecu) R)
              (cons ecu (apply-range (list ecu) sol)))))
          ((variable-p t2)
           (cons (cons (cons t2 t1) R) sol))            ;;; *ORIENTA*
          ((not (equal (car t1) (car t2))) nil)         ;;; *CONFLICTO*
          (t (mv-let (pair-args bool)
                     (pair-args (cdr t1) (cdr t2))
                   (if bool
                       (cons (append pair-args R) sol) ;;; *DESCOMP*
                     nil))))))                          ;;; *CONFLICTO*
```

Esta función recibe un par de sistemas de ecuaciones `S-sol`, selecciona (mediante `sel`) una ecuación del primero de este par de sistemas y en función de la forma de la ecuación seleccionada, aplica una de las reglas de $\Rightarrow_u$. Nótese el uso de la función `pair--args` para implementar las reglas **Conflicto** y **Descomp**. La implementación de la regla **Elimina** usa las funciones auxiliares `apply-syst` y `apply-range`. Dado un sistema S y una sustitución θ, la función `apply-syst` obtiene el sistema $\theta(S)$. Si T es un sistema tal que las variables que aparecen en los lados izquierdos de sus ecuaciones no aparecen en el dominio de θ[9], entonces la función `apply-range` obtiene el sistema $\theta(T)$.

```
(defun apply-syst (sigma S)
  (if (endp S)
      nil
    (cons (cons (apply-subst t sigma (caar S))
                (apply-subst t sigma (cdar S)))
          (apply-syst sigma (cdr S)))))

(defun apply-range (sigma S)
  (if (endp S)
      nil
    (cons (cons (caar S) (apply-subst t sigma (cdar S)))
          (apply-range sigma (cdr S)))))
```

[9] Nótese que éste es el caso cuando se aplica la regla **Elimina**.

Podemos ahora definir un función que aplique iterativamente las reglas de transformación definidas por `transform-mm-sel`. Estas aplicación sucesiva de las reglas ha de repetirse hasta que se obtenga un par de sistemas en el que el primero de ellos está vacío, o bien hasta que se detecte fallo. La condición de parada es exactamente la misma que en el caso del algoritmo de equiparación y viene implementada por la función `normal-form--syst` (sección 3.3.2). La función `solve-system-sel` implementa esta aplicación iterativa de las reglas de transformación:

```
(defun solve-system-sel (S-sol)
  (declare (xargs :measure (unification-measure S-sol)))
  (if (normal-form-syst S-sol)
      S-sol
    (solve-system-sel (transform-mm-sel S-sol))))
```

Como se observa, la admisión de esta función necesita que se proporcione explícitamente una función ordinal `unification-measure` que decrece en cada iteración, justificando así su terminación. Más adelante detallaremos esta cuestión.

La siguiente función `mgs-sel` implementa un procedimiento que encuentra, siempre que exista, una solución de máxima generalidad del sistema de ecuaciones que recibe como entrada. Para ello, dado un sistema de ecuaciones `S`, llama a la función `solve--system-sel` sobre el par de sistemas `(cons S nil)`. Si esta llamada termina con éxito, se devuelve la lista cuyo único elemento es el unificador calculado (el segundo del par de sistemas finalmente devuelto por `solve-system-sel`). En caso contrario, se devuelve `nil`.

```
(defun mgs-sel (S)
  (let ((solve-system-sel (solve-system-sel (cons S nil))))
    (if solve-system-sel (list (cdr solve-system-sel)) nil)))
```

Nótese que la función `mgs-sel` define formalmente el algoritmo *unifica* presentado en 4.4.1. Diremos que `mgs-sel` falla si devuelve `nil`. Como en el caso de la equiparación, el artificio de incluir la sustitución calculada dentro de una lista, está justificado por la necesidad de distinguir entre `nil` como representación de la sustitución identidad y `nil` como señal de fallo en la resolución del sistema.

Finalmente y como un caso particular, definimos las funciones `unifiable-sel` y `mgu--sel` que usan la función anterior para definir un algoritmo (no determinista) de unificación de pares de términos:

```
(defun unifiable-sel (t1 t2)
  (mgs-sel (list (cons t1 t2))))

(defun mgu-sel (t1 t2)
  (first (unifiable-sel t1 t2)))
```

El predicado `unifiable-sel` comprueba si dos términos son unificables y en ese caso, `mgu-sel` calcula un unificador de máxima generalidad. Estas dos funciones nos servirán para formular los teoremas que a continuación se presentan.

4.4.3 Propiedades principales

En esta sección presentamos las propiedades del algoritmo de unificación definido. Antes de eso, damos la definición del predicado `solution`, que formaliza el concepto de solución (o unificador) de un sistema de ecuaciones, tal y como se describe en la definición 4.21:

```
(defun solution (sigma S)
  (if (endp S)
      t
    (and (equal (apply-subst t sigma (caar S))
                (apply-subst t sigma (cdar S)))
         (solution sigma (cdr S)))))
```

Necesitaremos, además, definir formalmente el concepto de sustitución idempotente, definido en 4.24. Esta definición es difícil de formalizar en la lógica de ACL2, ya que expresa la igualdad funcional de dos sustituciones. En su lugar, el teorema de caracterización de la idempotencia dado en 4.25 nos sirve de base para nuestra definición:

```
(defun idempotent (S)
  (and (system-substitution S)
       (disjointp (variables nil (co-domain S)) (domain S))))
```

Más adelante mostraremos la propiedad principal que tienen aquellas sustituciones que verifican el predicado `idempotent`. Es interesante destacar que esta definición también es aplicable a sistemas de ecuaciones. Desde el punto de vista de los sistemas de ecuaciones, el predicado `idempotent` implementa la definición de *sistema en forma resuelta*, definición 4.23. Sacaremos provecho de esta dualidad más adelante.

Una vez presentadas estas dos definiciones, podemos formular las principales propiedades del algoritmo de unificación definido por `mgs-sel`, formalizando así el teorema 4.33. En primer lugar, los siguientes teoremas expresan la corrección y completitud del algoritmo de unificación de sistemas de ecuaciones. La corrección del algoritmo establece que en el caso de no devolver fallo, obtiene una solución del sistema que recibe como entrada (y por tanto se trata de un sistema unificable). La completitud afirma que al recibir como entrada un sistema unificable, el algoritmo no falla.

```
(defthm mgs-sel-soundness
  (implies (mgs-sel S)
           (solution (first (mgs-sel S)) S)))

(defthm mgs-sel-completeness
  (implies (solution sigma S)
           (mgs-sel S)))
```

Además, los siguientes teoremas establecen que en caso de que el algoritmo `mgs-sel` no falle, devuelve un unificador de máxima generalidad e idempotente:

```
(defthm mgs-sel-most-general-solution
 (implies (solution sigma S)
          (subs-subst (first (mgs-sel S)) sigma)))
```

```
(defthm mgs-sel-idempotent
 (idempotent (first (mgs-sel S))))
```

Por último, el siguiente teorema enuncia la correspondiente propiedad de clausura para el algoritmo dado por `mgs-sel`: si recibe como entrada un sistema de ecuaciones en una signatura, el unificador devuelto es una sustitución en la misma signatura:

```
(defthm substitution-s-p-mgs-sel
  (implies (system-s-p S)
           (substitution-s-p (first (mgs-sel S)))))
```

Como caso particular para sistemas con una sóla ecuación, podemos también presentar las principales propiedades de las funciones `mgu-sel` y `unifiable-sel`, formalizando así el teorema 4.34. Los dos siguientes teoremas expresan la corrección y completitud de tales definiciones. Es decir, `(unifiable-sel t1 t2)` es distinto de `nil` si y sólo si existe un unificador de ambos términos y en ese caso `(mgu-sel t1 t2)` es un unificador de máxima generalidad, idempotente y en la misma signatura que `t1` y `t2`:

```
(defthm unifiable-sel-completeness
  (implies (equal (instance t1 sigma) (instance t2 sigma))
           (unifiable-sel t1 t2)))

(defthm unifiable-sel-soundness
  (implies (unifiable-sel t1 t2)
           (equal (instance t1 (mgu-sel t1 t2))
                  (instance t2 (mgu-sel t1 t2)))))

(defthm mgu-sel-idempotent
  (idempotent (mgu-sel t1 t2)))

(defthm mgu-sel-most-general-unifier
  (implies (equal (instance t1 sigma) (instance t2 sigma))
           (subs-subst (mgu-sel t1 t2) sigma)))

(defthm mgu-sel-substitution-s-p
  (implies (and (term-s-p t1) (term-s-p t2))
           (substitution-s-p (mgu-sel t1 t2))))
```

4.4.4 Descripción de la demostración

Veamos una descripción de los principales lemas que guían a ACL2 hacia la prueba de los resultados anteriores. Seguiremos la demostración a mano presentada en la sección 4.4.1, que a su vez sigue una línea argumental análoga a la prueba de las propiedades del algoritmo de equiparación de la sección 3.3.

Sustituciones idempotentes

Antes de presentar la demostración de las propiedades, veamos la principal propiedad del predicado que define la idempotencia de sustituciones, que justifica la definición escogida. Si una sustitución es idempotente, es solución de sí misma:

```
(defthm idempotence
  (implies (idempotent S)
           (solution S S)))
```

Como se observa, en la formulación de este teorema estamos sacando partido de que la representación escogida permita ver una sustitución como un sistema de ecuaciones. Esto también ocurre en el siguiente teorema:

```
(defthm substitutions-solution-system
  (implies (solution sigma S)
           (equal (apply-subst flg sigma (apply-subst flg S term))
                  (apply-subst flg sigma term))))
```

Este teorema formaliza (y generaliza) el lema 4.26 y establece que todo sistema de ecuaciones subsume, como sustitución, a cualquiera de sus soluciones. Es decir, una consecuencia inmediata de este resultado es que si `sigma` es solución de `S`, entonces se tiene `(subs-subst S sigma)`.

Es fácil ver que los dos teoremas anteriores permiten concluir un análogo al teorema 4.27, que establece que toda sustitución idempotente (o lo que es lo mismo, un sistema en forma resuelta) es unificador de máxima generalidad de sí mismo. Por este motivo, los dos teoremas anteriores serán fundamentales para probar el teorema `mgs-sel-most-general-solution`, que veremos más adelante.

Terminación de la relación $\Rightarrow_u$

La admisión de la definición de `solve-system-sel` necesita probar previamente que termina para cualquier par de sistemas que reciba como entrada. Es decir, debemos probar formalmente que no es posible aplicar infinitas reducciones respecto de la relación $\Rightarrow_u$. Para ello, como sugiere el teorema 4.32, podemos definir la siguiente función de medida:

```
(defun unification-measure (S-sol)
  (cons  (cons (1+ (n-system-var (first S-sol)))
               (size-system (first S-sol)))
         (n-variables-right-hand-side (first S-sol))))
```

La función `unification-measure` asigna un ordinal a cada par de sistemas de ecuaciones. Tal y como se ha definido, permite simular el orden lexicográfico en $\mathbb{N}\times\mathbb{N}\times\mathbb{N}$ (sección 2.8 de [7]), a partir del orden entre ordinales. Comparar los ordinales `(unification-measure S1-sol1)` y `(unification-measure S2-sol2)` mediante `e0-ord-<` tiene el efecto de comparar lexicográficamente ternas de números naturales. En primer lugar se compara el número de variables distintas del primer sistema del par. En caso de igualdad, se comparan el número de símbolos. Y si nuevamente hay igualdad, se compara el número de ecuaciones cuyo lado derecho es una variable. Las funciones `n-system-var`,

`size-system` y `n-variables-right-hand-side` (cuyas sencillas definiciones omitimos) calculan, respectivamente, tales cantidades. Estas funciones son precisamente M_1, M_2 y M_3 de la demostración del teorema 4.32.

Los siguientes teoremas establecen cómo decrecen tales cantidades después de la aplicación de un paso de transformación a un par de sistemas que no está en forma normal:

```
(defthm transform-does-not-increases-n-system-var
  (implies (not (normal-form-syst S-sol))
           (>= (n-system-var (first S-sol))
               (n-system-var (first (transform-mm-sel S-sol))))))

(defthm if-n-system-var-is-equal-decrease-size
  (implies (and (not (normal-form-syst S-sol))
                (= (n-system-var (first S-sol))
                   (n-system-var (first (transform-mm-sel S-sol)))))
           (>= (size-system (first S-sol))
               (size-system (first (transform-mm-sel S-sol))))))

(defthm if-n-system-var-and-size-are-equal-decrease-n-variables-r-h-s
  (implies (and (not (normal-form-syst S-sol))
                (= (n-system-var (first S-sol))
                   (n-system-var (first (transform-mm-sel S-sol))))
                (= (size-system (first S-sol))
                   (size-system (first (transform-mm-sel S-sol)))))
           (< (n-variables-right-hand-side
                (first (transform-mm-sel S-sol)))
              (n-variables-right-hand-side (first S-sol)))))
```

Esto teoremas implican trivialmente que la aplicación de un paso de transformación sobre un par de sistemas que no está en forma normal hace que el ordinal que devuelve la función `unification-measure` decrezca estrictamente respecto de la relación `e0-ord-<`, demostrando así la terminación de la función `solve-system-sel`:

```
(defthm unification-measure-decreases
  (implies (not (normal-form-syst S-sol))
           (e0-ord-< (unification-measure (transform-mm-sel S-sol))
                     (unification-measure S-sol))))
```

Invariantes en las transformaciones

Como se ve en la prueba dada en los preliminares, la demostración de las principales propiedades del algoritmo de unificación `mgs-sel` consiste fundamentalmente en probar que una serie de propiedades permanecen invariantes a medida que se realizan los sucesivos pasos de transformación.

Como establecen los lemas 4.29 (apartado *a)*) y 4.30, podemos probar que el conjunto de soluciones del par de sistemas[10] es un invariante en las transformaciones que se llevan a cabo con $\Rightarrow_u$. Los tres teoremas siguientes enuncian este resultado:

[10] Considerando al sistema $\perp$ como un par de sistemas sin soluciones.

```
(defthm transform-mm-sel-equivalent-1
  (implies (and (consp S-sol)
                (consp (first S-sol))
                (solution sigma (union-systems S-sol)))
           (solution sigma (union-systems (transform-mm-sel S-sol)))))

(defthm transform-mm-sel-equivalent-2
  (implies (and (consp S-sol)
                (consp (first S-sol))
                (transform-mm-sel S-sol)
                (solution sigma (union-systems (transform-mm-sel S-sol))))
           (solution sigma (union-systems S-sol))))

(defthm transform-sel-unsolvable
  (implies (and (not (transform-mm-sel S-sol))
                (consp S-sol)
                (consp (first S-sol)))
           (not (solution sigma (union-systems S-sol)))))
```

Otro invariante de las transformaciones que se llevan a cabo es la idempotencia del segundo sistema y el hecho de que su dominio no contiene variables del primero de los sistemas (lema 4.29, apartado *b)*).

```
(defthm transform-mm-sel-preserves-idempotency
  (let* ((S (first S-sol)) (sol (cdr S-sol)))
    (implies (and (consp S-sol) (consp S)
                  (transform-mm-sel S-sol)
                  (idempotent sol)
                  (disjointp (system-var S) (domain sol)))
             (and (idempotent (cdr (transform-mm-sel S-sol)))
                  (disjointp
                    (system-var (first (transform-mm-sel S-sol)))
                    (domain (cdr (transform-mm-sel S-sol)))))))))
```

El último invariante es el que permite probar la propiedad de clausura del algoritmo `mgs-sel`. Si el paso de transformación se aplica sobre un par de sistemas tal que el primero de ellos es un sistema en una signatura y el segundo una sustitución en la misma signatura, entonces el sistema y sustitución resultantes lo son en la misma signatura:

```
(defthm transform-mm-sel-preserves-system-s-p
  (implies (and (consp S-sol)
                (consp (first S-sol))
                (system-s-p (first S-sol)))
           (system-s-p (first (transform-mm-sel S-sol)))))

(defthm transform-mm-sel-preserves-substitution-s-p
  (implies (and (consp S-sol)
                (consp (first S-sol))
```

```
            (system-s-p (first S-sol))
            (substitution-s-p (cdr S-sol)))
      (substitution-s-p (cdr (transform-mm-sel S-sol)))))))
```

La regla de eliminación

Como en la prueba a mano, la demostración de la terminación de la relación $\Rightarrow_s$ y de las propiedades invariantes durante el proceso de transformación, se tiene analizando cada una de las reglas que definen la relación. Algunas de estas propiedades se tienen de manera inmediata para algunas de las reglas. En otros casos, es necesario demostrar una serie de lemas previos.

Sería muy prolijo comentar con detalle la demostración de cada una de las propiedades para cada cada una de las reglas. Remitimos al lector al libro `unification-pattern.lisp`. Como ejemplo ilustrativo, en este apartado discutiremos con más detalle las propiedades de la regla **Elimina** relativas a la conservación del conjunto de soluciones y la terminación.

Para demostrar que la regla de eliminación aplica una transformación que conserva el conjunto de unificadores, como en la prueba a mano presentada en el lema 4.29, resulta fundamental el resultado enunciado por el teorema `substitutions-solution-system` dado anteriormente en esta subsección al hablar de las propiedades de las sustituciones idempotentes. Recuérdese que este teorema formaliza el lema 4.26 y establece que si `sigma` es solución de `S`, entonces `sigma` compuesta con `S` es igual que `sigma`. Como consecuencia se tiene el siguiente teorema:

```
(defthm main-property-eliminate
  (implies (solution sigma S1)
           (equal (solution sigma (apply-syst S1 S))
                  (solution sigma S))))
```

Si `equ` es la ecuación que se selecciona en el paso de eliminación, entonces podemos usar el teorema anterior en el caso particular en el que `S1` es justamente `(list equ)` y `S` el conjunto de restantes ecuaciones, para concluir que el conjunto de unificadores se conserva después de un paso de transformación aplicado con la regla **Elimina**.

Por lo que respecta a la regla **Elimina**, la terminación de la relación $\Rightarrow_u$ es fácil de probar. El teorema `transform-does-not-increases-n-system-var`, presentado anteriormente, establecía que para cualquier regla aplicada (y en particular la regla de eliminación) el número de variables distintas en el primero de los sistemas nunca aumentaba. En el caso de la regla de eliminación esta cantidad decrece estrictamente, ya que la variable que se elimina deja de aparecer en el primero de los sistemas del par:

```
(defthm eliminate-variables-strict
 (implies (and (member ecu S)
               (variable-p (car ecu))
               (not (member (car ecu)
                            (variables t (cdr ecu)))))
          (> (n-system-var S)
             (n-system-var
                (apply-syst (list ecu) (eliminate ecu S))))))
```

Conclusión de los teoremas principales

Es evidente que los invariantes en las transformaciones se conservan cuando se aplican de manera iterativa, como hace la función `solve-system-sel`. Por tanto se tienen los siguientes teoremas:

```
(defthm solve-system-sel-equivalent-1
  (implies (and (consp S-sol)
                (solution sigma (union-systems S-sol)))
           (solution sigma (union-systems (solve-system-sel S-sol)))))

(defthm solve-system-sel-equivalent-2
  (implies (and (consp S-sol)
                (solve-system-sel S-sol)
                (solution sigma (union-systems (solve-system-sel S-sol))))
           (solution sigma (union-systems S-sol))))

(defthm solve-system-sel-unsolvable
  (implies (and (consp S-sol)
                (not (solve-system-sel S-sol)))
           (not (solution sigma (union-systems S-sol)))))

(defthm solve-system-sel-preserves-idempotency
  (let* ((S (first S-sol)) (sol (cdr S-sol)))
     (implies (and (consp S-sol)
                   (solve-system-sel S-sol)
                   (idempotent sol)
                   (disjointp (system-var S) (domain sol)))
              (idempotent (cdr (solve-system-sel S-sol))))))

(defthm solve-system-sel-substitution-s-p
  (let* ((S (first S-sol)) (sol (cdr S-sol)))
     (implies (and (consp S-sol)
                   (solve-system-sel S-sol)
                   (system-s-p S) (substitution-s-p sol))
              (substitution-s-p (cdr (solve-system-sel S-sol))))))
```

Los teoremas anteriores establecen que después de aplicar exhaustivamente las reglas de transformación (hasta que se llega a una forma normal) se tienen la siguientes propiedades:

- El conjunto de soluciones del sistema obtenido finalmente es el mismo que el del par de sistemas de partida (teoremas `solve-system-sel-equivalent-1`, `solve--system-sel-equivalent-2` y `solve-system-sel-unsolvable`).

- Si inicialmente el segundo de los sistemas es idempotente y su dominio no tiene variables en común con el primero de los sistemas, el sistema finalmente obtenido es idempotente (teorema `solve-system-sel-preserves-idempotency`).

- Si se parte de un sistema y una sustitución en una signatura, el sistema finalmente obtenido es una sustitución en la misma signatura (teorema `solve-system-sel--substitution-s-p`).

Estos teoremas, a su vez, permiten deducir fácilmente las principales propiedades del algoritmo `mgs-sel`, presentadas en la subsección anterior. Recuérdese que `mgs-sel` llama a la función `solve-system-sel` sobre el par de sistemas `(cons S nil)`.

Veamos primero la demostración del teorema `mgs-sel-soundness`. Supongamos que `(mgs-sel S)` no devuelve fallo:

- Como caso particular del teorema `solve-system-sel-equivalent-2` (para `S-sol` igual a `(cons S nil)`), toda solución del sistema obtenido después de aplicar exhaustivamente los pasos de transformación, lo será del sistema `S`.
- Según el teorema `solve-system-sel-preserves-idempotency`, el sistema que finalmente se obtiene es un sistema en forma resuelta (o lo que es lo mismo, una sustitución idempotente).
- El teorema `idempotence` nos permite concluir que la sustitución devuelta por `mgs--sel` es solución de sí misma y según las consideraciones anteriores, lo será también del sistema de partida `S`.

El teorema `mgs-sel-most-general-solution` se tiene si previamente demostramos el siguiente lema:

```
(defthm mgs-sel-most-general-solution-main-lemma
 (implies (solution sigma S)
          (equal (instance (instance term (first (mgs-sel S))) sigma)
                 (instance term sigma))))
```

Es decir si `sigma` es una solución del sistema `S`, entonces `sigma` compuesto con la sustitución que devuelve `mgs-sel` es igual a `sigma`. Como consecuencia de este lema y teniendo en cuenta el teorema de completitud de la relación de subsunción entre sustituciones (teorema `subs-subst-completeness`, véase la subsección 3.4.3), se concluye el teorema `mgs-sel-most-general-solution` presentado anteriormente.

Expliquemos brevemente la demostración del lema `mgs-sel-most-general-solution-main-lemma`. Supongamos que `sigma` es solución del sistema `S`. Como caso particular del teorema `solve-system-sel-equivalent-1` (para `S-sol` igual a `(cons S nil)`), `sigma` también será solución del sistema obtenido después de aplicar los pasos de transformación. Es decir, `sigma` es solución del sistema devuelto por `mgs-sel`. El teorema `substitutions-solution-system` (presentado al describir las propiedades de las sustituciones idempotentes) nos permite entonces concluir el lema `mgs-sel-most-general--solution-main-lemma`.

Por último, los teoremas `mgs-sel-completeness`, `mgs-sel-idempotent` y `substitution-s-p-mgs-sel` son un caso particular de los teoremas `solve-system-sel-unsolvable`, `solve-system-sel-preserves-idempotency` y `solve-system-sel-substitution--s-p` respectivamente, sin más que instanciar `S-sol` por `(cons S nil)`.

En los dos primeros apartados de la subsección C.2.2, damos algunos detalles adicionales sobre cómo hemos llevado a cabo la demostración automática de estos resultados.

4.4.5 Un algoritmo de unificación ejecutable

Definimos en esta subsección un algoritmo de unificación ejecutable, instanciando el patrón general definido por `mgs-sel`. Para ello, especificamos una función de selección concreta, además de efectuar algunas modificaciones con el objetivo de mejorar la eficiencia del algoritmo. Las definiciones y eventos presentadas en esta subsección se encuentran en el libro `unification.lisp`.

Definición

La función de selección que vamos a definir busca una ecuación tal que se pueda concluir inmediatamente que el sistema no es unificable. Si en el sistema aparece una ecuación de la forma $f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m)$, con $f \neq g$, se selecciona con el objetivo de que la unificación falle en el siguiente paso de transformación. Como en el caso de la equiparación, la razón por la que definimos esta función de selección es que esperamos usar el algoritmo de unificación en contextos en los que en la mayoría de los casos la unificación falla (por ejemplo, en el cálculo de pares críticos de un sistema de ecuaciones, véase la sección 7.5), esperando así detectar cuanto antes el fallo. La función de selección viene definida por `find-not-unifiable`:

```
(defun find-not-unifiable (S)
  (if (endp (cdr S))
      (car S)
    (let* ((equ (car S)) (t1 (car equ)) (t2 (cdr equ)))
        (cond ((or (variable-p t1) (variable-p t2))
               (find-not-unifiable (cdr S)))
              ((eql (car t1) (car t2))
               (find-not-unifiable (cdr S)))
              (t equ)))))
```

En la figura 4.4 definimos el algoritmo `mgs-mv` que unifica sistemas de ecuaciones. Como caso particular también definimos un algoritmo de unificación de términos, llamado `mgu-mv`. Su definición es esencialmente igual a la de `mgs-sel` y `mgu-sel`, sustituyendo la función de selección `sel` por la función `find-not-unifiable`.

Sin embargo, existen algunas modificaciones respecto del patrón general definido por `mgs-sel`. Estas modificaciones están encaminadas a mejorar la eficiencia del algoritmo ejecutable:

- La función `transform-mm` recibe el par de sistemas sobre el que actúa como dos argumentos separados y devuelve tres valores agrupados en un multivalor: los dos primeros para el par de sistemas transformados y el tercero un valor booleano indicando el éxito o fallo del paso de transformación. El uso de multivalores aumenta la eficiencia, ya que evita el coste de la construcción del par punteado.

- El uso de multivalores se transmite también a las funciones `solve-system`, `mgs-mv` y `mgu-mv`.

- La comparación entre símbolos se lleva a cabo usando `eql`, más eficiente que `equal`. Esto está permitido, ya que los términos que se reciben como entrada van a ser

```
(defun transform-mm (S sol)
  (let* ((ecu (find-not-unifiable S))
         (t1 (car ecu)) (t2 (cdr ecu))
         (R (eliminate ecu S)))
    (cond ((equal t1 t2) (mv R sol t))
          ((variable-p t1)
           (if (member t1 (variables t t2))
               (mv nil nil nil)
             (mv (substitute-syst t1 t2 R)
                 (cons ecu (substitute-range t1 t2 sol))
                 t)))
          ((variable-p t2)
           (mv (cons (cons t2 t1) R) sol t))
          ((not (eql (car t1) (car t2)))
           (mv nil nil nil))
          (t (mv-let (pairs bool)
                     (pair-args (cdr t1) (cdr t2))
                     (if bool
                         (mv (append pairs R) sol t)
                       (mv nil nil nil)))))))

(defun solve-system (S sol bool)
  (if (or (not bool) (not (consp S)))
      (mv S sol bool)
    (mv-let (S1 sol1 bool1)
            (transform-mm S sol)
            (solve-system S1 sol1 bool1))))

(defun mgs-mv (S)
  (mv-let (S1 sol1 bool1)
          (solve-system S nil t)
          (mv sol1 bool1)))

(defun mgu-mv (t1 t2)
  (mgs-mv (list (cons t1 t2))))
```

Figura 4.4: Un algoritmo de unificación ejecutable

términos cuyos símbolos verifican el predicado `eqlablep`. Más detalles sobre este particular en el apéndice B.

- La implementación de la regla **Elimina** se lleva a cabo con las funciones `substitute-syst` y `substitute-range` que sustituyen, respectivamente, a las menos eficientes `apply-syst` y `apply-range` que se usan en la versión no determinista. Estas funciones llevan a cabo la sustitución directa de una variable por un término en un

sistema (o en su rango) sin tener que recurrir a la función `apply-subst` (véanse las definiciones en la sección `3.2` del libro `terms.lisp`).

Lo que sigue son algunos ejemplos de ejecución de la función `mgu-mv` para calcular un unificador de máxima generalidad de dos términos[11]:

```
ACL2 !>(mgu-mv '(g (g x (f y))) '(g (g u u)))
(((U . (F Y)) (X . (F Y))) T)
ACL2 !>(mgu-mv '(g x) '(f y))
(NIL NIL)
ACL2 !>(mgu-mv '(f x y) '(f y (h x)))
(NIL NIL)
ACL2 !>(mgu-mv '(f x (g v) (a)) '(f (h y z) y v))
(((Y . (G (A))) (V . (A)) (X . (H (G (A)) Z))) T)
ACL2 !>(mgu-mv '(k z (f x (b) z)) '(k (h x) (f (g (a)) y z)))
(((Y. (B)) (X . (G (A))) (Z . (H (G (A))))) T)
ACL2 !>(mgu-mv '(k z (f x (b) z)) '(k (h x) (f (g z)) y z))
(NIL NIL)
```

Las principales propiedades del algoritmo de unificación `mgu-mv`

Mediante instanciación funcional de las propiedades del algoritmo de unificación no determinista, podemos ahora fácilmente obtener las propiedades del algoritmo de unificación de pares de términos definido por `mgu-mv`[12]. Nótese que `mgu-mv` es una función que recibe como entrada dos términos y devuelve un multivalor doble consistente en un valor booleano (el segundo de ellos, indicando éxito o fallo en la unificación) y el unificador calculado (el primero de ellos). Para mejorar la formulación de los teoremas siguientes, definimos las siguientes funciones, correspondientes al segundo y primer valor devuelto por la función `mgu-mv`. Así, el predicado `unifiable` comprueba si dos términos son unificables y en ese caso la función `mgu` calcula un unificador de máxima generalidad[13]:

```
(defun unifiable (t1 t2)
  (mv-let (mgu unifiable)
          (mgu-mv t1 t2)
          unifiable))

(defun mgu (t1 t2)
  (mv-let (mgu unifiable)
          (mgu-mv t1 t2)
          mgu))
```

Con todas estas definiciones podemos expresar los siguientes teoremas que verifican formalmente a la función `mgu-mv`. Estos teoremas establecen que dos términos son unificables si y sólo si el algoritmo `mgu-mv` no devuelve fallo y en ese caso, calcula un unificador

[11] Hemos conservado la notación de par punteado para una mayor legibilidad.

[12] De la misma manera se podrían deducir las propiedades del algoritmo de unificación de sistemas definido por `mgs-mv`.

[13] Estas definiciones son sólo para mejorar la presentación de los teoremas que se presentan a continuación. La ejecución del algoritmo de unificación se llevará a cabo con `mgu-mv`, que comprueba la unificabilidad y calcula el unificador de máxima generalidad al mismo tiempo.

de máxima generalidad, idempotente y en la misma signatura que los términos sobre los que actúa (es decir, demuestran formalmente el teorema 4.34):

```
(defthm  mgu-completeness
  (implies (equal (instance t1 sigma)
                  (instance t2 sigma))
           (unifiable t1 t2)))

(defthm mgu-soundness
  (implies (unifiable t1 t2)
           (equal (instance t1 (mgu t1 t2))
                  (instance t2 (mgu t1 t2)))))

(defthm mgu-idempotent
  (idempotent (mgu t1 t2)))

(defthm mgu-most-general-unifier
  (implies (equal (instance t1 sigma)
                  (instance t2 sigma))
           (subs-subst (mgu t1 t2) sigma)))

(defthm mgu-substitution-s-p
  (implies (and (term-s-p t1) (term-s-p t2))
           (substitution-s-p (mgu t1 t2))))
```

En el último apartado de la subsección C.2.2, explicamos cómo estos teoremas se obtienen a partir de instancias funcionales de las correspondientes versiones no deterministas.

4.4.6 Especificaciones basadas en reglas

Hemos presentado la verificación de un algoritmo de unificación que se especifica mediante un conjunto de reglas de transformación. Usando las mismas técnicas, en la sección 3.3 habíamos definido y verificado un algoritmo de equiparación especificado mediante reglas. Sin embargo, los algoritmos de renombrado (sección 4.1.3) y de anti–unificación (sección 4.3.1) se han verificado a partir de una definición recursiva "clásica" en la estructura de los términos. En un trabajo previo a esta formalización, habíamos definido y verificado también un algoritmo de equiparación de términos basado en una definición "clásica" en la estructura de los términos. Este trabajo se encuentra en el libro `subsumption-definition-v0.lisp`. Comparando, podemos señalar una serie de ventajas obtenidas en la verificación mecánica de algoritmos que se especifican mediante reglas:

- Se separan claramente la lógica de los algoritmos, de su control. En nuestro caso, el control del algoritmo lo da la función de selección y la lógica del mismo viene definida por las reglas de transformación.

- Como la función de selección no está completamente especificada, se está verificando con el mismo esfuerzo una "familia" de algoritmos, uno por cada función de selección

concreta. La regla de inferencia derivada de instanciación funcional nos permite establecer las propiedades de los algoritmos que se obtienen, si se define completamente la función de selección.

- El grado de abstracción es mayor y por tanto es necesario manejar menos conceptos intermedios que en la verificación de la versión "clásica" del algoritmo.

- Las técnicas de prueba son similares para todos los algoritmos especificados mediante reglas: funciones de selección, invariantes, terminación, etc. Compruébese la analogía entre las verificaciones de los algoritmos de equiparación y de unificación.

La especificación basada en reglas, sin embargo, no parece muy natural cuando, como en el caso de la anti–unificación, la respuesta buscada es un término en lugar de una sustitución.

4.5 Supremo de dos términos

Usando el algoritmo de unificación de términos, podemos demostrar que cualesquiera dos términos que tengan una instancia en común tienen un supremo. En esta sección presentamos la demostración de tal resultado en ACL2. Los eventos que conducen a los teoremas presentados se encuentran en el libro `mg-instance.lisp`.

Preliminares

Un unificador de máxima generalidad de dos términos y el supremo de los mismos en el orden de subsunción tienen una estrecha relación, como se establece en el siguiente teorema.

Teorema 4.36 *Sean s y t dos términos cualesquiera. Dada ξ un renombrado de las variables de t tal que $\mathcal{V}(s) \cap \mathcal{V}(\xi(t)) = \emptyset$ y γ un unificador de máxima generalidad de s y $\xi(t)$. Entonces $\gamma(s)$ es un supremo de s y t en el orden de subsunción. En particular, si $\mathcal{V}(s) \cap \mathcal{V}(t) = \emptyset$ y μ es un unificador de máxima generalidad de s y t, entonces $\mu(s)$ es un supremo de s y t en el orden de subsunción.*

Demostración:
Sea u cota superior de s y t. Esto implica que $\xi(t) \preccurlyeq u$. Sea σ_1 tal que $\sigma_1(\xi(t)) = u$ y σ_2 tal que $\sigma_2(s) = u$. Puesto que $\mathcal{V}(\xi(t)) \cap \mathcal{V}(s) = \emptyset$, entonces podemos tomar σ_1 y σ_2 verificando además que $\mathcal{D}(\sigma_1) \cap \mathcal{D}(\sigma_2) = \emptyset$. Tiene sentido, por tanto, hablar de $\sigma_{12} = \sigma_1 \cup \sigma_2$. Es claro que σ_{12} es unificador de s y $\xi(t)$. Por tanto, existe δ tal que $\sigma_{12} = \delta\gamma$. Luego $\delta(\gamma(s)) = \sigma_{12}(s) = u$. Es decir, $\gamma(s) \preccurlyeq u$. Concluimos por tanto que $\gamma(s)$ es más general que cualquier cota superior de s y t. Por otra parte, $\gamma(s)$ es cota superior de s y t ya que $s \preccurlyeq \gamma(s)$ (evidente) y $t \preccurlyeq \gamma(s)$ (ya que $\gamma(s) = \gamma\xi(t)$). Luego $\gamma(s)$ es supremo de s y t. El caso particular se tiene tomando ξ como la sustitución identidad. □

La diferencia entre encontrar un unificador de máxima generalidad de dos términos y encontrar un supremo radica en que en el primer caso, una única sustitución se aplica a los dos términos, mientras que en en el segundo, podemos aplicar diferentes sustituciones.

Supremo de dos términos en ACL2 y principales propiedades

La función `mg-instance-mv` implementa el cálculo del supremo de dos términos dados respecto de la relación de subsunción. Esta función devuelve un multivalor doble: el segundo de ellos es un valor booleano indicando la existencia o no de tal supremo. En caso afirmativo, el primero de los valores obtenidos por `mg-instance-mv` es el supremo de los términos que se reciben como entrada:

```
(defun mg-instance-mv (t1 t2)
  (let ((rename-t1 (number-rename t1 0 -1))
        (rename-t2 (number-rename t2 1 1)))
    (mv-let (mgu unifiable)
            (mgu-mv rename-t1 rename-t2)
            (if unifiable
                (mv (instance rename-t1 mgu) t)
              (mv nil nil)))))
```

La definición de la función `mg-instance-mv` está inspirada en el teorema 4.36. Como se observa, se renombran las variables de los términos, usando `number-rename`, de manera que sus variables quedan separadas. A continuación se llama al algoritmo de unificación sobre los términos renombrados. Si la unificación falla, no existe supremo. En caso contrario, el supremo se calcula aplicando el unificador de máxima generalidad obtenido a uno cualquiera de los términos renombrados (el primero en este caso).

Para formular los teoremas que presentaremos más adelante, será más cómodo usar la siguiente función que prescinde del uso de multivalores:

```
(defun mg-instance (t1 t2)
  (mv-let (mg-instance bool)
          (mg-instance-mv t1 t2)
    (if bool
        (number-rename mg-instance 1 1)
      nil)))
```

La función `mg-instance` llama a `mg-instance-mv`. Si ésta falla, `mg-instance` devuelve `nil`. En caso contrario, toma el término calculado por `mg-instance-mv` y renombra sus variables a números. El que se produzca este último renombrado tiene una doble finalidad: en primer lugar mejora la presentación del término calculado y en segundo lugar, permite usar `nil` como indicación de fallo, sin recurrir a multivalores: en caso de existencia de supremo, el término devuelto por `mg-instance` nunca puede ser `nil` ya que sus variables son numéricas. A continuación presentamos algunos ejemplos del cálculo del supremo de dos términos (o de la detección de la inexistencia del mismo) usando `mg-instance`:

```
ACL2 !>(mg-instance '(f x (h y)) '(f (k u) u))
(F (K (H 1)) (H 1))
ACL2 !>(mg-instance '(f x (h x)) '(f (k u) u))
NIL
ACL2 !>(mg-instance '(f x (h y)) '(f (k u) z))
(F (K 1) (H 2))
```

```
ACL2 !>(mg-instance '(f x) 'x)
(F 1)
ACL2 !>(mg-instance '(f u v u v u) '(f x y x x y))
(F 1 1 1 1 1)
ACL2 !>(mg-instance '(f u v u v u) '(f x y x x y z))
NIL
```

Los siguientes teoremas establecen las propiedades fundamentales de `mg-instance` y demuestran que existe el supremo de dos términos (calculado por `mg-instance`) si y sólo si tales términos tienen instancias comunes:

```
(defthm common-instance-implies-mg-instance
  (implies (and (subs t1 term)
                (subs t2 term))
           (mg-instance t1 t2)))

(defthm mg-instance-upper-bound
  (implies (mg-instance t1 t2)
           (and (subs t1 (mg-instance t1 t2))
                (subs t2 (mg-instance t1 t2)))))

(defthm mg-instance-least-upper-bound
  (implies (and (subs t1 term)
                (subs t2 term))
           (subs (mg-instance t1 t2) term)))
```

Además se tiene también la correspondiente propiedad de clausura para `mg-instance`:

```
(defthm mg-instance-term-s-p
  (implies (and (term-s-p t1) (term-s-p t2))
           (term-s-p (mg-instance t1 t2))))
```

Descripción de la demostración

La demostración se divide en dos partes bien diferenciadas y está basada en la del teorema 4.36. En primer lugar, razonamos con los términos renombrados y demostramos las propiedades, relativas a la relación de subsunción, que tiene la función `mg-instance-mv`. Estas propiedades se trasladan fácilmente a los términos originales (sin renombrar) ya que el renombrado obtiene términos equivalentes respecto de la relación de subsunción.

La propiedad de clausura se verifica para `mg-instance-mv`, como especifica el siguiente resultado:

```
(defthm mg-instance-mv-term-s-p
  (implies (and (term-s-p t1) (term-s-p t2))
           (term-s-p (first (mg-instance-mv t1 t2)))))
```

Veamos ahora también los resultados encaminados a probar que `mg-instance-mv` calcula un supremo respecto de subsunción, de los renombrados de `t1` y `t2`. Primeramente,

es inmediato de la definición de `mg-instance-mv` que cuando `(mg-instance-mv t1 t2)` no falla, obtiene una instancia del correspondiente renombrado de `t1`. Por tanto, usando completitud de `subs` se tiene:

```
(defthm mg-instance-mv-subsumes-rename-1
  (implies (second (mg-instance-mv t1 t2))
           (subs (number-rename t1 0 -1)
                 (first (mg-instance-mv t1 t2)))))
```

De la misma manera y teniendo en cuenta que el término es obtenido a partir de un unificador de los renombrados de `t1` y `t2`, usando el teorema `mgu-soundness` (sección anterior) se tiene el resultado análogo para el renombrado de `t2`:

```
(defthm mg-instance-mv-subsumes-rename-2
  (implies (second (mg-instance-mv t1 t2))
           (subs (number-rename t2 1 1)
                 (first (mg-instance-mv t1 t2)))))
```

Probamos ahora que el término obtenido por `(mg-instance-mv t1 t2)` subsume a cualquier otra instancia común de los renombrados de `t1` y `t2`. Supongamos que `term` es una instancia común de `(number-rename t1 0 -1)` y `(number-rename t1 1 1)`, obtenida mediante las sustituciones `sigma1` y `sigma2` respectivamente. A partir de estas sustituciones, podemos obtener una única sustitución que sea unificador de los renombrados de `t1` y `t2` respectivamente. La siguiente función nos sirve para tal propósito:

```
(defun disjoint-union-subst (sigma1 sigma2 t1 t2)
  (append (restriction sigma1 (variables t (number-rename t1 0 -1)))
          (restriction sigma2 (variables t (number-rename t2 1 1)))))
```

El hecho de que los renombrados de `t1` y `t2` tengan variables separadas hace que la sustitución construida por `disjoint-union-subst` sea un unificador de ambos términos, ya que actúa sobre tales términos como lo hacen `sigma1` y `sigma2`, respectivamente:

```
(defthm disjoint-union-subst-unifier
 (implies (equal (instance (number-rename t1 0 -1) sigma1)
                 (instance (number-rename t2 1 1) sigma2))
          (equal (instance (number-rename t2 1 1)
                           (disjoint-union-subst sigma1 sigma2 t1 t2))
                 (instance (number-rename t1 0 -1)
                           (disjoint-union-subst sigma1 sigma2 t1 t2))))))
```

Es decir, mediante separación de variables ha sido posible pasar de dos sustituciones `sigma1` y `sigma2` que servían para obtener una instancia común, a una única sustitución, dada por `disjoint-union-subst`, que obtiene la misma instancia común (es decir, tenemos un unificador). Por tanto, por el teorema `mgu-most-general-unifier`, ambos términos renombrados son unificables y el unificador de máxima generalidad que calcula `mgu-mv` subsume al unificador construido:

```
(defthm subsumption-sust-mgu-disjoint-union
 (implies (equal (instance (number-rename t1 0 -1) sigma1)
                 (instance (number-rename t2 1 1) sigma2))
          (subs-subst (mgu (number-rename t1 0 -1)
                           (number-rename t2 1 1))
                      (disjoint-union-subst sigma1 sigma2 t1 t2)))))
```

Por el teorema de corrección de la relación `subs-subst` (`subs-subst-soundness`, véase sección 3.4.3) y el teorema de completitud de la relación `subs`, se tiene que cualquier instancia de un término que se obtenga con el unificador de máxima generalidad, subsume a la instancia del mismo término que se obtenga con el unificador construido por `disjoint--union-subst`. En particular, si el término al que nos referimos es el renombrado de `t1` se obtiene el siguiente teorema:

```
(defthm mg-instance-mv-lub-rename-bridge-lemma
 (implies (equal (instance (number-rename t1 0 -1) sigma1)
                 (instance (number-rename t2 1 1) sigma2))
          (subs (first (mg-instance-mv t1 t2))
                (instance (number-rename t1 0 -1)
                          (disjoint-union-subst sigma1 sigma2 t1 t2)))))
```

Aplicando la propiedad de corrección de `subs`, es posible formular las hipótesis del teorema anterior en términos de la relación de subsunción, expresando justamente que si los términos renombrados tienen una cota superior bajo la relación de subsunción, entonces, el término obtenido mediante `mg-instance-mv` es menor que esa cota superior.

```
(defthm mg-instance-mv-lub-rename
 (implies (and (subs (number-rename t1 0 -1) term)
               (subs (number-rename t2 1 1) term))
          (subs (first (mg-instance-mv t1 t2)) term))
```

Por último, téngase en cuenta que cuando los renombrados de `t1` y `t2` tienen una cota superior común bajo la relación de subsunción, entonces la unificación de ambos nunca devuelve fallo (por la corrección de `mgu-mv`, teorema `mgu-completeness`, sección 4.4). Por tanto, `mg-instance-mv` tampoco:

```
(defthm if-subsume-common-therm-mg-instance-mv
  (implies (and (subs (number-rename t1 0 -1) term)
                (subs (number-rename t2 1 1) term))
           (second (mg-instance-mv t1 t2))))
```

Los teoremas anteriores sirven para probar que existe un supremo de los renombrados respectivos de `t1` y `t2` si y sólo si tienen una instancia común; y en ese caso, `mg-instance-mv` calcula un supremo de los términos renombrados, bajo la relación de subsunción.

Una vez probado que `(mg-instance-mv t1 t2)` calcula un supremo de los términos `(number-rename t1 0 -1)` y `(number-rename t2 1 1)`, podemos probar que cuando no falla, el término calculado por `(mg-instance t1 t2)` es el supremo de `t1` y `t2`, sin más

que tener en cuenta que, respecto de la relación de subsunción, los términos equivalentes (según la relación `renamed`) se comportan de manera equivalente (sección 4.1.4). En este caso:

- La función `number-rename` obtiene términos equivalentes respecto a la relación `subs` (teorema `number-renamed-term-renamed-term`, sección 4.1.3).
- Por tanto los términos `t1` y `t2` son equivalentes a sus respectivos renombrados.
- Y el término que devuelve `mg-instance` (cuando no falla), es equivalente al que devuelve `mg-instance-mv`.

Usando un razonamiento análogo se prueba también la propiedad de clausura de `mg-instance` a partir de la propiedad de clausura demostrada anteriormente sobre `mg--instance-mv`.

El uso del mecanismo de reescritura congruente, tal y como se explica en la subsección 4.1.4 hace muy simple y directa la demostración automática de las propiedades de `mg-instance` a partir de las propiedades de `mg-instance-mv`.

4.6 El retículo de los términos de primer orden

4.6.1 Preliminares

Como se ha probado en las secciones anteriores, todo par de términos tiene un ínfimo respecto de la relación de subsunción. Lo mismo no se puede afirmar del supremo, ya que existen pares de términos que no son unificables. Para dotar de una estructura de retículo al conjunto de los términos de primer orden en un signatura dada, vamos a incluir un elemento nuevo, que notaremos $\top$, que consideraremos más particular que cualquier término y por tanto se puede considerar como el supremo de cualquier par de términos no unificables.

El siguiente teorema resume todas las propiedades reticulares del conjunto de los términos de primer orden que hemos probado anteriormente.

Teorema 4.37 *Sea $\top \notin T(\Sigma, X)$ un elemento cualquiera. Sea $\widehat{T} = T(\Sigma, X) \cup \{\top\}$ y extendamos el orden $\preccurlyeq$ al conjunto $\widehat{T}$ definiendo $s \preccurlyeq \top$ para todo $s \in T(\Sigma, X)$. Entonces $(\widehat{T}, \preccurlyeq)$ es un retículo bien fundamentado.*

Demostración:
Nótese que el hecho de añadir el elemento $\top$ no influye en la buena fundamentación de $\widehat{T}$, ya que las mismas propiedades se tienen para $T(\Sigma, X)$. Lo mismo ocurre con la existencia de ínfimo. Para probar la existencia de supremo para cualquier par de elementos de $\widehat{T}$ nótese en primer lugar que si u es un supremo de s y t en $(T(\Sigma, X), \preccurlyeq)$, entonces lo es también en $(\widehat{T}, \preccurlyeq)$. Además es claro que el supremo de $\top$ y de cualquier $s \in \widehat{T}$ es $\top$. Por tanto, bastará ver que el supremo en $(\widehat{T}, \preccurlyeq)$ de dos términos cualesquiera $s, t \in T(\Sigma, X)$ que no tengan supremo en $(T(\Sigma, X), \preccurlyeq)$ es $\top$. Pero según el teorema 4.36, si s y t no tienen supremo en $(T(\Sigma, X), \preccurlyeq)$, entonces no existe una cota superior de ambos términos y por tanto es claro que $\top$ es la menor cota superior de s y t en en $(T(\Sigma, X), \preccurlyeq)$. □

Es de destacar que un resultado de la teoría de retículos nos dice que en todo semirretículo inferior bien fundamentado, los subconjuntos acotados superiormente tienen supremo. Por tanto, desde un punto de vista teórico, esto nos permitiría deducir, a partir de la demostración de la existencia de ínfimo y de la buena fundamentación de la subsunción, el hecho de que $(\widehat{T}, \preccurlyeq)$ es un retículo completo, sin necesidad de demostrar la corrección y completitud de un algoritmo de unificación.

4.6.2 El retículo de los términos

Como compilación de los principales resultados presentados en este capítulo y en el anterior, podemos establecer formalmente en ACL2 el teorema 4.37. Los teoremas que se presentan a continuación son consecuencia directa de estos resultados. La única diferencia es la introducción del objeto $\top$ que consideraremos mayor que cualquier término, respecto del orden de subsunción.

Algunas cuestiones técnicas

Representaremos en ACL2 al objeto $\top$ mediante el símbolo `'the-top-term`. La macro `the-top-term` define este elemento distinguido y la macro `is-the-top-term` define un reconocedor para el mismo:

```
(defmacro the-top-term ()
  ''the-top-term)

(defmacro is-the-top-term (term)
  `(equal ,term (the-top-term)))
```

Como veremos, necesitamos definir la sustitución testigo de que cualquier término subsume a $\top$. La macro `the-top-substitution` define tal objeto distinguido (notémoslo de ahora en adelante por $\sigma_\top$) y la macro `is-the-top-substitution` lo reconoce:

```
(defmacro the-top-substitution ()
  ''the-top-substitution)

(defmacro is-the-top-substitution (subst)
  `(equal ,subst (the-top-substitution)))
```

De esta manera podemos definir el conjunto $T(\Sigma, X) \cup \{\top\}$ mediante el siguiente predicado `ext-term-s-p`:

```
(defun ext-term-s-p (term)
  (or (term-s-p term)
      (is-the-top-term term)))
```

Podemos también, de manera análoga, definir un concepto "extendido" de sustitución en una signatura dada:

```
(defun ext-substitution-s-p (sigma)
  (or (substitution-s-p sigma)
      (is-the-top-substitution sigma)))
```

Más adelante nos hará falta asegurarnos que los términos y sustituciones obtenidos por los algoritmos verificados en las secciones anteriores son diferentes, respectivamente, a los objetos que representan a $\top$ y a $\sigma_\top$. Para forzar esta diferenciación necesitamos las siguientes funciones, `fix-tem` y `fix-substitution`, que obtienen términos y sustituciones equivalentes, pero distintos a `'the-top-term` y a `'the-top-substitution`, respectivamente.

```
(defun fix-term (term)
  (if (variable-p term) 0 term))

(defun fix-substitution (sigma)
  (if (atom sigma) nil sigma))
```

Los teoremas

La función `app<=` define la aplicación de una sustitución a un término, usando `instance` y teniendo en cuenta también lo que ocurre con $\top$, ya que $\delta(\top) = \top$ y que $\sigma_\top(s) = \top$, para cualquier sustitución δ y cualquier término s. Además, el teorema que sigue establece que la aplicación de una sustitución a un término es una operación cerrada:

```
(defun app<= (sigma term)
  (if (or (is-the-top-substitution sigma)
          (is-the-top-term term))
      (the-top-term)
    (instance term sigma)))

(defthm app<=-substitution-s-p
  (implies (and (ext-term-s-p term)
                (ext-substitution-s-p sigma))
           (ext-term-s-p (app<= sigma term))))
```

A partir de la función `subs`, la función `s<=` define la relación de subsunción en $T(\Sigma, X) \cup \{\top\}$, teniendo en cuenta que $\top$ es subsumido por cualquier término. La función `match<=` (que usa `match`) define la sustitución testigo de la subsunción entre dos términos. Las propiedades que se listan a continuación verifican que realmente se trata de la relación de subsunción:

```
(defun s<= (t1 t2)
  (cond  ((is-the-top-term t2) t)
         ((is-the-top-term t1) nil)
         (t (subs t1 t2))))

(defun match<= (t1 t2)
  (cond  ((is-the-top-term t2) (the-top-substitution))
         ((is-the-top-term t1) nil)
         (t (fix-substitution (matching t1 t2)))))

(defthm app<=-s<=-1
```

```
  (implies (equal t2 (app<= sigma t1))
           (s<= t1 t2)))

(defthm app<=-s<=-2
  (implies (s<= t1 t2)
           (equal (app<= (match<= t1 t2) t1) t2)))

(defthm match<=-ext-substitution-s-p
  (implies (and (ext-term-s-p t1) (ext-term-s-p t2))
           (ext-substitution-s-p (match<= t1 t2))))
```

La relación definida por `s<=` es un preorden bien fundamentado. La buena fundamentación viene justificada por la inmersión en los ordinales definida por la función `measure-s<`. Nótese que esta función asigna el ordinal ω^ω al elemento $\top$ y usa `subsumption-measure` para el resto de términos.

```
(defthm s<=-reflexivity
  (s<= term term))

(defthm s<=-transitivity
  (implies (and (s<= t1 t2) (s<= t2 t3))
           (s<= t1 t3)))

(defun s< (t1 t2)
  (and (s<= t1 t2) (not (s<= t2 t1))))

(defun measure-s< (term)
  (if (is-the-top-term term)
      '((1 . 0) . 0)
    (subsumption-measure term)))

(defthm s<-well-founded
  (and (e0-ordinalp (measure-s< term))
       (implies (s< t1 t2)
                (e0-ord-< (measure-s< t1) (measure-s< t2)))))
```

La función `glb-s<=` calcula el ínfimo de dos términos, usando para ello la función `anti-unify` y teniendo en cuanta el comportamiento especial de $\top$. Los teoremas que aparecen a continuación verifican que efectivamente así es:

```
(defun glb-s<= (t1 t2)
  (cond ((is-the-top-term t1) t2)
        ((is-the-top-term t2) t1)
        (t (fix-term (anti-unify t1 t2)))))

(defthm glb-s<=-lower-bound
  (and (s<= (glb-s<= t1 t2) t1)
       (s<= (glb-s<= t1 t2) t2)))
```

```
(defthm glb-s<=-is-greater-than-any-lower-bound
  (implies (and (s<= term t1)
                (s<= term t2))
           (s<= term (glb-s<= t1 t2))))

(defthm glb-s<=-closure-property
  (implies (and (ext-term-s-p t1) (ext-term-s-p t2))
           (ext-term-s-p (glb-s<= t1 t2))))
```

Finalmente definimos la función `lub-s<=`, que usa la función `mg-instance` para calcular el supremos de dos términos. Los teoremas que aparecen a continuación así lo establecen:

```
(defun lub-s<= (t1 t2)
  (cond ((or (is-the-top-term t1)
             (is-the-top-term t2))
         (the-top-term))
        ((mg-instance t1 t2) (fix-term (mg-instance t1 t2)))
        (t (the-top-term))))

(defthm lub-s<=-upper-bound
  (and (s<= t1 (lub-s<= t1 t2))
       (s<= t2 (lub-s<= t1 t2))))

(defthm lub-s<=-less-than-any-upper-bound
  (implies (and (s<= t1 term)
                (s<= t2 term))
           (s<= (lub-s<= t1 t2) term)))

(defthm lub-s<=-closure-property
  (implies (and (ext-term-s-p t1) (ext-term-s-p t2))
           (ext-term-s-p (lub-s<= t1 t2))))
```

Los teoremas anteriores demuestran formalmente en ACL2 que la relación de subsunción entre términos dota al conjunto de términos de primer orden en una signatura de una estructura de retículo bien fundamentado. Nótese que en realidad se ha probado un resultado más general: el conjunto de todos los objetos ACL2, vistos como representación de términos de primer orden en un sentido amplio (incluyendo términos propios e impropios, véase 3.1.2) es un retículo bien fundamentado. El conjunto de los términos en una signatura es un subrretículo de este retículo general.

Es de destacar el hecho de que la prueba es totalmente constructiva. De hecho todas las funciones definidas en esta sección son ejecutables en ACL2.

Sumario

En este capítulo:

- Hemos formalizado el concepto de equivalencia entre términos y de sustitución de renombrado.
- Hemos definido y verificado un algoritmo de renombrado de variables.
- Hemos demostrado que el orden de subsunción (estricto) está bien fundamentado.
- Hemos demostrado la existencia del ínfimo de cualquier par de términos, mediante la definición y verificación de un algoritmo de anti–unificación.
- Hemos definido y verificado un algoritmo de unificación basado en reglas de transformación (el algoritmo de Martelli y Montanari).
- A partir del algoritmo de unificación, hemos demostrado la existencia de supremo para cualquier par de términos unificables.
- Finalmente, hemos recopilado los resultados anteriores para demostrar que el conjunto de los términos de primer orden en una signatura (junto con un elemento adicional), tiene una estructura de retículo bien fundamentado.

Capítulo 5

Multiconjuntos y pruebas de terminación

Nos separamos momentáneamente del desarrollo de la teoría acerca de la lógica ecuacional, para explicar una utilidad que nos hará falta en el capítulo siguiente. Se trata del uso de multiconjuntos para definir relaciones bien fundamentadas.

Como se ha visto en el capítulo 2, la admisión del axioma que define una función recursiva en la lógica de ACL2, exige, como medida de prevención para evitar la aparición de inconsistencias, probar la terminación del algoritmo recursivo que implementa. Para ello, se ha de definir una medida, que tome valores en un conjunto de objetos en el que se ha definido una relación bien fundamentada, y se ha de probar que dicha medida, aplicada a los argumentos de la función, decrece respecto de la relación bien fundamentada en cada llamada recursiva.

Para definiciones recursivas sencillas, esta prueba de terminación se efectúa automáticamente por el sistema. Sin embargo, existen esquemas recursivos complicados que el sistema no es capaz de resolver sin ayuda por parte del usuario, que debe indicar expresamente la relación bien fundamentada y la función de medida con la que se intentará la prueba de terminación.

Una herramienta para demostrar la terminación de funciones recursivas son los multiconjuntos. Un multiconjunto se puede definir, de manera informal, como un conjunto con "elementos repetidos". Dershowitz y Manna [16] demostraron que toda relación bien fundamentada sobre un conjunto A induce una relación bien fundamentada sobre el conjunto de los multiconjuntos finitos cuyos elementos están en A. Este resultado permite usar relaciones bien fundamentadas entre multiconjuntos para demostrar que ciertas funciones definidas recursivamente terminan.

La principal contribución de este capítulo se desarrolla en la segunda sección con la formalización de dicho resultado en la lógica de ACL2 y su prueba automática en el demostrador. Dicho resultado se enuncia de manera general, permitiendo así la posterior definición en el sistema de relaciones bien fundamentadas entre multiconjuntos, inducidas por relaciones bien fundamentadas. Estas relaciones se pueden usar en la prueba de terminación de funciones recursivas definidas en ACL2, en conjunción con una medida adecuada que asocie un multiconjunto a cada llamada recursiva. De esta manera, la terminación de la función queda justificada si se prueba que el multiconjunto asociado a cada una de las llamadas recursivas es menor (respecto de la relación de multiconjuntos

inducida) que el multiconjunto asociado a la llamada original. Esta técnica para demostrar la terminación de algoritmos resulta especialmente útil en algunos casos, donde una prueba directa resultaría considerablemente más complicada.

Con el objetivo de hacer fácil la definición de relaciones bien fundamentadas entre multiconjuntos, hemos desarrollado (mediante una macro) un comando llamado `defmul`, que se presenta en la tercera sección. Además de definir la relación entre multiconjuntos inducida por una relación dada, este comando genera una prueba automática (mediante instanciación funcional) de la buena fundamentación de la relación definida, siempre que la relación original sea bien fundamentada. `defmul` es un buen ejemplo de cómo el demostrador puede ser extendido mediante herramientas específicas que hacen más fácil determinadas tareas rutinarias.

En las secciones cuarta y quinta, hemos ilustrado el uso de multiconjuntos en pruebas de terminación a través de dos interesantes ejemplos, que han sido tomados de [16] y que hemos formalizado y automatizado usando ACL2. El primero es una prueba de la terminación de una versión recursiva de cola de la función de Ackermann. El segundo muestra la terminación de una versión iterativa de la función 91 de McCarthy.

Pero la principal aplicación de esta formalización de las relaciones entre multiconjuntos, que originó el desarrollo aquí presentado, la describiremos en el capítulo siguiente: una demostración en ACL2 del lema de Newman para reducciones abstractas.

5.1 Multiconjuntos: definiciones y propiedades

Mostramos en esta sección la definición de la relación entre multiconjuntos y sus propiedades, tal y como se presentan habitualmente en la literatura sobre el tema. Los resultados de esta sección están tomados de [16].

Definición 5.1 *Un* **multiconjunto** M *sobre un conjunto* A *es una función de* A *en el conjunto de los números naturales. Decimos que* M *es un multiconjunto* **finito** *si hay un número finito de elementos de* A *tal que* $M(x) > 0$*. El conjunto de todos los multiconjuntos finitos sobre* A *lo notaremos por* $\mathcal{M}(A)$*.*

Intuitivamente, la función asigna a cada elemento de A el número de veces que aparece en el multiconjunto, formalizando así la idea de múltiples ocurrencias de un elemento en un conjunto. Usaremos notación conjuntista para representar multiconjuntos finitos, quedando claro en cada contexto si nos referimos a un conjunto o a un multiconjunto. Con esta notación conjuntista, el orden en que aparecen los elementos en su representación no es relevante, pero sí el número de ocurrencias de los mismos.

Ejemplo 5.2 Dado $A = \{a, b, c\}$, un ejemplo de multiconjunto sobre A es $M = \{a, b, b, b\}$, representando la función $M(a) = 1$, $M(b) = 3$ y $M(c) = 0$. El multiconjunto $\{a, b, b, b\}$ es idéntico al multiconjunto $\{b, b, a, b\}$, pero distinto al multiconjunto $\{a, b, b\}$.

Las operaciones y relaciones conjuntistas usuales las podemos generalizar al campo de los multiconjuntos, tomando en consideración las múltiples ocurrencias de los elementos:

Definición 5.3 *Sean* $X, Y \in \mathcal{M}(A)$*. Definimos:*

- $x \in X$*, (*x **pertenece** *a* X *ó* x *es* **elemento** *de* X*) si* $X(x) > 0$*.*

- $X \subseteq Y$, *(X* **incluido** *en Y ó X* **submulticonjunto** *de Y) si para cada $x \in A$ $X(x) \leq Y(x)$.*
- *El multiconjunto* **vacío** *(notado $\emptyset$), como la función que a cada $x \in A$ le asigna 0.*
- *La* **unión** *de X e Y, $X \cup Y$, como la función que a cada $x \in A$ le asigna $X(x)+Y(x)$.*
- *La* **diferencia** *de X e Y, $X \setminus Y$, como la función que a cada $x \in A$ le asigna $X(x) - Y(x)$, si $X(x) \geq Y(x)$ ó 0 en caso contrario.*
- *La* **intersección** *de X e Y, $X \cap Y$, como la función que a cada $x \in A$ le asigna $min\{X(x), Y(x)\}$.*

Ejemplo 5.4 $\emptyset \subseteq \{a, a, a\} \subseteq \{a, a, a, b, c\}$, $\{a, b, c\} \cup \{a, b, c\} = \{a, a, b, b, c, c\}$, $\{a, a, b, c\} \setminus \{a, b, b\} = \{a, c\}$, $\{a, a, b, c\} \cap \{a, c, c\} = \{a, c\}$.

A partir de un orden parcial definido sobre un conjunto A, es posible definir un orden sobre el conjunto de multiconjuntos sobre A: dado un multiconjunto, se obtiene otro más pequeño quitando una serie de elementos y sustituyéndolos por otra serie de elementos tales que cada uno de ellos es más pequeño que algún elemento de los que se han suprimido. Esta construcción puede ser descrita para relaciones binarias en general, como hacemos en la siguiente definición:

Definición 5.5 *Sea $>$ una relación binaria definida sobre un conjunto A. La* **relación entre multiconjuntos** *inducida por $>$ sobre $\mathcal{M}(A)$, notada por $>_{mul}$, se define de la siguiente manera: $M >_{mul} N$ si y sólo si existen $X, Y \in \mathcal{M}(A)$ tal que:*

$$\emptyset \neq X \subseteq M, \quad N = (M \setminus X) \cup Y \ y \ \forall y \in Y \exists x \in X, x > y$$

Ejemplo 5.6 Si $A = \{a, b, c, d, e\}$ y la relación $>$ está definida por $a > b$, $c > d$, entonces $\{a, a, b, c, d, e\} >_{mul} \{a, b, b, b, b, d, d, d, d, d, e\}$, lo cual se justifica reemplazando $X = \{a, c\}$ por $Y = \{b, b, b, d, d, d, d\}$.

Es posible probar que si $>$ es un orden parcial sobre un conjunto A, entonces $>_{mul}$ es un orden parcial sobre $\mathcal{M}(A)$. En tal caso, hablamos del **orden entre multiconjuntos** inducido por $>$.

Definición 5.7 *Una relación $>$ en un conjunto A se dice* **noetheriana** *(o que* **termina***) si no existe una sucesión infinita $\{x_n\}_{n \geq 0}$ de elementos en A tal que $x_i > x_{i+1}$ para todo $i \geq 0$.*

Una propiedad importante de la relación entre multiconjuntos finitos inducida por una relación dada, es que conserva la propiedad de ser noetheriana, tal y como enunciamos en el siguiente teorema.

Teorema 5.8 *(***Dersowitz y Manna***, [16]). Sea $>$ una relación noetheriana sobre un conjunto A. Entonces $>_{mul}$ es noetheriana.*

Demostración:
Sea $\perp \notin A$ un elemento cualquiera y consideremos $A_\perp = A \cup \{\perp\}$. Extendamos la relación $>$ a $A_\perp$, definiendo $a > \perp$ para todo $a \in A$. Es evidente que $>$ es noetheriana sobre $A_\perp$

ya que $>$ lo es sobre A. Supongamos que $>_{mul}$ no es noetheriana sobre $\mathcal{M}(A)$. Entonces existe una sucesión infinita en $\mathcal{M}(A)$ verificando:

$$M_1 >_{mul} M_2 >_{mul} M_3 >_{mul} M_4 >_{mul} \cdots$$

A partir de esta sucesión, podemos construir una sucesión infinita de árboles finitos (véase A.4) con etiquetas en $A_\perp$

$$t_1, t_2, t_3, t_4 \ldots$$

que verifica las siguientes propiedades:

(1) $t_i \subseteq t_{i+1}$.

(2) $t_i \neq t_{i+1}$.

(3) Si $u, v \in dom(t_i)$, $u \neq \epsilon$ y v es hijo de u, entonces $t_i(u) > t_i(v)$.

(4) $M_i = \{t_i(u) : u \text{ hoja de } t_i \text{ y } t_i(u) \neq \perp\}$.

Construyamos la sucesión a la vez que comprobamos las propiedades (1)–(4).

i) Sea $M_1 = \{a_1, \ldots, a_n\}$. Definimos $t_1 = \{(\epsilon, a)\} \cup \{(j, a_j) : 1 \leq j \leq n\}$, donde a es un elemento arbitrario de A (si A es vacío, el teorema se tiene trivialmente). Es evidente que la sucesión t_1 verifica las propiedades (1)–(4).

ii) Si tenemos construida la sucesión $t_1, \ldots, t_i$ cumpliendo las propiedades (1)–(4), construyamos el árbol t_{i+1} de la siguiente manera: puesto que $M_i >_{mul} M_{i+1}$, entonces existen X e Y tal que

$$\emptyset \neq X \subseteq M_i, M_{i+1} = (M_i \setminus X) \cup Y \text{ y } \forall y \in Y \exists x \in X, x > y$$

Para cada $y \in Y$, añadimos un nodo etiquetado con y, haciéndolo hijo de una hoja de t_i etiquetada con un $x \in X$ tal que $x > y$ (existe, ya que $X \subseteq M_i$ y $M_i = \{t_i(u) : u \text{ hoja de } t_i \text{ y } t_i(u) \neq \perp\}$). Además, por cada hoja correspondiente a un elemento de X, añadimos a t_i, un hijo etiquetado con $\perp$. Veamos que las propiedades (1)–(4) se cumplen para la sucesión $t_1, \ldots, t_{i+1}$. (1) y (2) se cumplen ya que t_{i+1} se obtiene de t_i sin cambiar nodos ni etiquetas existentes y además siempre se añaden nuevos nodos ya que $X \neq \emptyset$ y para cada nodo correspondiente a X se añade un hijo etiquetado con $\perp$. (3) también se cumple por que t_i lo cumple por hipótesis y los nodos añadidos para formar t_{i+1} tienen etiquetas menores que las de sus padres. (4) es evidente ya que $M_{i+1} = (M_i \setminus X) \cup Y$ y t_i lo verifica.

Sea $t = \bigcup_{i \geq 1} t_i$. t es un árbol ya que $t_i \subseteq t_{i+1}$ y además es infinito ya que $t_i \subset t_{i+1}$. Se cumple también que t está finitamente ramificado ya que en cada paso se añaden un número finito de nodos (pues estamos tratando con multiconjuntos finitos) y si un nodo pasa a ser hijo de otro en un paso determinado, a éste no se le añaden más hijos en pasos posteriores. Por tanto, aplicando el lema de König (teorema A.34) a t, obtenemos un camino infinito

$$u_0, u_1, u_2, u_3, u_4, \ldots$$

en t. Ahora bien, puesto que u_{i+1} es hijo de u_i, por 3) se tiene que $t(u_i) > t(u_{i+1})$, $i \geq 1$. Por tanto, es posible obtener una sucesión infinita tal que

$$t(u_1) > t(u_2) > t(u_3) > t(u_4) > \cdots$$

de elementos de $A_\perp$, lo cual está en contradicción con el hecho de que $>$ sea noetheriana en $A_\perp$. Concluimos por tanto que $>_{mul}$ es noetheriana sobre $\mathcal{M}(A)$. □

El teorema anterior supone una interesante forma de demostrar la noetherianidad a partir de otras relaciones noetherianas, tal y como mostramos en el siguiente ejemplo.

Ejemplo 5.9 Sea R la relación en $\mathbb{N}^*$ definida de la siguiente manera:

$$R = \{(u(i+1)v, ui \overset{i}{\cdots} iv) : u, v \in \mathbb{N}^*, i \in \mathbb{N}, i > 0\}$$

Se puede asociar un multiconjunto a cada elemento de $\mathbb{N}^*$ mediante la función $\phi : \mathbb{N}^* \to \mathcal{M}(\mathbb{N})$ definida por $\phi(i_1 i_2 \cdots i_n) = \{i_1, i_2, \cdots, i_n\}$. Veamos que ϕ es monótona respecto de R y $>_{mul}$, donde $>$ denota el orden usual entre números naturales.

Supongamos que xRz. Entonces por definición de R, existen $u, v \in \mathbb{N}^*, i \in \mathbb{N}$ tales que $x = u(i+1)v$ y $z = ui \overset{i}{\cdots} iv$. En ese caso,

$$\phi(x) = \phi(u) \cup \{i+1\} \cup \phi(v) >_{mul} \phi(u) \cup \{i, \overset{i}{\ldots}, i\} \cup \phi(v) = \phi(z).$$

Puesto que $>$ es noetheriana, entonces $>_{mul}$ también lo es y por tanto R también es noetheriana, ya que ϕ es monótona.

De manera análoga, las relaciones entre multiconjuntos proporcionan una herramienta para probar terminación de algoritmos, como ilustramos en el siguiente ejemplo, tomado de [16].

Ejemplo 5.10 A continuación describimos un algoritmo *cuenta–nodos* para contar el número de nodos de un árbol binario a. Supongamos, por simplificar, que disponemos del tipo de dato "árbol binario" con el predicado *hoja* (que reconoce si el árbol no tiene subárboles) y con las funciones *izq* y *der* (para obtener los subárboles izquierdo y derecho en caso de que el árbol no sea una hoja). Supondremos también que disponemos de un tipo de dato "pila", con las funciones usuales *apila*, *desapila* y *cima* y con la constante *pilavacia*.

cuenta–nodos(a) =
  $p :=$ *apila*$(a,$ *pilavacia*$)$
  $c := 0$
  mientras $p \neq$ *pilavacia* **hacer**
    $y :=$ *cima*(p)
    $p :=$ *desapila*(p)
    si *hoja*(y), **hacer** $c := c + 1$
    si no, **hacer** $p :=$ *apila*$($*izq*$(y),$ *apila*$($*der*$(y), p))$
  fin-mientras
  devolver c

Obsérvese cómo la pila almacenada en p puede tanto aumentar como disminuir su número de elementos, por lo que una prueba simple basada en la longitud de p no sirve para probar la parada del bucle **mientras**. Veamos cómo un orden entre multiconjuntos soluciona el problema.

En primer lugar, es posible probar que la relación de "subárbol" es una relación noetheriana entre árboles (considerando todo árbol mayor que sus subárboles). Por tanto, la correspondiente relación inducida entre multiconjuntos finitos de árboles es noetheriana. Así, podemos asignar a cada pila de árboles el multiconjunto de los árboles que contiene.

De esta manera, a cada iteración del bucle **mientras** se le puede asociar el multiconjunto correspondiente a los árboles de la pila almacenada en la variable p. Es fácil comprobar que dicho multiconjunto decrece en cada iteración, respecto del orden entre multiconjuntos. Esto no puede ocurrir infinitas veces y por tanto el algoritmo ha de terminar.

Este tipo de razonamiento informal en el que se usan multiconjuntos para probar la terminación de algoritmos, puede ser formalizado y automatizado en ACL2, como mostraremos en la sección 5.2.

Antes de acabar esta sección, veamos un resultado que será de utilidad en la formalización de las relaciones entre multiconjuntos que se presenta en la siguiente sección. La definición 5.5 de relación entre multiconjuntos, aunque bastante intuitiva, es complicada para usarla como base de un algoritmo que calcule la relación. En el caso de órdenes parciales (estrictos) es posible encontrar una definición equivalente, lo que supone una alternativa más simple desde el punto de vista de la implementación:

Teorema 5.11 *Sea $>$ un orden parcial sobre un conjunto A y $M, N \in \mathcal{M}(A)$. Entonces:*

$$M >_{mul} N \iff M \setminus N \neq \emptyset \quad \wedge \quad \forall y \in N \setminus M, \exists x \in M \setminus N, x > y$$

Demostración:

$\boxed{\Longrightarrow}$ Supongamos que $M >_{mul} N$. En ese caso, existen X e Y en las condiciones de la definición de $>_{mul}$.

1. Veamos en primer lugar que $M \setminus N$ es no vacío. Si no fuera así, $M \subseteq N$ y por tanto $M \subseteq (M \setminus X) \cup Y$. Veamos que esto implica que $X \subseteq Y$. Para ello, sea $x \in A$. Aplicando las definiciones de inclusión, diferencia y unión multiconjuntista, obtenemos que $M(x) \leq (M(x) - X(x)) + Y(x)$ y por tanto $X(x) \leq Y(x)$, para todo $x \in A$, lo cual implica $X \subseteq Y$. Puesto que $\forall y \in Y \exists x \in X$ tal que $x > y$, en particular $\forall y \in X \exists x \in X$ tal que $x > y$. Al ser $>$ es un orden parcial, esto implicaría que X es un multiconjunto infinito, lo cual es imposible.

2. Para probar la segunda propiedad, observemos que $N \setminus M = ((M \setminus X) \cup Y) \setminus M = ((M \cup Y) \setminus X) \setminus M = ((M \cup Y) \setminus M) \setminus X = Y \setminus X$ y que $M \setminus N = M \setminus ((M \setminus X) \setminus Y) = (M \setminus (M \setminus X)) \setminus Y = X \setminus Y$. Dichas igualdades se obtienen rutinariamente desarrollando las definiciones de diferencia y unión multiconjuntista y de las propiedades asumidas para X e Y, como en el apartado anterior.

   Sea ahora $y_1 \in N \setminus M = Y \setminus X$. Puesto que $y_1 \in Y$, entonces existe $y_2 \in X$ tal que $y_2 > y_1$. Si $y_2 \in X \setminus Y = M \setminus N$, ya está. En caso contrario $y_2 \in X \cap Y$, en cuyo caso repetimos el razonamiento, ahora con y_2, para obtener $y_3 \in X$ tal que $y_3 > y_2$. Al ser $>$ es un orden parcial y $X \cap Y$ es finito, es imposible que exista una sucesión infinita $y_1 < y_2 < y_3 < \ldots$. Es decir, debe existir un $y_n \in X \setminus Y = M \setminus N$ tal

que $y_n > \ldots > y_2 > y_1$. Puesto que $>$ tiene la propiedad transitiva, tenemos que $y_n > y_1$.

$\boxed{\Longleftarrow}$ Basta con tomar $X = M \setminus N$ e $Y = N \setminus M$, ya que $(M \setminus (M \setminus N)) \cup (N \setminus M) = N$ (nuevamente, desarrollando las definiciones). Nótese que esta implicación es cierta aunque la relación no sea un orden parcial. □

Desde el punto de vista de la implementación, el interés que tiene esta caracterización es que no es necesario buscar los multiconjuntos X e Y de la definición primitiva: usamos $M \setminus N$ y $N \setminus M$, respectivamente. Sin embargo, debemos apuntar que esta definición alternativa puede ser más restrictiva cuando la relación no es un orden parcial, como muestran los siguientes ejemplos:

Ejemplo 5.12

1. Sea R una relación en $A = \{a, b\}$ tal que aRb y bRa, entonces $\{a, b\}R_{mul}\{a, b\}$, tomando $X = Y = \{a, b\}$. Sin embargo $\{a, b\} \setminus \{a, b\} = \emptyset$.

2. Sea S definida en $A = \{a, b, c, d\}$, tal que aSb, bSc y cSd. En este caso tenemos $\{a, b, c\}S_{mul}\{b, c, d\}$, tomando $X = \{a, b, c\}$ e $Y = \{b, c, d\}$. Sin embargo, $d \in \{b, c, d\} \setminus \{a, b, c\}$, $\{a\} = \{a, b, c\} \setminus \{b, c, d\}$ y no es cierto que aSd.

No obstante, la restricción de que la relación original deba ser un orden parcial no es demasiado severa, ya que en nuestro caso aplicaremos la definición alternativa a relaciones noetherianas, que siempre cumplirán la propiedad irreflexiva y en la mayoría de los casos la propiedad transitiva. Incluso cuando no es así, la clausura transitiva de la relación es un orden parcial que conserva la noetherianidad. En cualquier caso, si se toma esta definición alternativa como definición original, al ser una restricción de la presentada en 5.5, el teorema 5.8 también se tiene, proporcionando de igual manera una forma de generar relaciones noetherianas entre multiconjuntos. Estas consideraciones nos llevan a tomar esta definición alternativa para formalizar en ACL2 el teorema 5.8.

5.2 Formalización

En esta sección mostramos cómo formalizar en la lógica de ACL2 los conceptos sobre multiconjuntos que se han descrito en la sección anterior, así como el resultado principal. Este resultado, una vez probado, constituye una útil herramienta para la automatización de pruebas no triviales en ACL2 de terminación de funciones recursivas. Los resultados de esta sección se encuentran en el libro `multiset.lisp`. En dicho libro, los símbolos que se definen pertenecen al paquete `MUL`.

5.2.1 Relaciones noetherianas

El primer paso en nuestra formalización consiste en expresar, dentro de la lógica de ACL2, el concepto de relación noetheriana. Es posible expresar en ACL2 un concepto restringido de noetherianidad, basado en el siguiente meta–teorema[1]: una relación $\rhd$ en un conjunto A es noetheriana si y sólo si existe una función $F : A \to Ord$ tal que $x \lhd y \Longrightarrow F(x) < F(y)$,

[1] En el apéndice A de [19] se encuentra una demostración del mismo.

donde $<$ denota en este caso el orden usual entre ordinales. Como se describió en la subsección 2.3.1, las relaciones ACL2 para las cuales se ha probado la existencia de dicha función F se denominan *bien fundamentadas*[2].

El concepto de relación bien fundamentada es un concepto clave en la lógica de ACL2. Esto se debe, por un lado, a que las relaciones bien fundamentadas se usan en la admisión de funciones recursivas mediante el principio de definición. Por otro lado, justifican la aplicación de la regla de inducción en la prueba de teoremas en los que están involucradas funciones recursivas. Debido a esto, el demostrador ofrece un soporte especial para el tratamiento de estas relaciones, una vez que se haya probado su buena fundamentación y declarado como tal usando el tipo de regla `well-founded-relation`.

Lo anteriormente expuesto nos lleva a formalizar en ACL2 las relaciones noetherianas a través del concepto de relación bien fundamentada. Podemos definir en ACL2 una relación bien fundamentada arbitraria `rel` definida sobre un conjunto de objetos que satisfacen una propiedad `mp`, de la siguiente manera:

```
(encapsulate
 ((mp (x) booleanp) (rel (x y) booleanp) (fn (x) e0-ordinalp))
 ...
 (defthm rel-well-founded-relation-on-mp
   (and (implies (mp x) (e0-ordinalp (fn x)))
        (implies (and (mp x)
                      (mp y)
                      (rel x y))
                 (e0-ord-< (fn x) (fn y))))
   :rule-classes :well-founded-relation))
```

En este caso, el predicado `mp` sirve para definir el conjunto de objetos (llamados *medidas*) que están ordenados de una manera bien fundamentada mediante `rel`. La función `fn` (llamada función de *inmersión*) es una función monótona que hace corresponder un ordinal a cada objeto de medida. Es decir, `rel`, `mp` y `fn` representan, respectivamente, a $\lhd$, A y F del meta–teorema anterior. En general, llamamos al teorema `rel-well-founded-relation-on-mp`, el *teorema de buena fundamentación* correspondiente a `rel`, `mp` y `fn`. Una vez que se haya probado (o asumido con `encapsulate`) un teorema de este tipo para una relación dada y que se haya almacenado como regla de tipo `well-founded-relation`, ésta puede ser usada en la prueba de terminación de funciones recursivas, e indirectamente justificar como correcto el esquema de inducción que sugieren tales funciones recursivas. En ACL2, toda relación bien fundamentada tiene que ser descrita mediante tres funciones (la relación propiamente dicha, un predicado que reconozca los objetos de medida y una función de inmersión) y mediante el correspondiente teorema de buena fundamentación. Como un caso particular, cuando `mp` es la función con valor constante igual a `t`, es posible omitir las referencias a `mp` en el enunciado del correspondiente teorema de buena fundamentación.

La formalización de relaciones noetherianas que hemos discutido es restrictiva en cierto sentido: puesto que sólo los ordinales hasta ε_0 están formalizados en la lógica de ACL2, el tipo ordinal maximal de las relaciones noetherianas que se pueden formalizar no es mayor

[2]El meta-teorema, junto con el teorema A.11, justifica el uso de este término.

que ε_0. Por tanto, la formalización del teorema 5.8 que presentamos tiene la misma restricción. Sin embargo, las demostraciones que hemos realizado no dependen de propiedades particulares de ε_0, excepto de su buena fundamentación.

5.2.2 Multiconjuntos. Relación entre multiconjuntos

Puesto que el lenguaje de la lógica de ACL2 es un subconjunto de Common Lisp, el razonamiento sobre listas y su estructura recursiva se maneja especialmente bien en el demostrador. Es por este motivo por el que representamos un multiconjunto finito en ACL2 como una lista propia. En concreto, dado un predicado `mp` que describe un conjunto A, definimos los multiconjuntos finitos sobre A mediante la siguiente función:

```
(defun mp-true-listp (l)
  (if (atom l)
      (equal l nil)
      (and (mp (car l)) (mp-true-listp (cdr l)))))
```

Nótese que esta función depende de la definición particular del predicado `mp`. Con esta representación, se tiene que distintas listas propias pueden representar el mismo multiconjunto. En concreto, dos listas propias representan el mismo multiconjunto si y sólo si una es permutación de la otra. Por tanto, el orden en que aparecen los elementos en una lista no es relevante desde el punto de vista de los multiconjuntos, aunque sí lo es el número de ocurrencias de cada elemento. Debemos tener esto en cuenta, por ejemplo, al definir la operación de diferencia entre multiconjuntos:

```
(defun remove-one (x l)
  (cond ((atom l) l)
        ((equal x (car l)) (cdr l))
        (t (cons (car l) (remove-one x (cdr l))))))

(defun multiset-diff (m n)
  (if (atom n)
      m
      (multiset-diff (remove-one (car n) m) (cdr n))))
```

Para implementar una definición de la relación entre multiconjuntos inducida por una relación dada, usaremos la definición alternativa, más restrictiva, presentada en el teorema 5.11. Ya se comentó al final de la sección 5.1 las razones por las que esta restricción no es importante en la práctica. Esta alternativa es más fácil de implementar e igualmente produce una relación noetheriana cuando la relación original lo es, como probaremos usando ACL2. Así, dada una relación binaria `rel` previamente definida (o introducida mediante un encapsulado), la relación `mul-rel` entre multiconjuntos inducida por `rel` se define de la siguiente manera:

```
(defun exists-rel-bigger (x l)
  (cond ((atom l) nil)
        ((rel x (car l)) t)
        (t (exists-rel-bigger x (cdr l)))))
```

```
(defun forall-exists-rel-bigger (n m)
  (if (atom n)
      t
      (and (exists-rel-bigger (car n) m)
           (forall-exists-rel-bigger (cdr n) m))))

(defun mul-rel (n m)
  (let ((m-n (multiset-diff m n))
        (n-m (multiset-diff n m)))
    (and (consp m-n) (forall-exists-rel-bigger n-m m-n))))
```

Como era de esperar, tanto la función `mul-rel` como las auxiliares `exists-rel-bigger` y `forall-exists-rel-bigger` dependen de la función `rel`.

5.2.3 Buena fundamentación de las relaciones entre multiconjuntos

Veamos ahora la formalización de un teorema análogo a 5.8, resultado principal de los que presentamos en este capítulo, en el cual se afirma la noetherianidad de `mul-rel`. Como ya se ha comentado en 5.2.1, para establecer la noetherianidad de una relación en ACL2, hemos de probar el teorema correspondiente de buena fundamentación. Por tanto, además de la relación, `mul-rel` en este caso, tenemos que proporcionar una función de inmersión y un predicado de medida adecuados, para probar a continuación el teorema de buena fundamentación correspondiente.

Puesto que `mul-rel` está pensada como relación binaria definida entre multiconjuntos finitos cuyos elementos cumplen `mp`, el predicado que define los objetos de medida es en este caso `mp-true-listp`. Supongamos por el momento que tenemos definida una función de inmersión adecuada, que llamaremos `map-fn-e0-ord`. Así, el teorema 5.8 se enuncia en la lógica de ACL2 mediante la siguiente fórmula:

```
(defthm multiset-extension-of-rel-well-founded
  (and (implies (mp-true-listp x)
                (e0-ordinalp (map-fn-e0-ord x)))
       (implies (and (mp-true-listp x)
                     (mp-true-listp y)
                     (mul-rel x y))
                (e0-ord-< (map-fn-e0-ord x)
                          (map-fn-e0-ord y))))
  :rule-classes :well-founded-relation)
```

Nótese que este resultado es lo suficientemente general como para que, mediante instanciación funcional, se pueda deducir la buena fundamentación de cualquier relación entre multiconjuntos construida a partir de una relación bien fundamentada (ya que no se ha asumido ninguna propiedad específica sobre `rel`, excepto aquellas que establecen su buena fundamentación). Por tanto, se puede usar para generar fácilmente relaciones entre multiconjuntos con la propiedad de buena fundamentación, como se discutirá en la sección 5.3.

En la subsección siguiente, se define `map-fn-e0-ord`, una función de inmersión adecuada, que hace que la fórmula anterior sea un teorema en la lógica de ACL2. Además se describe una prueba informal de este teorema, la cual ha servido para conducir al demostrador hasta la prueba formal.

5.2.4 Una prueba informal del teorema

La prueba del teorema 5.8 presentada en la sección 5.1, basada en el lema de König, es la que aparece originalmente en [16] y es la que habitualmente se encuentra en la literatura. Sin embargo, la formulación del resultado en ACL2 es distinta y por tanto la prueba también ha de serlo. En nuestro caso debemos definir una función monótona que asigne a cada lista propia de elementos que cumplen `mp`, un objeto ACL2 que represente un ordinal menor que ε_0.

Nuestra prueba está basada en el siguiente resultado de la teoría de ordinales (véase [28]): dado un ordinal α, el conjunto $\mathcal{M}(\alpha)$ de los multiconjuntos finitos de elementos de α (ordinales menores que α), ordenado mediante la relación entre multiconjuntos inducida por el orden usual entre ordinales, es isomorfo al ordinal ω^α y el isomorfismo viene dado por la función H tal que $H(\{\beta_1, \ldots, \beta_n\}) = \omega^{\beta_1} + \ldots + \omega^{\beta_n}$. Este resultado se prueba usando la forma normal de Cantor y sus propiedades.

Como subproducto, podemos concluir una interesante propiedad acerca de relaciones bien fundamentadas entre multiconjuntos. Puesto que $\alpha \leq \varepsilon_0$ implica que $\omega^\alpha \leq \omega^{\varepsilon_0} = \varepsilon_0$, esto significa que siempre es posible probar, dentro de la lógica de ACL2, la buena fundamentación de la relación entre multiconjuntos inducida por una relación bien fundamentada en ACL2 (es decir, siempre es posible su inmersión en ε_0). Esto no ocurre con el producto lexicográfico, ya que el tipo ordinal maximal del producto lexicográfico de dos relaciones definidas en ACL2 como bien fundamentadas puede ser mayor que ε_0 y por tanto no se podría probar en ACL2 el correspondiente teorema de buena fundamentación.

El isomorfismo anterior H sugiere la siguiente definición de la función `map-fn-e0-ord`: dado un multiconjunto de elementos que cumplen `mp`, aplíquese `fn` a cada elemento para obtener un multiconjunto de ordinales. Luego aplíquese H para obtener un ordinal menor que ε_0. Si los ordinales están representados en notación ACL2 (véase la sección 2.2), entonces la función H se puede definir fácilmente, siempre y cuando la función `fn` nunca devuelva `0`: basta con ordenar los ordinales del multiconjunto y añadir `0` como `cdr` final. Nótese que la restricción sobre `fn` puede ser superada sin dificultad, definiendo (la macro) `fn1` igual a `fn`, excepto para los números enteros, en cuyo caso se añade 1. De esta manera, `fn1` siempre devuelve ordinales distintos de cero para cualquier objeto que cumpla la propiedad `mp` y es monótona si y sólo si `fn` lo es. Las siguientes definiciones implementan las ideas expuestas:

```
(defun insert-e0-ord-< (x l)
  (cond ((atom l) (cons x l))
        ((not (e0-ord-< x (car l)))  (cons x l))
        (t (cons (car l) (insert-e0-ord-< x (cdr l))))))

(defun add1-if-integer (x) (if (integerp x) (1+ x) x))

(defmacro fn1 (x) `(add1-if-integer (fn ,x)))
```

```
(defun map-fn-e0-ord (l)
  (if (consp l)
      (insert-e0-ord-< (fn1 (car l)) (map-fn-e0-ord (cdr l)))
    0))
```

Una vez se ha definido la función `map-fn-e0-ord`, veamos un boceto de la prueba ACL2 del teorema de buena fundamentación para `mul-rel`, `mp-true-listp` y `map-fn-e0-ord`, tal y como se enuncia al final de la subsección anterior por `multiset-extension-of-rel--well-founded`. Nos centraremos en la parte más complicada del teorema, que afirma la monotonía de `map-fn-e0-ord`.

Prueba informal:

Notemos, por simplificar, las funciones `fn1` y `map-fn-e0-ord` como f y f_{mul}, y las relaciones `rel`, `mul-rel` y `e0-ord-<` como $\lhd$, $\lhd_{mul}$ and $<$, respectivamente. Sean M y N dos multiconjuntos, cuyos elementos satisfacen `mp`, tales que $N \lhd_{mul} M$. Hemos de probar que $f_{mul}(N) < f_{mul}(M)$ y lo haremos por inducción en el número de elementos de N. Obsérvese que M no puede ser vacío y que si N es vacío el resultado se cumple trivialmente. Por tanto, supongamos que M y N no son vacíos. Sean $u \in M$ y $v \in N$ tales que $f(u)$ y $f(v)$ son los mayores elementos (respecto del orden entre ordinales) de los multiconjuntos $f[N]$ y $f[M]$, respectivamente. Nótese que $f(u)$ y $f(v)$ son los resultados obtenidos de aplicar la función `car` a las listas $f_{mul}(N)$ y $f_{mul}(M)$, respectivamente. Puesto que $f(u)$ y $f(v)$ son ordinales, existen, en principio, tres posibilidades:

1. $f(u) < f(v)$. Entonces por definición de $<$, se tiene $f_{mul}(N) < f_{mul}(M)$.

2. $f(v) > f(u)$. Esto no es posible, ya que en ese caso $v \in N \setminus M$ y por definición de $\lhd_{mul}$, existe $z \in M \setminus N$ tal que $v \lhd z$. Por tanto $f(z) > f(v) > f(u)$. Esto está en contradicción con el hecho de que $f(u)$ sea el mayor elemento de $f[M]$.

3. $f(v) = f(u)$. En ese caso, $v \in M$, ya que en otro caso existiría $z \in M \setminus N$ tal que $v \lhd z$ y se tendría la misma contradicción que en el caso anterior. Sean $M' = M \setminus \{u\}$ y $N' = N \setminus \{v\}$. Por definición de $\lhd_{mul}$, se tiene que $N' \lhd_{mul} M'$. Además $f_{mul}(N')$ y $f_{mul}(M')$ son los resultados de aplicar la función `cdr` a las listas $f_{mul}(N)$ y $f_{mul}(M)$, respectivamente. Podemos aplicar la hipótesis de inducción para N' y M' para concluir que $f_{mul}(N') < f_{mul}(M')$ y por tanto $f_{mul}(N) < f_{mul}(M)$.

□

En la sección siguiente mostramos los principales lemas que guían al demostrador automático de ACL2 para realizar una prueba formal del teorema anterior, siguiendo las directrices de la prueba informal que se acaba de presentar.

5.2.5 Descripción de la demostración

El teorema `multiset-extension-of-rel-well-founded` consta de dos partes bien diferenciadas. La primera de ellas, que afirma que la función `map-fn-e0-ordinal` devuelve siempre un objeto que verifica `e0-ordinalp`, no entraña dificultad y se prueba en ACL2 casi sin ayuda por parte del usuario (véase `2.5.1` del libro `multiset.lisp`). La parte más

difícil es aquella que afirma la monotonía de `map-fn-e0-ordinalp` respecto de `mul-rel`. Nos centraremos en comentar la prueba automática de este resultado.

En primer lugar definimos la función `(max-fn1-list l)` que calcula un elemento `x` de `l` con el máximo valor de `(fn1 x)`:

```
(defun max-fn1-list (l)
  (cond ((atom l) nil)
        ((atom (cdr l)) (car l))
        (t (let ((max-cdr (max-fn1-list (cdr l))))
             (if (e0-ord-< (fn1 max-cdr) (fn1 (car l)))
                 (car l)
                 max-cdr))))))
```

Nótese que los elementos u y v a los que se alude en la prueba informal son, `(max-fn1-list m)` y `(max-fn1-list n)`, respectivamente. Es de esperar que `max-fn1-list` cumpla las propiedades deseadas. Es decir, si `l` es un multiconjunto no vacío, entonces `(max-fn1-list l)` debe ser un elemento de `l` con valor máximo de `fn1`. Efectivamente es así:

```
(defthm max-fn1-list-member
  (implies (consp l)
           (member (max-fn1-list l) l)))

(defthm max-fn1-list-maximal
  (implies (member x l)
           (not (e0-ord-< (fn1 (max-fn1-list l)) (fn1 x)))))
```

Siguiendo el boceto de la prueba, expuesto en la subsección 5.2.4, hemos de realizar una demostración por inducción. Como era de esperar, un esquema de inducción tan particular no es generado por el sistema a menos que se le indique expresamente. La siguiente función recursiva servirá para generar el esquema de inducción buscado:

```
(defun induction-multiset (n m)
  (declare (xargs :measure (ACL2::len n)))
  (cond ((atom n) (if (atom m) 1 2))
        ((atom m) 3)
        (t (let* ((max-m (max-fn1-list m))
                  (max-n (max-fn1-list n))
                  (fn1-max-m (fn1 max-m))
                  (fn1-max-n (fn1 max-n)))
             (cond ((equal fn1-max-m fn1-max-n)
                    (if (member max-n m)
                        (induction-multiset
                         (remove-one max-n n)
                         (remove-one max-n m))
                        5))
                   ((e0-ord-< fn1-max-n fn1-max-m) 6)
                   ((e0-ord-< fn1-max-m fn1-max-n) 7)
                   (t 8)))))))
```

Obsérvese que la admisión de esta función necesita especificar una medida que ayude a probar su terminación. En este caso, la medida es `(ACL2::len n)`[3], lo cual justifica formalmente la afirmación realizada en la prueba informal de que se trata de una prueba "por inducción en el número de elementos de N". Nótese que en la prueba de admisión de la función es necesario el teorema `max-fn1-list-member`. Los valores concretos que toma esta función son totalmente irrelevantes. Lo importante es que esta función sirve para comunicar al sistema de un esquema de inducción concreto. Su única llamada recursiva proporciona el paso de inducción. El resto de llamadas proporcionan los casos base. Un consejo de tipo `:induct` nos sirve para comunicar al sistema que intente una prueba del teorema usando el esquema de inducción sugerido por la función que se indica:

```
(defthm map-fn-e0-ord-measure
  (implies (and (mp-true-listp n)
                (mp-true-listp m)
                (mul-rel n m))
           (e0-ord-< (map-fn-e0-ord n)
                     (map-fn-e0-ord m)))
    :hints (("Goal" :induct (induction-multiset n m))))
```

El esquema de inducción con el que se intenta esta conjetura, sugerido por la función `induction-multiset`, aparece en la figura 5.1[4]. Hemos numerado los casos que aparecen en él, de la misma forma que lo hace el demostrador. Recuérdese que `(:P M N)` simboliza la propiedad que se quiere demostrar, en este caso la fórmula del teorema `map-fn-e0-ord-measure`. Dicho esquema de inducción hace que se consideren ocho casos, mutuamente excluyentes, en la prueba del teorema. Sólo uno de ellos, el `*1/4`, es un caso inductivo, en el que se puede asumir como hipótesis de inducción la propiedad `:P` para `(remove-one (max-fn1-list n) m)` y `(remove-one (max-fn1-list n) n)`. Algunos de los casos son ciertos porque las condiciones que los describen son contradictorias, bien por sí mismas o bien en conjunción con las hipótesis de `:P`. Veamos qué ocurre caso por caso.

El caso `*1/8` es imposible. En primer lugar, téngase en cuenta que estamos tratando con ordinales:

```
(defthm max-fn1-e0-ordinalp
  (implies (and (consp l) (mp-true-listp l))
           (e0-ordinalp (fn1 (max-fn1-list l))))))
```

Además, el orden entre los ordinales de ACL2 es un orden total, tal y como afirma el siguiente teorema:

```
(defthm e0-ord-<-trichotomy
  (implies (and (e0-ordinalp o1)
                (e0-ordinalp o2)
                (not (equal o1 o2))
                (not (e0-ord-< o1 o2)))
           (e0-ord-< o2 o1)))
```

[3] Recuérdese que los símbolos del libro `multiset` se definen dentro del paquete `MUL`, de ahí que necesitemos especificar el paquete en el que se define la función `len`.

[4] Para mejorar la legibilidad, hemos mantenido la macro `fn1` sin expandir.

```
(AND (IMPLIES (AND (NOT (ATOM N))                                        ;;; *1/8
                   (NOT (ATOM M))
                   (NOT (EQUAL (FN1 (MAX-FN1-LIST M)) (FN1 (MAX-FN1-LIST N))))
                   (NOT (E0-ORD-< (FN1 (MAX-FN1-LIST N)) (FN1 (MAX-FN1-LIST M))))
                   (NOT (E0-ORD-< (FN1 (MAX-FN1-LIST M)) (FN1 (MAX-FN1-LIST N)))))
              (:P M N))

     (IMPLIES (AND (NOT (ATOM N))                                        ;;; *1/7
                   (NOT (ATOM M))
                   (NOT (EQUAL (FN1 (MAX-FN1-LIST M)) (FN1 (MAX-FN1-LIST N))))
                   (NOT (E0-ORD-< (FN1 (MAX-FN1-LIST N)) (FN1 (MAX-FN1-LIST M))))
                   (E0-ORD-< (FN1 (MAX-FN1-LIST M)) (FN1 (MAX-FN1-LIST N))))
              (:P M N))

     (IMPLIES (AND (NOT (ATOM N))                                        ;;; *1/6
                   (NOT (ATOM M))
                   (NOT (EQUAL (FN1 (MAX-FN1-LIST M)) (FN1 (MAX-FN1-LIST N))))
                   (E0-ORD-< (FN1 (MAX-FN1-LIST N)) (FN1 (MAX-FN1-LIST M))))
              (:P M N))

     (IMPLIES (AND (NOT (ATOM N))                                        ;;; *1/5
                   (NOT (ATOM M))
                   (EQUAL (FN1 (MAX-FN1-LIST M)) (FN1 (MAX-FN1-LIST N)))
                   (NOT (MEMBER (MAX-FN1-LIST N) M)))
              (:P M N))

     (IMPLIES (AND (NOT (ATOM N))                                        ;;; *1/4
                   (NOT (ATOM M))
                   (EQUAL (FN1 (MAX-FN1-LIST M)) (FN1 (MAX-FN1-LIST N)))
                   (MEMBER (MAX-FN1-LIST N) M)
                   (:P (REMOVE-ONE (MAX-FN1-LIST N) M)
                       (REMOVE-ONE (MAX-FN1-LIST N) N)))
              (:P M N))

     (IMPLIES (AND (NOT (ATOM N)) (ATOM M))                              ;;; *1/3
              (:P M N))

     (IMPLIES (AND (ATOM N) (NOT (ATOM M)))                              ;;; *1/2
              (:P M N))

     (IMPLIES (AND (ATOM N) (ATOM M))                                    ;;; *1/1
              (:P M N)))
```

Figura 5.1: Esquema de inducción generado por `induction-multiset`

Los dos teoremas anteriores nos permiten deducir que en el caso `*1/8`, el teorema es trivialmente cierto, ya que no se pueden dar tales condiciones, que implicarían la existencia de dos ordinales no comparables respecto de `e0-ord-<`.

El caso `*1/7` tampoco es posible, esta vez en conjunción con las hipótesis del teorema. Usando la notación de la prueba informal presentada en la subsección anterior, bastaría con probar que bajo las hipótesis del teorema, no es posible que $f(u) < f(v)$ (es decir, el caso 2 de la prueba informal).

Para ello, nótese que si `m` y `n` son dos multiconjuntos no vacíos tal que `(mul-rel n m)`,

entonces cualquier elemento `x` de `n` que no esté en `m` tiene asignado por `fn1` un valor estrictamente menor que el elemento de `m` con mayor valor de `fn1`:

```
(defthm forall-exists-rel-bigger-max-fn1-list-lemma
  (implies (and (consp m)
                (mp-true-listp n)
                (mp-true-listp m)
                (member x n)
                (not (member x m))
                (forall-exists-rel-bigger (multiset-diff n m)
                                          (multiset-diff m n)))
           (e0-ord-< (fn1 x) (fn1 (max-fn1-list m)))))
```

Si, por el contrario, `x` está en `m`, su valor de `fn1` no puede ser mayor que el elemento de `m` con mayor valor de `fn1`, como ya establece `max-fn1-list-maximal`. Esta propiedad y la anterior, permiten deducir, para el caso particular de que `x` sea `(max-fn1-list n)`, el siguiente resultado: si `m` y `n` son dos multiconjuntos no vacíos tales que `(mul-rel n m)`, entonces el valor máximo de `fn1` en `n` no puede ser mayor que el valor máximo de `fn1` en `m` (lo cual descarta que el caso `*1/7` se pueda producir).

```
(defthm forall-exists-rel-bigger-max-fn1-list
  (implies (and (consp n)
                (consp m)
                (mp-true-listp n)
                (mp-true-listp m)
                (forall-exists-rel-bigger (multiset-diff n m)
                                          (multiset-diff m n)))
           (not (e0-ord-< (fn1 (max-fn1-list m)) (fn1 (max-fn1-list n))))))
```

El caso `*1/6` se corresponde con el caso 1 de la prueba informal (es decir, $f(u) > f(v)$). Como se afirma en ella, el resultado se tiene "por definición de $<$", donde $<$ denota el orden `e0-ord-<`. Esto es así porque, en el caso de que los ordinales que compara `e0-ord-<` vengan representados por listas (como en este caso) entonces se comparan recursivamente los primeros elementos respectivos y si éstos son iguales, se comparan los restantes. Y como se afirma nuevamente en la prueba informal, *"$f(u)$ y $f(v)$ son los resultados de aplicar la función* `car` *a las listas $f_{mul}(N)$ y $f_{mul}(M)$"*. Además *"$f_{mul}(N')$ y $f_{mul}(M')$ son los resultados de aplicar la función* `cdr` *a las listas $f_{mul}(N)$ y $f_{mul}(M)$"* (donde $M' = M \setminus \{u\}$ y $N' = N \setminus \{v\}$). Esta dos últimas afirmaciones informales se pueden probar en el demostrador y almacenar como reglas de reescritura:

```
(defthm another-definition-of-map-fn-e0-ord-rewrite-rules
  (implies (and (mp-true-listp l)
                (consp l))
           (and (equal (car (map-fn-e0-ord l)) (fn1 (max-fn1-list l)))
                (equal (cdr (map-fn-e0-ord l))
                       (map-fn-e0-ord (remove-one (max-fn1-list l) l))))))
```

El papel de esta regla de reescritura es fundamental en la prueba del teorema. Por ejemplo, en el caso `*1/6`, el sistema intenta expandir la definición de `e0-ord-<` al intentar

probar `(e0-ord-< (map-fn-e0-ord n) (map-fn-e0-ord m))`. Para ello aplica la función `car` a cada uno de sus argumentos. Dicha regla de reescritura permite simplificar la expresión anterior a `(e0-ord-< (fn1 (max-fn1-list n)) (fn1 (max-fn1-list m)))`, que se encuentra entre las hipótesis. Consúltese `2.5.2` del libro `multiset.lisp` para obtener más detalles sobre estas reglas de reescritura y sobre un intento fallido previo.

El caso `*1/5` no puede ocurrir. Este caso se corresponde con la afirmación en la prueba informal de que si $f(u) = f(v)$, entonces necesariamente $v \in M$. Una simple instanciación del teorema `forall-exists-rel-bigger-max-fn1-list-lemma` (sustituyendo `x` por `(max-fn1-list n)`), nos lleva a dicho resultado, que excluye la posibilidad de que se den las condiciones de `*1/5`:

```
(defthm forall-exists-rel-bigger-max-fn1-list-lemma-corollary
  (implies (and (consp n)
                (consp m)
                (mp-true-listp n)
                (mp-true-listp m)
                (equal (fn1 (max-fn1-list n))
                       (fn1 (max-fn1-list m)))
                (forall-exists-rel-bigger (multiset-diff n m)
                                          (multiset-diff m n)))
           (member (max-fn1-list n) m)))
```

El caso `*1/4` es el único caso inductivo de la prueba. En la prueba informal, consiste en probar el resultado suponiendo $f(u) = f(v)$ y $v \in M$ y asumiendo (como hipótesis de inducción) que el teorema es cierto para $M' = M \setminus \{u\}$ y $N' = N \setminus \{v\}$. Es decir, la hipótesis de inducción para este caso es:

```
(IMPLIES (AND (MP-TRUE-LISTP (REMOVE-ONE (MAX-FN1-LIST N) N))
              (MP-TRUE-LISTP (REMOVE-ONE (MAX-FN1-LIST N) M))
              (MUL-REL (REMOVE-ONE (MAX-FN1-LIST N) N)
                       (REMOVE-ONE (MAX-FN1-LIST N) M)))
         (E0-ORD-< (MAP-FN-E0-ORD (REMOVE-ONE (MAX-FN1-LIST N) N))
                   (MAP-FN-E0-ORD (REMOVE-ONE (MAX-FN1-LIST N) M)))))
```

Para que la conclusión de esta hipótesis se pueda usar, las hipótesis de esta implicación se tienen que cumplir. Las dos primeras se tienen de manera fácil y la tercera se verifica por que estamos suponiendo `(mul-rel n m)`, que es equivalente a `(mul-rel (remove-one n x) (mul-rel (remove-one m x)))`, siempre que `x` sea un elemento tanto de `n` como de `m`. Para que el sistema pueda deducir esta última equivalencia, basta con el siguiente lema (ya que `mul-rel` está definido a partir de diferencias entre multiconjuntos):

```
(defthm multiset-diff-removing-the-same-element
  (implies (and (member x n) (member x m))
           (equal (multiset-diff (remove-one x m)
                                 (remove-one x n))
                  (multiset-diff m n))))
```

Una vez que hemos visto que podemos asumir como cierta la conclusión de la hipótesis de inducción, veamos como se puede usar para probar la conclusión del teorema. Nuevamente, la regla `another-definition-of-map-fn-e0-ord-rewrite-rules` juega un papel fundamental para expandir la definición de `e0-ord-<`. La fórmula `(e0-ord-< (map--fn-e0-ord n) (map-fn-e0-ord m))` se reescribe, precisamente, en la conclusión de la hipótesis de inducción, lo que termina la prueba del caso `*1/4`.

Los casos `*1/3`, `*1/2` y `*1/1` se corresponden con los casos en que al menos uno de los multiconjuntos `m` ó `n` sean vacíos. Estos casos se prueban fácilmente por el sistema, con mínima ayuda por parte del usuario.

5.2.6 Algunos lemas útiles sobre la relación entre multiconjuntos

Además del teorema `multiset-extension-of-rel-well-founded`, el libro `multiset-.lisp` incluye una serie de definiciones y teoremas no locales que proporcionan un conjunto de reglas que automatizan, en cierta medida, el razonamiento sobre la relación entre multiconjuntos. Estas reglas van a ser muy útiles cuando probemos propiedades de terminación de funciones mediante multiconjuntos.

Reglas de reescritura

Las siguientes reglas de reescritura simplifican ciertas expresiones sobre `multiset-diff`:

```
(defthm multiset-diff-true-listp
  (implies (and (true-listp m) (true-listp n))
           (true-listp (multiset-diff m n))))

(defthm multiset-diff-remove-one-not-consp
  (not (consp (multiset-diff (remove-one x m) m))))

(defthm list-multiset-diff-1
  (implies (not (member x m))
           (equal (multiset-diff (list x) m) (list x))))

(defthm list-multiset-diff-2
  (implies (not (member x m))
           (equal (multiset-diff m (list x)) m)))

(defthm multiset-diff-append-1
  (equal (multiset-diff (append m1 m2) (append m1 m3))
         (multiset-diff m2 m3)))
```

Nótese que estas reglas permiten manipular a nivel simbólico expresiones acerca de relaciones entre multiconjuntos. Por ejemplo, la última regla, `multiset-diff-append-1` permite simplificar expresiones del tipo `(mul-rel (append` M_1 M_2`) (append` M_1 M_3`))` a otra más simple como `(mul-rel` M_2 M_3`)`.

La equivalencia `equal-set` y congruencias asociadas

Aunque en un multiconjunto el orden en el que aparecen los elementos no es relevante, la representación que hemos elegido (con listas), hace que dos objetos que representan el mismo multiconjunto no sean necesariamente iguales desde el punto de vista de la lógica (es decir, respecto de `equal`). Por ejemplo, uno podría pensar que la siguiente fórmula, una versión simétrica de `multiset-diff-append-1`, pueda ser un teorema:

```
;(equal (multiset-diff (append m2 m1) (append m3 m1))
;       (multiset-diff m2 m3)))
```

Sin embargo, esto no es cierto. Tomemos, por ejemplo, como `m1`, `m2` y `m3` las listas `'(1 3)`, `'(3 2)` y `nil`, respectivamente. Sin embargo, sí que es cierto que `(multiset-diff (append m2 m1) (append m3 m1))` tiene los mismos elementos que `(multiset-diff m2 m3)`. Respecto de la relación `mul-rel`, esto es más que suficiente: recordemos que en la definición de `mul-rel`, las llamadas a `multiset-diff` son argumentos de los predicados `consp` y `forall-exists-rel-bigger`, y en estos predicados es posible sustituir, sin que afecte a su valor, cada uno de sus argumentos por otros, siempre que representen al mismo conjunto de elementos (ni siquiera es necesario que representen al mismo multiconjunto). Podemos, por tanto, usar reescritura congruente (subsección 2.3.1).

Recuérdese que en la página 50 habíamos definido la función `equal-set`, implementando la igualdad conjuntista entre listas, y la habíamos declarado con `defequiv` como relación de equivalencia. Esto hace que el demostrador automático trate a `equal-set` de manera muy parecida a cómo trata al predicado `equal`. En particular, permite almacenar los teoremas que expresan equivalencias respecto de `equal-set` como reglas de reescritura. Es el caso del siguiente resultado, que motiva la presente discusión:

```
(defthm multiset-diff-append-2
  (equal-set (multiset-diff (append m1 m2)
                            (append m3 m2))
             (multiset-diff m1 m3)))
```

Esta regla de reescritura permite simplificar expresiones del tipo `(mul-rel (append` M_2 M_1`) (append` M_3 M_1`))` a otra más simple como `(mul-rel` M_2 M_3`)`. La única diferencia con respecto a reglas de reescritura con `equal` es que dicha simplificación sólo se realizará sobre expresiones que figuran como argumentos de funciones para las cuales se ha probado previamente que tal simplificación no afecta a su valor. En concreto, las funciones `consp` y `forall-exists-rel-bigger`, como se discutió anteriormente, mantienen su valor si sus argumentos se sustituyen por otros que son iguales desde el punto de vista conjuntista. Es posible declarar tales congruencias mediante `defcong`:

```
(ACL2::defcong equal-set equal (consp l) 1)

(ACL2::defcong equal-set iff (forall-exists-rel-bigger l m) 1)

(ACL2::defcong equal-set iff (forall-exists-rel-bigger l m) 2))
```

Nótese que estas dos últimas congruencias son ciertas para cualquier definición concreta de la relación `rel`. En la sección 5.3 veremos cómo se declaran estas congruencias cada vez que definamos una relación entre multiconjuntos.

A veces, esta conjunción de equivalencia, congruencias y reglas de reescritura permite al demostrador simplificar las diferencias entre multiconjuntos que aparecen en la definición de `mul-rel`. En la sección 6.4 veremos cómo esta regla es de utilidad en la prueba del lema de Newman.

Una regla `:meta` muy útil

Es bastante usual (como se pone de manifiesto en las secciones 5.4 y 5.5), que al usar relaciones entre multiconjuntos para probar la terminación de determinadas funciones, los multiconjuntos asociados a las llamadas recursivas difieran, respecto del multiconjunto asociado a la llamada inicial, en los elementos iniciales de la lista que los representa. Es decir, hay que probar que un multiconjunto de la forma `(list*` x_1 x_2 ... x_k `l)` es menor que otro de la forma `(list*` y_1 y_2 ... y_n `l)`. En dicha prueba sería útil eliminar `l`, la parte final común de ambos multiconjuntos. En concreto, nos sería útil una regla de reescritura de la forma

```
(defthm equal-set-list-multiset-diff
  (equal-set (multiset-diff (list* x1 x2 ... xk l)
                            (list* y1 y2 ... yn l))
             (multiset-diff (list x1 x2 ... xk)
                            (list y1 y2 ... yn)))
```

donde $x_1, \ldots, x_k, y_1, \ldots, y_n$ son variables. De hecho, un resultado de este tipo se puede probar mediante una simple instanciación de `multiset-diff-append-2`. Pero el problema es que necesitaríamos una regla *para cada* valor concreto de n y k. Este problema es un caso típico que puede ser solucionado usando reglas `meta`. Se trata de simplificar expresiones atendiendo a la forma de su representación, previa prueba de que tales transformaciones son correctas desde el punto de vista de la lógica.

Hemos definido una regla `:meta` para efectuar la simplificación que hemos descrito. Dicha regla meta extiende el simplificador del demostrador automático, de manera que sea capaz de efectuar la simplificación anterior sobre expresiones que tengan la forma de `(multiset-diff (list*` $x_1 \ldots x_k$ `l) (list*` $y_1 \ldots y_n$ `l))`. Previamente, hemos tenido que probar que tal transformación mantiene el significado de la expresión. Además, según lo comentado anteriormente, estas simplificaciones sólo se producirán en aquellos argumentos de funciones para los que `equal-set` ha sido declarado como congruencia. El lector puede consultar el libro `multiset.lisp` para obtener detalles sobre la definición de esta regla `:meta`.

5.3 El comando `defmul`

Tal y como se ha descrito en la sección anterior, hemos formalizado y demostrado la buena fundamentación de la relación `mul-rel` en un marco lo más abstracto posible, ya que no se han asumido propiedades particulares de las funciones `rel`, `mp` y `fn`, excepto aquellas necesarias para la buena fundamentación de `rel`. Esto nos permite usar instancias funcionales del teorema `multiset-extension-of-rel-well-founded` para demostrar la buena fundamentación de cualquier relación entre multiconjuntos inducida por una relación bien fundamentada definida previamente en ACL2.

A tal efecto, supongamos que tenemos previamente definida (mediante `defun` o `encapsulate`) una relación $relx$, de la cual se sabe que cumple la propiedad de buena fundamentación sobre un conjunto de objetos de medida que satisfacen la propiedad mpx, justificado por la función de inmersión fnx. Es decir, se ha probado el siguiente teorema (usando las variables x e y) y se ha almacenado como regla de buena fundamentación:

```
(defthm nombre
  (and (implies (mpx x) (e0-ordinalp (fnx x)))
       (implies (and (mpx x)
                     (mpx y)
                     (relx x y))
                (e0-ord-< (fnx x) (fnx y))))
  :rule-classes :well-founded-relation)
```

Para definir en la lógica de ACL2 la relación entre multiconjuntos inducida por la relación $relx$, probar su buena fundamentación y almacenar el teorema con la correspondiente regla de tipo `:well-founded-relation`, hemos de realizar, en principio, los siguientes eventos en ACL2:

- las definiciones que se necesitan para implementar la relación entre multiconjuntos inducida por $relx$: las funciones `exists-`$relx$`-bigger`, `forall-exists-`$relx$`-bigger` y `mul-`$relx$ de manera totalmente análoga a las dadas en la subsección 5.2.2,
- la definición de los multiconjuntos que constituyen los objetos de medida, mpx`--true-listp`,
- la definición de `map-`fnx`-e0-ord`, la función de inmersión de tales multiconjuntos en los ordinales y
- el teorema de buena fundamentación para `mul-`$relx$, mpx`-true-listp` y `map-`fnx`--e0-ord`. Este teorema puede ser probado directamente usando una instanciación funcional del teorema `multiset-extension-of-rel-well-founded`. La sustitución funcional que hay que usar debe asignar a los símbolos `x`, `y`, `rel`, `fn`, `mp`, `exists-rel-bigger`, `forall-exists-rel-bigger`, `mul-rel`, `mp-true-listp`, `map--fn-e0-ord` los correspondientes a la nueva relación definida: x, y, $relx$, fnx, mpx, `exists-`$relx$`-bigger`, `forall-exists-`$relx$`-bigger`, `mul-`$relx$, mpx`-true-listp` y `map-`fnx`-e0-ord`, respectivamente.

En lugar de tener que realizar todos los eventos anteriores cada vez que se necesite definir la relación entre multiconjuntos inducida por una relación bien fundamentada, este proceso se puede automatizar. Hemos definido un comando llamado `defmul` (mediante una macro), como una herramienta que nos proporciona una manera automática de realizar esta tarea. En este caso, para definir la relación (bien fundamentada) `mul-`$relx$, inducida por $relx$, basta con la siguiente llamada a `defmul`:

```
(defmul (relx nombre mpx fnx x y))
```

Esta llamada a `defmul` lleva a cabo todos los eventos anteriores. Además, se definen las congruencias de `forall-exists-`$relx$`-bigger` respecto de `equal-set` en sus dos argumentos, congruencias que pueden ser de utilidad en el razonamiento sobre la relación de

multiconjuntos definida, como se explica en la subsección 5.2.6. Tales eventos se completan con éxito, sin que se requiera ayuda por parte del usuario.

La definición de `defmul` se encuentra en el libro `defmul.lisp`, que debe ser incluido por cualquier libro que use este comando.

En las dos subsecciones siguientes, mostramos cómo formalizar en la lógica de ACL2 dos ejemplos no triviales de pruebas de terminación de funciones recursivas. Ambos casos están basados en sendos ejemplos tomados de [16]. En el primero de ellos usamos multiconjuntos para probar la terminación de una versión recursiva de cola de la función de Ackermann. En el segundo ejemplo, utilizamos la misma técnica para admitir una versión iterativa de la función 91 de McCarthy.

Los dos ejemplos muestran una función cuya terminación se prueba a través de una relación bien fundamentada entre multiconjuntos (definida con `defmul`) y una función de medida. Cuando mostramos la definición de la función por primera vez, su código aparece comentado, como una manera de subrayar que ambas funciones (la relación bien fundamentada y la función de medida) tienen que ser proporcionadas para probar que la función termina.

5.4 Una versión iterativa de la función de Ackermann

Los resultados presentados en esta sección se encuentran en el libro `ackerman.lisp`. Este libro incluye a `defmul.lisp`. Lo que sigue es la definición estándar de la función de Ackermann en la lógica de ACL2:

```
(defun ack (m n)
  (declare (xargs :measure (cons (+ (nfix m) 1) (nfix n))))
  (cond ((zp m) (+ n 1))
        ((zp n) (ack (- m 1) 1))
        (t (ack (- m 1) (ack m (- n 1))))))
```

La admisión de esta definición no presenta mayores problemas: basta proporcionar una medida lexicográfica de sus argumentos. No ocurre lo mismo con el siguiente algoritmo `ack-it`, que utiliza recursión de cola para calcular la función de Ackermann. Nuestro objetivo es probar su terminación en la lógica de ACL2, demostrando además que efectivamente la función que define es igual a `ack`:

```
; (defun ack-it-aux (S z)
;   (if (endp S)
;       z
;     (let ((head (first S))
;           (tail (rest S)))
;       (cond ((zp head) (ack-it-aux tail (+ z 1)))
;             ((zp z) (ack-it-aux (cons (- head 1) tail) 1))
;             (t (ack-it-aux (cons head (cons (- head 1) tail))
;                            (- z 1)))))))

; (defun ack-it (m n) (ack-it-aux (list m) n))
```

La idea del algoritmo que define `ack-it-aux` es simple: en cada iteración se cumple que `(ack-it-aux S z) = (ack` s_k `(ack` s_{k-1} `...` `(ack` s_1 `z)))`, donde `S` es una pila con k elementos, $(s_1 \dots s_k)$. Por tanto, como caso particular, se puede probar que `(ack m n)` es igual a `(ack-it m n)`.

Una prueba de la terminación de `ack-it-aux` es difícil. Obsérvese que en la tercera llamada recursiva, la pila se incrementa de tamaño, mientras que el segundo argumento decrece. Por el contrario, en la primera y segunda llamadas recursivas, el segundo argumento se incrementa, mientras que la pila no aumenta de tamaño.

Sin embargo, una medida multiconjuntista puede servir para probar, de manera fácil, la terminación de `ack-it-aux`, como se muestra en [16]. La siguiente prueba informal muestra la idea principal de la prueba.

Prueba informal: Notemos como S y z a los argumentos de la función `ack-it-aux` y supongamos que $S = (s_1, \dots, s_k)$. Sea ϕ una función definida sobre dichos argumentos, asignándoles un multiconjunto de pares de números naturales definido como sigue: $\phi(S, z) = \{(s_1, z), (s_2 + 1, 0), \dots, (s_k + 1, 0)\}$. Notando como $\prec$ al orden lexicográfico en $\mathbb{N} \times \mathbb{N}$ y aplicando el teorema 5.8, se tiene que $\prec_{mul}$ es una relación bien fundamentada en $\mathcal{M}(\mathbb{N} \times \mathbb{N})$ (ya que $\prec$ lo es en $\mathbb{N} \times \mathbb{N}$, teorema A.13). Por tanto, para probar la terminación de `ack-it-aux` bastará con ver que la medida ϕ decrece, respecto de $\prec_{mul}$, en cada llamada recursiva de la función.

Acorde con la definición de la función, distinguiremos tres posibilidades:

1. $s_1 = 0$: en este caso

   $$\phi((s_2, \dots, s_k), z + 1) = \{(s_2, z + 1), \dots, (s_k + 1, 0)\}$$

   Este multiconjunto es menor que el multiconjunto $\phi(S, z)$, ya que $(s_2 + 1, 0) \succ (s_2, z + 1)$ y además (s_1, z) desaparece.

2. $s_1 \neq 0$ y $z = 0$: entonces

   $$\phi((s_1 - 1, s_2, \dots, s_k), 1) = \{(s_1 - 1, 1), (s_2 + 1, 0), \dots, (s_k + 1, 0)\}$$

   También este multiconjunto es menor que $\phi(S, z)$, ya que $(s_1, z) \succ (s_1 - 1, 1)$.

3. $s_1 \neq 0$ y $z \neq 0$: el multiconjunto asociado a esta llamada recursiva es

   $$\phi((s_1, s_1 - 1, s_2, \dots, s_k), z - 1) = \{(s_1, z - 1), (s_1, 0), \dots, (s_2 + 1, 0), \dots, (s_k + 1, 0)\}$$

   Nuevamente, este multiconjunto es menor que $\phi(S, z)$, ya que $(s_1, z) \succ (s_1, z - 1)$ y $(s_1, z) \succ (s_1, 0)$.

□

Usando la herramienta `defmul`, podemos obtener una prueba ACL2 de la terminación de `ack-it-aux`, basándonos en la anterior prueba informal. En primer lugar, definimos la relación bien fundamentada sobre los pares de números naturales, que llamaremos `rel-ack`. La siguiente secuencia de eventos realiza tal tarea:

```
(defun rel-ack (p1 p2)
  (cond ((< (car p1) (car p2)) t)
        ((= (car p1) (car p2)) (< (cdr p1) (cdr p2)))))

(defun mp-ack (p)
  (and (consp p)
       (integerp (car p)) (>= (car p) 0)
       (integerp (cdr p)) (>= (cdr p) 0)))

(defun fn-ack (p) (cons (+ 1 (car p)) (cdr p)))

(defthm rel-ack-well-founded
  (and (implies (mp-ack x)
                (e0-ordinalp (fn-ack x)))
       (implies (and (mp-ack x) (mp-ack y) (rel-ack x y))
                (e0-ord-< (fn-ack x) (fn-ack y))))
  :rule-classes :well-founded-relation)
```

Ahora es fácil definir la relación (bien fundamentada) entre multiconjuntos de pares de números naturales inducida por `rel-ack`, sin más que realizar la siguiente llamada a `defmul`:

```
(defmul (rel-ack rel-ack-well-founded mp-ack fn-ack x y))
```

Esta llamada a `defmul` provoca que se defina la función `mul-rel-ack`, como relación bien fundamentada, con propiedad de medida `mp-ack-true-listp` y función de inmersión `map-fn-ack-e0-ord`. Ahora podemos usar la relación `mul-rel-ack` como relación bien fundamentada en la prueba de terminación de la función `ack-it-aux`, junto con una función de medida adecuada. La función `measure-ack-it-aux` implementa tal función de medida, siguiendo la idea de la prueba informal anterior (necesitamos para ello la función auxiliar `get-pairs-add1-0`):

```
(defun get-pairs-add1-0 (S)
  (if (endp S)
      nil
    (cons (cons (+ (nfix (car S)) 1) 0)
          (get-pairs-add1-0 (cdr S)))))

(defun measure-ack-it-aux (S z)
  (if (endp S)
      nil
      (cons (cons (nfix (car S)) (nfix z))
            (get-pairs-add1-0 (cdr s)))))
```

Proporcionando `mul-rel-ack` como relación bien fundamentada y `measure-ack-it--aux` como función de medida, se puede obtener una prueba en ACL2 de la terminación de `ack-it-aux`, lo que hace que sea admitida como la definición de una función en la lógica de ACL2:

```
(defun ack-it-aux (S z)
  (declare (xargs :measure (measure-ack-it-aux S z)
                  :well-founded-relation mul-rel-ack))
  (if (endp S)
      z
    (let ((head (first S))
          (tail (rest S)))
      (cond ((zp head) (ack-it-aux tail (+ z 1)))
            ((zp z) (ack-it-aux (cons (- head 1) tail) 1))
            (t (ack-it-aux (cons head (cons (- head 1) tail))
                           (- z 1)))))))
```

El intento de prueba de terminación que esta definición genera en ACL2 se completa con éxito sin dificultad. Basta con probar algunos lemas previos que ayuden a demostrar que la medida multiconjuntista decrece (respecto de `mul-rel-ack`) en cada una de las llamadas recursivas. Una vez que se ha admitido la definición de `ack-it-aux`, podemos definir `ack-it`:

```
(defun ack-it (m n) (ack-it-aux (list m) n))
```

Podemos probar también que `ack-it` calcula la función de Ackermann:

```
(defthm ack-it-equal-ack
   (equal (ack-it m n) (ack m n)))
```

Para probar este resultado sólo es necesario probar un lema previo expresando que `(ack-it-aux S z) = (ack` s_k `(ack` s_{k-1} `...(ack` s_1 z`)))`, donde `S=` $(s_1 \ldots s_k)$. El teorema `ack-it-equal-ack` es un caso particular para `S=(list m)` y `z=n`.

En el apéndice C, subsección C.3.1, damos algunos detalles adicionales sobre la demostración automática de estos resultados.

5.5 La función 91 de McCarthy

Presentamos en esta sección la admisión de una versión iterativa de la función 91 de McCarthy. No pretendemos realizar un estudio detallado de tal función. Nuestra intención es mostrar cómo las relaciones entre multiconjuntos pueden ayudar a probar propiedades de terminación no triviales. Un tratamiento detallado (en ACL2) de la función 91 de McCarthy y de su generalización dada por Knuth, aparece en Cowles [15], donde incluso los teoremas se generalizan sobre cuerpos arquimedianos arbitrarios.

El libro `mccarthy-91.lisp` contiene la secuencia de eventos que llevan a la prueba automática de los resultados que se presentan a continuación. Este libro incluye el libro `defmul.lisp`.

La "función 91" es una función definida sobre el conjunto de los números enteros, originalmente dada por McCarthy [18] y definida por el siguiente esquema recursivo:

```
(defun mc (x)
  (declare (xargs :mode :program))
  (cond ((not (integerp x)) x)
```

```
        ((> x 100) (- x 10))
        (t (mc (mc (+ x 11))))))
```

Esta definición no puede ser aceptada por ACL2, de ahí que la hayamos definido en modo `:program`. La razón es que habría que encontrar una medida `(m x)` que probara

```
;(implies (and (integerp x) (not (> x 100)))
;         (e0-ord-< (m (mc (+ x 11))) (m x)))
```

sin tener ningún axioma sobre la función `mc` (puesto que aún no habría sido admitida). Si esto fuera posible, también sería cierto el mismo teorema sustituyendo `(mc (+ x 11))` por una variable cualquiera, obteniendo como teorema una fórmula de la cual existen instancias falsas, lo que derivaría una inconsistencia.

En su lugar, tratamos de definir la siguiente versión iterativa del esquema recursivo anterior, dada por las siguientes funciones:

```
; (defun mc-aux (n z)
;   (cond ((or (zp n) (not (integerp z))) z)
;         ((> z 100) (mc-aux (- n 1) (- z 10)))
;         (t (mc-aux (+ n 1) (+ z 11)))))

; (defun mc-it (x) (mc-aux 1 x))
```

Como demostraremos, el algoritmo recursivo definido por `mc-it` (a través de `mc-it--aux`) es una manera (bastante complicada, por cierto) de calcular la siguiente función `f91`:

```
(defun f91 (x)
  (cond ((not (integerp x)) x)
        ((> x 100) (- x 10))
        (t 91)))
```

La idea principal en la definición de `mc-aux` es que en cada iteración se cumple `(mc-aux n z)`=`(f91 (f91 .`^{n}`.(f91 z)))` y como consecuencia `(mc-it x)` = `(f91 x)`.

Al igual que el ejemplo anterior, obtener una prueba de que el algoritmo definido por `mc-aux` termina puede ser difícil: obsérvese el diferente comportamiento de las dos llamadas recursivas. En la primera de ellas, los dos argumentos decrecen y en la segunda, ambos se incrementan. En [16], se describe una medida multiconjuntista en la que nos basaremos para probar en ACL2 la terminación de la función `mc-aux`, siguiendo el camino marcado por la siguiente prueba informal:

Prueba informal: Sean n y z los argumentos de la función `mc-it-aux` y notemos por f a la función `f91`. Sea $D = \{x \in \mathbb{Z} : x \leq 111\}$[5] y $\sqsubset$ la relación definida en D por $x \sqsubset y$ si y sólo si $x > y$ (donde $>$ es el orden usual entre enteros). Es fácil ver que $\sqsubset$ es una relación bien fundamentada en D, por lo que, por el teorema 5.8, $\sqsubset_{mul}$ es una relación bien fundamentada en $\mathcal{M}(D)$.

[5]Durante el desarrollo de la prueba ACL2 de este resultado, descubrimos un error menor en la prueba que aparece en [16]: es necesario considerar los enteros menores o iguales a 111 y no sólo los estrictamente menores que 111.

Sea ψ una función definida sobre los argumentos de `mc-it-aux`, asignándoles un elemento de $\mathcal{M}(D)$ definido por

$$\psi(n, z) = \{z, f(z), f(f(z)), \ldots, f^{n-1}(z)\}$$

Para probar la terminación de `mc-it-aux`, bastará con probar que la medida ψ decrece, respecto de $\sqsubset_{mul}$, en cada llamada recursiva de la función. Obsérvese que en las llamadas recursivas podemos suponer que n es un número natural distinto de 0 y que z es un entero. Distinguimos tres casos, según el valor de z:

1. $z > 100$: en este caso los argumentos de la llamada recursiva que se produce se miden con el multiconjunto

   $$\psi(n-1, z-10) = \{z-10, f(z-10), \ldots, f^{n-2}(z-10)\}$$

   Puesto que $f(z) = z - 10$ entonces se tiene

   $$\psi(n-1, z-10) = \{f(z), \ldots, f^{n-1}(z)\}$$

   por lo que evidentemente $\psi(n-1, z-10) \sqsubset_{mul} \psi(n, z)$.

2. $90 \leq z \leq 100$: en este caso tenemos

   $$\psi(n+1, z+11) = \{z+11, f(z+11), \ldots, f^{n}(z+11)\}$$

   Nótese que $z + 11 > 100$ y por tanto $f(z + 11) = z + 1$. Además $f(z + 1) = 91$, ya que $z + 1 \leq 101$. En consecuencia, $f^2(z + 11) = 91 = f(z)$ y esto implica que

   $$\psi(n+1, z+11) = \{z+11, z+1, f(z), \ldots, f^{n-1}(z)\}$$

   y por tanto $\psi(n + 1, z + 11) \sqsubset_{mul} \psi(n, z)$, ya que z ha sido reemplazado por $z + 11$ y $z + 1$, que verifican $z + 11 \sqsubset z$ y $z + 1 \sqsubset z$.

3. $z < 90$: al igual que en el caso anterior, tenemos que

   $$\psi(n+1, z+11) = \{z+11, f(z+11), \ldots, f^{n}(z+11)\}$$

   Nótese que en este caso, $f^n(z + 11) = 91$, para cualquier n y de la misma manera $f^n(z) = 91$. Por tanto, podemos expresar el anterior multiconjunto como:

   $$\psi(n+1, z+11) = \{z+11, 91, f(z), \ldots, f^{n-1}(z)\}$$

   por lo que nuevamente $\psi(n + 1, z + 11) \sqsubset_{mul} \psi(n, z)$, ya que z ha sido reemplazado por $z + 11$ y 91, que verifican $z + 11 \sqsubset z$ y $91 \sqsubset z$.

□

Para formalizar en la lógica de ACL2 el anterior argumento de terminación, definimos en primer lugar la relación bien fundamentada `rel-mc` que inducirá la relación multiconjuntista. Obsérvese que en este caso, la propiedad que define los objetos de medida es `t`, aún cuando sólo los números enteros menores o iguales que `111` serán comparables respecto de `rel-mc`. Se podría pensar que la propiedad de medida adecuada en este caso debería ser

`integerp-<=-111`, en lugar de `t`. Pero hay una sutil diferencia: la función de medida que definiremos puede devolver multiconjuntos cuyos elementos sean enteros mayores que `111`, aunque tales elementos no sean comparables respecto a la relación `rel-mc`. La siguiente secuencia de eventos define a `rel-mc` y la almacena en el sistema como una relación bien fundamentada:

```
(defun integerp-<=-111 (x)
  (and (integerp x) (<= x 111)))

(defun rel-mc (x y)
  (and (integerp-<=-111 x) (integerp-<=-111 y) (< y x)))

(defun fn-mc (x)
  (if (integerp-<=-111 x) (- 111 x) 0))

(defthm rel-mc-well-founded
  (and (e0-ordinalp (fn-mc x))
       (implies (rel-mc x y)
                (e0-ord-< (fn-mc x) (fn-mc y))))
  :rule-classes :well-founded-relation)
```

Para definir la relación bien fundamentada inducida por `rel-mc` definida entre multiconjuntos (objetos que satisfacen `true-listp`, en este caso), basta con la siguiente llamada a `defmul`:

```
(defmul (rel-mc rel-mc-well-founded t fn-mc x y))
```

Esta llamada a `defmul` hace que se defina la función `mul-rel-mc` como relación bien fundamentada, con propiedad de medida `true-listp` y función de inmersión `map-fn-e0--ord`. Ahora `mul-rel-mc`, junto con una función de medida adecuada, la podemos usar como relación bien fundamentada en una prueba de la terminación de `ack-it-aux`. La prueba informal que hemos dado anteriormente, nos sugiere la definición de una función de medida adecuada, llamada `measure-mc-aux`:

```
(defun measure-mc-aux (n z)
  (if (zp n) nil (cons z (measure-mc-aux (- n 1) (f91 z)))))
```

Ahora podemos definir `mc-aux`, proporcionando `mul-rel-mc` y `measure-mc-aux` como relación bien fundamentada y función de medida, respectivamente, que justifican su terminación:

```
(defun mc-aux (n z)
  (declare (xargs :measure (measure-mc-aux n z)
                  :well-founded-relation mul-rel-mc))
  (cond ((or (zp n) (not (integerp z))) z)
        ((> z 100) (mc-aux (- n 1) (- z 10)))
        (t (mc-aux (+ n 1) (+ z 11)))))
```

El intento de prueba que se genera con esta definición con el objeto de admitirla, se completa con éxito, necesitándose sólo un lema previo (véase sección C.3.2).

Una vez que se ha definido `mc-aux`, podemos definir la función `mc-it`:

```
(defun mc-it (x) (mc-aux 1 x))
```

Es posible probar que dicha función implementa un algoritmo para calcular `f91` (para ello se necesita un lema previo expresando que `(mc-aux n z)`=`(f91 (f91 .`^{n}`.(f91 z)))` y luego usarlo para `n=1` y `z=x`).

```
(defthm mc-it-equal-f91
  (equal (mc-it x) (f91 x)))
```

También se puede probar que `mc-it` verifica el esquema de recursión originalmente definido por McCarthy:

```
(defthm mc-it-recursive-schema
  (equal (mc-it x)
         (cond ((not (integerp x)) x)
               ((> x 100) (- x 10))
               (t (mc-it (mc-it (+ x 11)))))))
```

Incluso hemos probado que toda función que verifique tal esquema de recursión necesariamente ha de ser igual a `f91`. Consúltese `mccarthy-91.lisp` para más detalles.

En el apéndice C, subsección C.3.2, damos algunos detalles adicionales sobre la demostración automática de estos resultados.

Sumario

En este capítulo:

- Hemos probado formalmente que la relación entre multiconjuntos inducida por una relación bien fundamentada está bien fundamentada.
- Este resultado se ha demostrado de manera totalmente general, lo que nos permite definir un comando, llamado `defmul`, para generar de manera automática la relación bien fundamentada entre multiconjuntos, inducida por una relación bien fundamentada concreta.
- Hemos ilustrado el uso de `defmul` para probar la terminación de funciones recursivas, mediante dos ejemplos: una versión recursiva de cola de la función de Ackermann y una versión iterativa de la función 91 de McCarthy.

Capítulo 6

Reducciones abstractas

En este capítulo mostramos cómo formalizar las reducciones abstractas en la lógica de ACL2. Muchos de los conceptos que presentamos en este capítulo se aplicarán al estudio de la lógica ecuacional y de los sistemas de reescritura de términos en el capítulo siguiente.

El concepto de reducción es una abstracción matemática, que nos permite discutir sobre las propiedades comunes a cualquier actividad que se produce paso a paso, tales como la computación que se realiza al ejecutar un programa, el recorrido de un grafo dirigido, o la progresiva transformación de un objeto, hasta que (eventualmente) se obtiene un objeto irreducible, lo que se conoce como forma normal.

Dos son las propiedades acerca de las reducciones sobre las que se centra el presente capítulo. En primer lugar, la terminación, o noetherianidad, que asegura que no se pueden aplicar sucesivamente un número infinito de pasos de reducción. Así, tenemos garantizada la obtención de formas normales. Otra propiedad importante en una reducción es la confluencia, propiedad que asegura que no es posible obtener dos formas normales distintas para un mismo elemento. De esta manera, toda reducción noetheriana y confluente tiene la siguiente importante propiedad: cualquier elemento tiene una única forma normal, independientemente de cómo se realicen las reducciones. Esta característica es importante para obtener procedimientos de decisión de la relación de equivalencia asociada a una reducción.

En este capítulo, definimos todos estos conceptos (reducción, formas normales, terminación,...) en la lógica de ACL2 y demostramos algunos resultados interesantes. El resultado más importante es una prueba automática del lema de Newman, que nos permite deducir la confluencia de toda reducción noetheriana y localmente confluente.

La principal característica en la formalización que mostramos en este capítulo es que los conceptos y propiedades se presentan en un marco general, permitiendo más tarde trasladar los resultados a reducciones concretas mediante instanciación funcional. En particular, los resultados de este capítulo serán fundamentales para la formalización en ACL2 de la lógica ecuacional y de los sistemas de reescritura de términos que presentamos en el capítulo 7.

6.1 Reducciones abstractas: conceptos y propiedades

Recordamos en esta sección los principales conceptos y propiedades que pretendemos formalizar, tal y como se presentan habitualmente en la literatura. En concreto, lo que sigue

está basado en el capítulo 2 de [1].

Definición 6.1 *Una* **reducción** *en un conjunto A es una relación binaria en A. Diremos entonces que A es el* **dominio de definición** *de la reducción.*

Usualmente, una reducción se representa con notación en forma de flecha (como $\to$ ó $\to_1$), aunque podremos usar otros símbolos, como R ó $>$. En lo que sigue se supondrá que $\to$ es una reducción sobre un conjunto A. La relación entre dos elementos se suele expresar en notación infija, como por ejemplo $x \to y$. Los siguientes conceptos y notaciones nos serán de utilidad.

Definición 6.2

$\to_1 \circ \to_2$	$=$	$\{(x,z) : (\exists y)[x \to_1 y \wedge y \to_2 z]\}$	**Composición**
$\overset{-1}{\to}$	$=$	$\{(y,x) : x \to y\}$	**Inversa**
$\leftarrow$	$=$	$\overset{-1}{\to}$	**Inversa**
$\overset{0}{\to}$	$=$	$\{(x,x) : x \in A\}$	**Identidad**
$\overset{=}{\to}$	$=$	$\to \cup \overset{0}{\to}$	**Clausura reflexiva**
$\overset{i+1}{\to}$	$=$	$\to \circ \overset{i}{\to}$	$i+1-$ **composición**
$\overset{+}{\to}$	$=$	$\bigcup_{i>0} \overset{i}{\to}$	**Clausura transitiva**
$\overset{*}{\to}$	$=$	$\overset{+}{\to} \cup \overset{0}{\to}$	**Clausura reflexiva y transitiva**
$\leftrightarrow$	$=$	$\leftarrow \cup \to$	**Clausura simétrica**
$\overset{+}{\leftrightarrow}$	$=$	$(\leftrightarrow)^+$	**Clausura simétrica y transitiva**
$\overset{*}{\leftrightarrow}$	$=$	$(\leftrightarrow)^*$	**Clausura de equivalencia**

La palabra *clausura* empleada en las definiciones anteriores tiene un sentido preciso. Dada una relación R y una propiedad P, la P-clausura de R es la menor relación que contiene a R y que cumple la propiedad P. Por ejemplo, se puede probar que $\overset{*}{\leftrightarrow}$ es la menor relación de equivalencia que contiene a $\to$ y análogamente con el resto de clausuras que se han definido. La relación $\overset{*}{\leftrightarrow}$ también se suele llamar **relación de equivalencia descrita por** la reducción $\to$.

Sigamos con más definiciones que nos serán de utilidad en el desarrollo posterior. Todas las definiciones se entienden respecto de una reducción $\to$ definida en A.

Definición 6.3 *Decimos que y es un* **sucesor** *(***inmediato***) de x si $x \overset{+}{\to} y$ ($x \to y$). Si no existe z tal que $y \to z$ decimos que y está en* **forma normal** *(o que es* **irreducible***) respecto de $\to$. Si $x \overset{*}{\leftrightarrow} y$ e y está en forma normal, decimos que y es una forma normal de x (respecto de $\to$). Notamos por $x\downarrow$ a la forma normal de x, cuando ésta existe y es única. Si existe z tal que $x \overset{*}{\to} z$ e $y \overset{*}{\to} z$, lo notamos por $x \downarrow y$ y decimos que x e y* **convergen**. *Una $\to$-derivación es una sucesión (finita o infinita) $x_1, x_2, x_3, \ldots$ de elementos tales que $x_i \to x_{i+1}$. La notaremos por $x_1 \to x_2 \to x_3 \cdots$. Dos elementos $x, y \in A$ son* **equivalentes** *(respecto de $\to$) si $x \overset{*}{\leftrightarrow} y$.*

Nos será también útil cierta terminología para designar determinados tipos de $\leftrightarrow$-derivaciones.

Definición 6.4 *Una $\leftrightarrow$-derivación de la forma $x \overset{*}{\leftarrow} u \overset{*}{\rightarrow} y$ se denomina una* **montaña**. *Si es de la forma $x \overset{*}{\rightarrow} z \overset{*}{\leftarrow} y$ decimos que es un* **valle**. *Y si es de la forma $x \leftarrow u \rightarrow y$, la llamamos* **pico local**.

Ejemplo 6.5 En el conjunto de las fracciones positivas, es posible definir la siguiente reducción $\rightarrow_S$:

$$\frac{a}{b} \rightarrow_S \frac{c}{d} \iff (\exists n \in \mathbb{N}_+)[n \text{ primo } \wedge a = nc \wedge b = nd]$$

Entonces $\{\frac{6}{4}, \frac{3}{2}\}$ es el conjunto de sucesores de $\frac{12}{8}$ además $\frac{3}{2}$ es una forma normal de $\frac{12}{8}$. $\frac{12}{8} \rightarrow_S \frac{6}{4} \rightarrow_S \frac{3}{2}$ es una $\rightarrow_S$-derivación. Es fácil probar que la equivalencia de fracciones coincide con la relación $\overset{*}{\leftrightarrow}_S$ (ya que dos fracciones equivalentes se reducen por $\overset{*}{\rightarrow}_S$ a la misma fracción equivalente irreducible).

Ejemplo 6.6 Sea $\rightarrow_P$ la reducción en el conjunto $\mathbb{N}^*$ de las cadenas de números naturales definida por:

$$\rightarrow_P = \{(uijv, ujiv) : u, v \in \mathbb{N}^*, i, j \in \mathbb{N}, i > j\}$$

Es posible probar que la reducción $\rightarrow_P$ describe la relación de equivalencia "ser permutación de". Es decir, $u \overset{*}{\leftrightarrow}_P v$ si y sólo si u es una permutación de v. Los elementos irreducibles respecto de $\rightarrow_P$ son las cadenas de números naturales ordenadas de manera creciente.

Los ejemplos anteriores nos ilustran los dos aspectos esenciales de toda reducción. Por un lado, existe un aspecto computacional: es la especificación de una cierta computación dirigida, que obtiene paso a paso objetos cada vez más "reducidos", hasta que eventualmente se llega a una forma normal. Por otro lado, un aspecto declarativo, ya que $\rightarrow$ puede ser considerada como una descripción de $\overset{*}{\leftrightarrow}$. Las propiedades que definiremos a continuación van encaminadas a establecer la relación existente entre ambos aspectos o, dicho de otro modo, a expresar cuándo la reducción puede ser usada para computar su clausura de equivalencia.

Una propiedad importante en una reducción es la existencia de al menos una forma normal equivalente para cada elemento de su dominio de definición:

Definición 6.7 *Una reducción $\rightarrow$ se dice* **normalizadora** *si todo elemento de A tiene una forma normal.*

La siguiente propiedad, asegura la existencia de a lo sumo una forma normal equivalente para cada elemento.

Definición 6.8 *Decimos que una reducción $\rightarrow$ tiene la* **propiedad de Church-Rosser** *si para todo $x, y \in A$ tal que $x \overset{*}{\leftrightarrow} y$, se tiene que $x \downarrow y$.*

Proposición 6.9 *Supongamos que $\rightarrow$ es una reducción con la propiedad de Church-Rosser y sean $x, y \in A$ en forma normal tal que $x \overset{*}{\leftrightarrow} y$. Entonces $x = y$.*

Demostración:
Si $x \overset{*}{\leftrightarrow} y$, por la propiedad de Church-Rosser se tiene que $x \downarrow y$. Es decir existe $z \in A$ tal que $x \overset{*}{\rightarrow} z \overset{*}{\leftarrow} y$. Puesto que x e y están en forma normal, necesariamente se tiene que $x = z = y$ □

Corolario 6.10 *Si $\rightarrow$ es una reducción con la propiedad de Church-Rosser, todo elemento $x \in A$ tiene a lo sumo una forma normal.*

Demostración:
Si $x \overset{*}{\leftrightarrow} y$ y $x \overset{*}{\leftrightarrow} z$, con y y z en forma normal, entonces $y \overset{*}{\leftrightarrow} z$. Por tanto podemos concluir, por la proposición 6.9, que $y = z$. □

Nótese que si una reducción es normalizadora y Church-Rosser, tiene sentido la notación $x \downarrow$ para designar a la única forma normal de x. El siguiente teorema expresa la propiedad fundamental de las reducciones normalizadoras con la propiedad de Church-Rosser.

Teorema 6.11 *Supongamos que $\rightarrow$ es una reducción normalizadora con la propiedad de Church-Rosser. Entonces, para todo $x, y \in A$, se tiene que $x \overset{*}{\leftrightarrow} y$ si y sólo si $x \downarrow = y \downarrow$.*

Demostración:
$\boxed{\Longrightarrow}$ Si $x \overset{*}{\leftrightarrow} y$, entonces $x \downarrow \overset{*}{\leftrightarrow} x \overset{*}{\leftrightarrow} y \overset{*}{\leftrightarrow} y \downarrow$, luego $x \downarrow \overset{*}{\leftrightarrow} y \downarrow$. Puesto que tanto $x \downarrow$ como $y \downarrow$ están en forma normal, por la proposición 6.9 se tiene que $x \downarrow = y \downarrow$.
$\boxed{\Longleftarrow}$ En este caso, $x \overset{*}{\leftrightarrow} x \downarrow = y \downarrow \overset{*}{\leftrightarrow} y$ y por tanto $x \overset{*}{\leftrightarrow} y$. □

El teorema 6.11 establece que en cada clase de equivalencia de $\overset{*}{\leftrightarrow}$ existe un único elemento en forma normal. Si se dispone de un método efectivo de "simplificación" que obtenga la forma normal de un elemento, entonces es posible computar la relación $\overset{*}{\leftrightarrow}$ simplemente comparando las formas normales de los elementos.

Definición 6.12 *Sea $\sim$ una relación de equivalencia sobre un conjunto A. Un* **simplificador canónico** *para $\sim$ es una función computable $S : A \rightarrow A$ tal que para cualesquiera $x, y \in A$ se verifica que $S(x) \sim x$ y tal que $x \sim y \Rightarrow S(x) = S(y)$.*

Ejemplo 6.13 La aplicación definida para el conjunto de las fracciones positivas tal que a cada fracción le hace corresponder su equivalente irreducible, es un simplificador canónico para la equivalencia de fracciones. Nótese que se trata de una función evidentemente computable (por ejemplo, dividir numerador y denominador por el m.c.d. de ambos).

Ejemplo 6.14 La aplicación definida en $\mathbb{N}^*$ tal que a cada cadena v de números naturales le asigna la cadena obtenida ordenando los elementos de v de manera creciente es un simplificador canónico para la relación "ser permutación de" definida en $\mathbb{N}^*$.

Proposición 6.15 *Sea A un conjunto cualquiera en el que es computable la relación identidad y $\sim$ una relación de equivalencia sobre A. Entonces $\sim$ es decidible si existe un simplificador canónico para $\sim$.*

Demostración:
Es consecuencia directa de que $x \sim y$ si y sólo si $S(x) = S(y)$. Esta equivalencia se tiene por que de $x \sim S(x)$ y de $y \sim S(y)$ se sigue que $S(x) = S(y) \Rightarrow x \sim y$. La implicación contraria se tiene por definición. □

La proposición 6.15 permite formular el siguiente teorema sobre la decidibilidad de la relación $\overset{*}{\leftrightarrow}$ cuando $\rightarrow$ es normalizadora y tiene la propiedad de Church-Rosser.

Teorema 6.16 *Sea A un conjunto cualquiera, $\sim$ una relación de equivalencia sobre A y $\rightarrow$ una reducción normalizadora con la propiedad de Church-Rosser, definida en A, tal que $\overset{*}{\leftrightarrow}=\sim$. Supongamos además que existe una función computable $n : A \rightarrow A$ tal que $n(x)$ es una forma normal de x y que la identidad en A es computable. Entonces $\sim$ es decidible.*

Demostración:
Por la proposición 6.15, basta ver que n es un simplificador canónico para $\overset{*}{\leftrightarrow}$. Y esto se tiene por el teorema 6.11 y por el hecho de que $x \overset{*}{\leftrightarrow} n(x)$. □

Las propiedades de normalización y de Church-Rosser pueden ser sustituidas por otras más manejables desde el punto de vista de los sistemas de reescritura de términos (tema 7). Por ejemplo, es más usual trabajar con la propiedad de noetherianidad en lugar de la normalización. Aunque la normalización basta para tener formas normales, la búsqueda de una forma normal es más fácil si se asegura que eventualmente cualquier $\rightarrow$-derivación termina en una forma normal. Aunque ya se había definido este concepto en el capítulo 5, volvemos aquí a enunciar su definición en el contexto de las reducciones abstractas.

Definición 6.17 *Una reducción $\rightarrow$ se dice* **noetheriana** *(o que* **termina***) si no existen $\rightarrow$-derivaciones infinitas.*

Ejemplo 6.18 La reducción$\rightarrow_S$ del ejemplo 6.5 es noetheriana. No existen $\rightarrow_S$-derivaciones infinitas, ya que esto implicaría la existencia de un par de números naturales positivos a y b y de una sucesión infinita de números primos, $n_1, n_2, n_3, \ldots$ tales que $n_1, n_1n_2, n_1n_2n_3, \ldots$ es una sucesión infinita estrictamente creciente de divisores comunes de a y b, lo cual es imposible pues los divisores comunes de a y b están acotados superiormente (por el mínimo de ambos, por ejemplo).

Ejemplo 6.19 La reducción $\rightarrow_P$ del ejemplo 6.6 es noetheriana. Dada una cadena $w \in \mathbb{N}$, definimos

$$d(w) = |\{(i, j) : w = uivjz, u, v, z \in \mathbb{N}^*, i, j \in \mathbb{N}, i > j\}|$$

Intuitivamente, $d(w)$ cuenta el número de "desórdenes relativos" que hay entre los elementos de la cadena w. Nótese que si $w_1 \rightarrow_P w_2$, entonces $d(w_1) > d(w_2)$. Puesto que d toma sus valores en los números naturales, no es posible por tanto que exista una $\rightarrow_P$-derivación infinita.

Proposición 6.20 *Toda reducción noetheriana es normalizadora.*

Demostración:
Aplicamos inducción noetheriana a la propiedad $P(x) \equiv$ "x tiene una forma normal". Sea

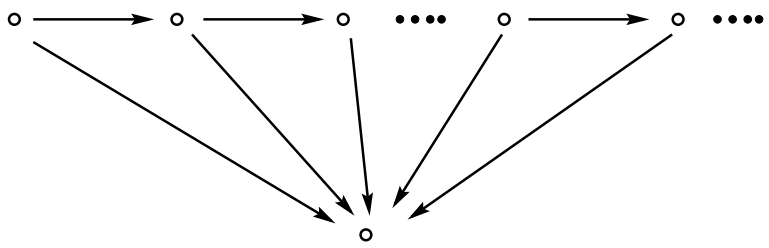

Figura 6.1: Reducción normalizadora no noetheriana

$x \in A$ y supongamos que se verifica $P(y)$ para cualquier $y \in A$ tal que $x \to y$. Si x está en forma normal entonces es evidente, ya que $x \overset{*}{\leftrightarrow} x$. Si no, sea y tal que $x \to y$. Aplicando hipótesis de inducción, se tiene que existe z en forma normal tal que $y \overset{*}{\leftrightarrow} z$. Por tanto $x \overset{*}{\leftrightarrow} z$. □

La demostración anterior es un ejemplo típico del uso de inducción noetheriana para probar propiedades de reducciones noetherianas. La noetherianidad es una propiedad estrictamente más fuerte que la de normalización, como muestra la figura 6.1.

Existen también otras propiedades que, bajo ciertas condiciones, aseguran la existencia de formas normales únicas, enunciadas en la siguiente definición.

Definición 6.21 *Una reducción $\to$ se dice:*

Confluente *si $x \overset{*}{\leftarrow} u \overset{*}{\rightarrow} y$ implica que $x \downarrow y$*

Semi-confluente *si $x \leftarrow u \overset{*}{\rightarrow} y$ implica que $x \downarrow y$*

Localmente confluente *si $x \leftarrow u \rightarrow y$ implica que $x \downarrow y$*

La figura 6.2 muestra gráficamente las propiedades de Church-Rosser, confluencia y semiconfluencia. Este tipo de diagramas son usuales en la literatura sobre reducciones abstractas. Las lineas continuas representan $\leftrightarrow$-derivaciones cuantificadas universalmente. Las lineas discontinuas representan $\leftrightarrow$-derivaciones cuantificadas existencialmente. Por ejemplo, la representación de la confluencia indica que *para todo* u, x, y tal que $x \overset{*}{\leftarrow} u \overset{*}{\rightarrow} y$, *existe* z tal que $u \overset{*}{\rightarrow} z \overset{*}{\leftarrow} y$.

Veamos la relación que hay entre estas propiedades y la propiedad de Church-Rosser. El siguiente teorema establece que la confluencia y la semi-confluencia son propiedades equivalentes a la propiedad de Church-Rosser.

Teorema 6.22 *Dada una reducción $\to$, las siguientes condiciones son equivalentes:*

(1) $\to$ tiene la propiedad de Church-Rosser.

(2) $\to$ es confluente.

(3) $\to$ es semi-confluente.

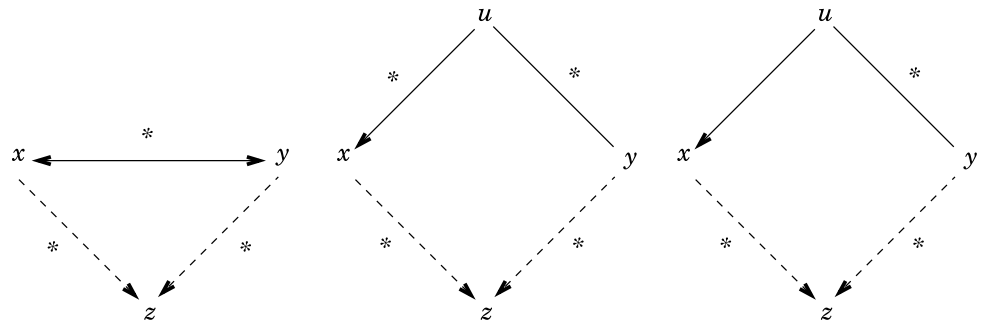

Figura 6.2: Propiedad de Chuch-Rosser, confluencia y semiconfluencia

Demostración:

$\boxed{(1) \Longrightarrow (2)}$ Si $x \overset{*}{\leftarrow} u \overset{*}{\rightarrow} y$, entonces $x \overset{*}{\leftrightarrow} y$ y por tener la reducción $\rightarrow$ la propiedad de Church-Rosser, se tiene $x \downarrow y$.

$\boxed{(2) \Longrightarrow (3)}$ Es inmediato a partir de las definiciones.

$\boxed{(3) \Longrightarrow (1)}$ Supongamos que $x \overset{*}{\leftrightarrow} y$. Probemos que $x \downarrow y$ por inducción en el número n tal que $x \overset{n}{\leftrightarrow} y$. Existen dos posibilidades:

i) Si $x \overset{0}{\leftrightarrow} y$, entonces $x = y$ y es evidente que $x \downarrow y$.

ii) Si $x \overset{n+1}{\leftrightarrow} y$, sea x' tal que $x \leftrightarrow x' \overset{n}{\leftrightarrow} y$. Aplicando hipótesis de inducción, existe z tal que $x' \overset{*}{\rightarrow} z \overset{*}{\leftarrow} y$. Pueden ocurrir dos casos:

   Caso 1 $x \rightarrow x'$, en cuyo caso es evidente que $x \downarrow y$.

   Caso 2 $x \leftarrow x'$ y entonces por ser $\rightarrow$ semi-confluente, se tiene que existe v tal que $x \overset{*}{\rightarrow} v$ y $z \overset{*}{\rightarrow} v$. Por tanto, $x \overset{*}{\rightarrow} v$ e $y \overset{*}{\rightarrow} z \overset{*}{\rightarrow} v$, con lo que $x \downarrow y$.

□

El demostrar la confluencia de una reducción, aunque más simple que mostrar la propiedad de Church-Rosser, puede ser complicado, ya que hay que probar que x e y convergen para cualquier montaña $x \overset{*}{\leftarrow} u \overset{*}{\rightarrow} y$. La tarea se simplificaría considerablemente si sólo hubiera que considerar picos locales. Es decir, si la confluencia local bastara para concluir la propiedad de Church-Rosser. Lamentablemente, esto no es cierto en general, como muestra el siguiente ejemplo.

Ejemplo 6.23 Considérese la reducción que representa la figura 6.3. Es fácil comprobar que es localmente confluente, analizando cada pico local. Sin embargo, no es confluente, ya que tiene dos elementos equivalentes, a y d, que están en forma normal.

Sin embargo, en el caso de reducciones noetherianas, la confluencia local sí que basta para deducir la confluencia y, por tanto, la propiedad de Church-Rosser. Este resultado fue demostrado por primera vez por M.H. Newman en [56] (por lo que es conocido como

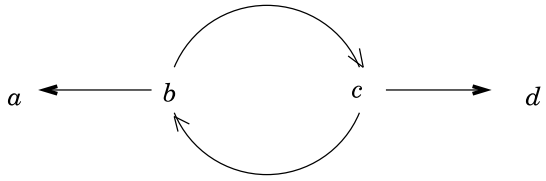

Figura 6.3: Reducción localmente confluente no confluente

Lema de Newman). La demostración que aquí presentamos es considerablemente más simple que la original y se debe a G. Huet [30].

Teorema 6.24 (Lema de Newman) *Una reducción noetheriana y localmente confluente es confluente.*

Demostración:

Supongamos que $\rightarrow$ es una reducción definida en A, noetheriana y localmente confluente. Sea $P(x)$ la propiedad "para cualesquiera y, z tales que $y \overset{*}{\leftarrow} x \overset{*}{\rightarrow} z$, se tiene que $y \downarrow z$". Es claro que $\rightarrow$ es confluente si y sólo si $(\forall x \in A)[P(x)]$. Vamos a probar que para cualquier $x \in A$ se cumple $P(x)$ y lo haremos por inducción noetheriana, lo cual está justificado por la noetherianidad de $\rightarrow$.

Sea $x \in A$ y supongamos que para cualquier $x' \in A$ tal que $x \rightarrow x'$ se verifica $P(x')$. Sea y, z tales que $y \overset{*}{\leftarrow} x \overset{*}{\rightarrow} z$. Si $y = x$ ó $z = x$ entonces es claro que $y \downarrow z$. Si no, sean y_1,z_1 tales que $y \overset{*}{\leftarrow} y_1 \leftarrow x \rightarrow z_1 \overset{*}{\rightarrow} z$. La demostración procede tal y como se esquematiza en la figura 6.4. Por ser $\rightarrow$ localmente confluente, existe u tal que $y_1 \overset{*}{\rightarrow} u$ y $z_1 \overset{*}{\rightarrow} u$. Aplicando hipótesis de inducción (a y_1, para $y_1 \overset{*}{\rightarrow} u$, $y_1 \overset{*}{\rightarrow} y$) se tiene que existe v tal que $y \overset{*}{\rightarrow} v$ y $u \overset{*}{\rightarrow} v$. Aplicando hipótesis de inducción (a z_1, para $z_1 \overset{*}{\rightarrow} v$, $z_1 \overset{*}{\rightarrow} z$) se tiene que existe w tal que $v \overset{*}{\rightarrow} w$ y $z \overset{*}{\rightarrow} w$. Entonces, puesto que $y \overset{*}{\rightarrow} v \overset{*}{\rightarrow} w$ y $z \overset{*}{\rightarrow} w$, se tiene que $y \downarrow z$.

□

Definición 6.25 *Una reducción se dice* **convergente** *si es noetheriana y localmente confluente*

Nótese que como consecuencia del lema de Newman, toda reducción convergente es confluente.

Ejemplo 6.26 El lema de Newman nos sirve para probar que la reducción $\rightarrow_S$ del ejemplo 6.5 es confluente y, por tanto, cumple la propiedad de Church-Rosser. Puesto que ya probamos que $\rightarrow_S$ es noetheriana en el ejemplo 6.18, basta probar que es localmente confluente.

Supongamos, que $\frac{a}{b} \rightarrow_S \frac{c_1}{d_1}$ y $\frac{a}{b} \rightarrow_S \frac{c_2}{d_2}$. Por tanto existen dos números primos n y m tales que $a = n \cdot c_1$, $b = n \cdot d_1$, $a = m \cdot c_2$ y $b = m \cdot d_2$. Si $n = m$, entonces $\frac{c_1}{d_1} = \frac{c_2}{d_2}$ y evidentemente ambas fracciones convergen. Si $n \neq m$, entonces n y m son divisores de a y b, y ya que son números primos, también lo es $n \cdot m$. Luego existen e_1, e_2, f_1, f_2 tales que $c_1 = m \cdot e_1$, $d_1 = m \cdot f_1$, $c_2 = n \cdot e_2$ y $d_2 = n \cdot f_2$. Ahora bien, $a = n \cdot m \cdot e_1 = m \cdot n \cdot e_2$, lo que

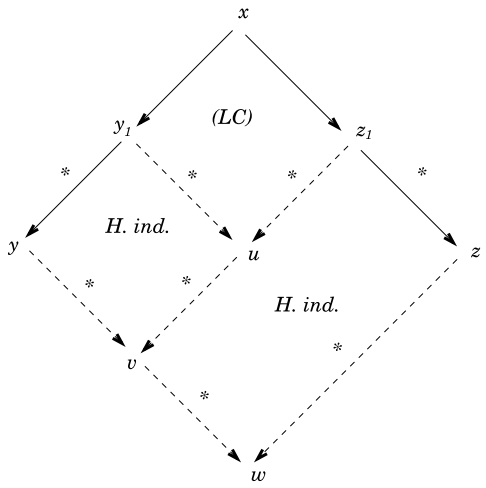

Figura 6.4: Lema de Newman

implica $e_1 = e_2$. De la misma manera $f_1 = f_2$. Por tanto tenemos $\frac{c_1}{d_1} \rightarrow_S \frac{e_1}{f_1} = \frac{e_2}{f_2} \leftarrow_S \frac{c_2}{d_2}$, lo que prueba la confluencia local de la relación $\rightarrow_S$.

Ejemplo 6.27 Veamos que la reducción $\rightarrow_P$ del ejemplo 6.6 es localmente confluente (lo cual implicará, por el lema de Newman y la noetherianidad de $\rightarrow_P$ vista en 6.19, que la reducción tiene la propiedad de Church-Rosser). Escribiremos $u \rightarrow_{P,i} v$ cuando $u \rightarrow_P v$ y v se obtiene de u intercambiando dos números naturales, desordenados entre sí y contiguos, estando el primero de ellos en la posición i de v. Supongamos que $u \leftarrow_{P,i} v \rightarrow_{P,j} w$. Si $i = j$, entonces $u = w$. Si $i + 1 < j$ (análogo el caso $j < i + 1$), entonces

$$u \rightarrow_{P,j} u_1 = w_1 \leftarrow_{P,i} w$$

Si, por el contrario, $i + 1 = j$ (si $j + 1 = i$ el razonamiento es análogo), entonces se tiene que

$$u \rightarrow_{P,j} u_1 \rightarrow_{P,i} u_2 = w_2 \leftarrow_{P,j} w_1 \leftarrow_{P,i} w$$

En cualquiera de los casos, se tiene que $u \downarrow_P v$.

Como consecuencia de los resultados expuestos anteriormente podemos concluir un resultado sobre la decidibilidad de relaciones de equivalencia descritas mediante reducciones convergentes. Antes necesitaremos definir la noción de test de reducibilidad.

Definición 6.28 *Sea A un conjunto cualquiera y $\rightarrow$ una reducción sobre A. Una función red se dice un* **test de reducibilidad** *para $\rightarrow$, si es una función computable $red : A \rightarrow A \cup \{\Box\}$, (donde $\Box$ es un elemento cualquiera que no pertenece a A) tal que para todo x irreducible, $red(x) = \Box$ y en caso contrario $x \rightarrow red(x)$.*

Teorema 6.29 *Sea A un conjunto cualquiera, $\sim$ una relación de equivalencia sobre A y $\rightarrow$ una reducción convergente sobre A tal que $\overset{*}{\leftrightarrow} = \sim$. Supongamos que existe un test de reducibilidad para $\rightarrow$. Entonces $\sim$ es decidible.*

Demostración:
Sea red un test de reducibilidad para $\rightarrow$ y supongamos que $\square \notin A$ es el elemento tal que si $red(x) = \square$ entonces x es irreducible. Sea $S : A \rightarrow A$ la función definida por:

$$S(x) = \begin{cases} x & \text{si } red(x) = \square \\ S(red(x)) & \text{en otro caso} \end{cases}$$

Por el teorema 6.16, bastará ver que S es una función computable y que $S(x)$ es una forma normal de x, para todo $x \in A$.

Es evidente que el algoritmo que define la función S termina, ya que $\rightarrow$ es noetheriana. Puesto que red es computable, se tiene por tanto que S es computable.

Para ver que $S(x)$ es una forma normal de x, probemos que $x \stackrel{*}{\rightarrow} S(x)$ y que $S(x)$ es irreducible. Si $P(x)$ es la propiedad "$x \stackrel{*}{\rightarrow} S(x)$ y $S(x)$ es irreducible", probemos $(\forall x \in A)[P(x)]$ por inducción noetheriana (tiene sentido ya que $\rightarrow$ es noetheriana). Por tanto, supongamos que $x \in A$ es tal que para cualquier $y \in A$ tal que $x \rightarrow y$ se verifica que $y \stackrel{*}{\rightarrow} S(y)$ y que $S(y)$ es irreducible. Si x está en forma normal, $S(x) = x$ y se tiene trivialmente $P(x)$. Si no, se verifica que $x \rightarrow red(x)$, por lo que es posible aplicar la hipótesis de inducción a $red(x)$. Es decir, $red(x) \stackrel{*}{\rightarrow} S(red(x))$ y $S(red(x))$ irreducible. Pero puesto que $S(x) = S(red(x))$ (ya que x no está en forma normal), se tiene que $x \rightarrow red(x) \stackrel{*}{\rightarrow} S(red(x)) = S(x)$. Luego $x \stackrel{*}{\rightarrow} S(x)$ y $S(x)$ irreducible. □

Ejemplo 6.30 Según el teorema 6.29 y lo visto en el ejemplo 6.26 la relación de equivalencia $\stackrel{*}{\leftrightarrow}_S$ es decidible, ya que es evidente que existe un test de reducibilidad de fracciones (basta calcular el m.c.d. de numerador y denominador)

Ejemplo 6.31 También se tiene la decidibilidad de la relación $\stackrel{*}{\leftrightarrow}_P$, ya que como se ha visto anteriormente $\rightarrow_P$ es noetheriana y Church-Rosser y existe un test de reducibilidad de cadenas: basta recorrer la cadena para ver si existe una pareja de números naturales contiguos y desordenados.

Por supuesto, la decidibilidad de las relaciones de los dos ejemplos anteriores se puede demostrar directamente sin necesidad de recurrir a mostrar una reducción que las describa. Estos ejemplos tienen un carácter meramente ilustrativo, pero existen otros casos en los que estos métodos basados en reducciones son realmente útiles para probar la decidibilidad de una relación de equivalencia. Un ejemplo relevante se muestra en el capítulo 7, donde una reducción, la reescritura de términos, sirve para probar la decidibilidad de algunas teorías ecuacionales.

En el resto de este capítulo presentamos una formalización en la lógica de ACL2 de los resultados sobre reducciones abstractas presentados en esta sección.

6.2 Reducciones abstractas: formalización

En esta sección sentamos las bases de la representación escogida para razonar sobre cualquier tipo de reducción usando ACL2. Tal representación ha de ser lo suficientemente abstracta para que sea posible trasladar los resultados a cualquier reducción concreta.

6.2.1 Representación

En primer lugar, obsérvese que si $x \to y$, más importante que el hecho de que x e y están relacionados de alguna manera, es el hecho de que y se obtiene a partir de x mediante la aplicación de cierta transformación u **operador**. Estamos así enfatizando más el concepto de *simplificación* que el de relación. De esta manera, en su formulación más abstracta, podemos ver una reducción como una función binaria que, a partir de un objeto y de un operador, devuelve otro objeto, llevando a cabo **un paso de reducción**. Por ejemplo, en el caso de la reducción del ejemplo 6.6, los objetos son las cadenas de números naturales y los operadores vienen descritos por posiciones en las que se produce el intercambio.

Por supuesto, cualquier operador no puede ser aplicado a cualquier objeto. Si pensamos nuevamente en el ejemplo 6.6, un operador que indica intercambio en una posición donde no hay elementos consecutivos desordenados no puede ser aplicado. Puesto que la lógica de ACL2 es una lógica de funciones totales, para discriminar qué operadores son aplicables y cuáles no, será necesario otro componente en la representación de una reducción abstracta: una función binaria que indicará cuándo es *legal* aplicar un operador a un objeto (lo que llamaremos un **test de aplicabilidad**).

Estas consideraciones nos llevan a representar una reducción cualquiera en ACL2 a través de tres funciones. En primer lugar, necesitamos especificar el conjunto en el cual pretendemos que esté definida la reducción y lo haremos mediante el correspondiente predicado unario. Además, según lo expuesto anteriormente, necesitaremos otra función que represente la aplicación de un operador a un objeto, llevando a cabo un paso de reducción. Finalmente, una tercera función comprobará si es "legal" la aplicación de un operador a un objeto. Como veremos más adelante, estas tres funciones bastan para poder formalizar conceptos tales como la propiedad de Church-Rosser, normalización, confluencia local, o la clausura de equivalencia. En principio, cualquier terna de funciones, una de ellas unaria (la que define el dominio de definición) y las otras dos binarias (las que definen la aplicación de un paso de reducción y el test de aplicabilidad), pueden representar a una reducción concreta.

Ejemplo 6.32 Podemos describir en ACL2 la relación $\to_P$ del ejemplo 6.6 mediante tres funciones, `natural-true-listp` (que define el dominio), `perm-reduce-one-step` (que aplica un paso de reducción) y `perm-legal` (el test de aplicabilidad), tal y como se muestra en la figura 6.5.

Estamos representando aquí las cadenas de números naturales mediante listas de números naturales y los operadores mediante números naturales que indican posiciones en las listas. El test de aplicabilidad comprueba que en la posición indicada por el operador y en la siguiente, se encuentran dos elementos desordenados entre sí (función `nth-disorder-p`).

En lo que sigue, usaremos esta reducción para ilustrar los conceptos y propiedades formalizadas. Las definiciones y teoremas correspondientes a este ejemplo se encuentran en libro `perm.lisp`.

Veamos ahora cómo podemos formalizar, usando la lógica de ACL2, un teorema que enuncie una propiedad general sobre reducciones abstractas. Tal teorema debe ser expresado de manera lo suficientemente general como para que se puede usar la correspondiente instancia para cualquier reducción concreta. Por tanto, debemos razonar sobre una reduc-

```
(defun natural-true-listp (l)
  (if (endp l)
      (equal l nil)
      (and (integerp (car l))
           (>= (car l) 0)
           (natural-true-listp (cdr l)))))

(defun perm-reduce-one-step (l i)
  (if (zp i)
      (list* (cadr l) (car l) (cddr l))
      (cons (car l) (perm-reduce-one-step (cdr l) (- i 1)))))

(defun nth-disorder-p (l i)
  (cond ((or (not (consp l)) (not (consp (cdr l)))) nil)
        ((zp i) (< (cadr l) (car l)))
        (t (nth-disorder-p (cdr l) (- i 1)))))

(defun perm-legal (l i)
  (and (natural-true-listp l)
       (integerp i) (<= 0 i)
       (nth-disorder-p l i)))
```

Figura 6.5: Reducción y aplicabilidad en el ejemplo de las permutaciones

ción sin definirla completamente, simplemente asumiendo como ciertas las propiedades que se necesiten en la hipótesis del teorema que se quiera probar. Atendiendo a estas consideraciones, usaremos `encapsulate` para introducir tres funciones, que llamaremos `q`, `reduce-one-step` y `legal`, representando, respectivamente el dominio de definición, la función que aplica un paso de reducción y el test de aplicabilidad, correspondientes a una reducción abstracta:

```
(encapsulate
 ((q (x) boolean)
  (reduce-one-step (x u) element)
  (legal (x u) boolean)
  ...)

 ...)
```

Si queremos demostrar algún resultado sobre reducciones abstractas en el que se asumen una serie de propiedades como hipótesis (por ejemplo, noetherianidad o la propiedad de Church-Rosser) y se concluyen otras (por ejemplo, decidibilidad de la relación de equivalencia asociada), estableceremos las propiedades asumidas dentro del encapsulado y a partir de ahí deduciremos las conclusiones fuera del ámbito del encapsulado.

De esta manera, los resultados probados para reducciones abstractas se pueden trasladar fácilmente a reducciones concretas, sin más que elegir una representación concreta

para los objetos y los operadores, y definir específicamente las correspondientes funciones que definen el dominio, un paso de reducción y la aplicabilidad. Si probamos que una reducción concreta tiene las propiedades asumidas como hipótesis en un teorema sobre reducciones abstractas, entonces el mecanismo de instanciación funcional permitirá hacer uso del teorema para deducir que se tienen sus conclusiones para la reducción concreta.

6.2.2 Formalización de la clausura de equivalencia: pruebas abstractas

Veamos ahora cómo definimos en nuestra formalización la clausura de equivalencia descrita por una reducción. Debido a la naturaleza constructiva de la lógica de ACL2, para definir la relación $x \overset{*}{\leftrightarrow} y$ inducida por una reducción $\rightarrow$ es necesario incluir un argumento con la secuencia de pasos $x = x_0 \leftrightarrow x_1 \leftrightarrow x_2 \ldots \leftrightarrow x_n = y$. Tal secuencia de pasos la llamaremos una **prueba abstracta** o simplemente una **prueba** (cuando este término no cree confusión con las pruebas o demostraciones hechas en la lógica de ACL2).

Para representar en ACL2 una prueba abstracta, definimos previamente lo que es un **paso de prueba**. Un paso de prueba simboliza la relación de un objeto con otro mediante aplicación de un paso de reducción, en sentido directo o en sentido inverso. De esta manera, representamos un paso de prueba mediante una estructura `r-step`, con cuatro campos: `direct` (un campo booleano indicando si el paso es directo o no), `operator` (el operador que se aplica) y `elt1`, `elt2` (los elementos que se conectan):

```
(defstructure r-step direct operator elt1 elt2)
```

La macro `defstructure`, desarrollada por Bishop Brock, simula en la lógica de ACL2 al comando `defstruct` de Common Lisp [69]. Esta llamada genera automáticamente la definición de un constructor de estructuras, `make-r-step` y cuatro funciones de acceso a los campos de estas estructuras (con los mismos nombres que los campos). También genera una función reconocedora de este tipo de estructuras, `r-step-p`. En [8], el lector puede encontrar más detalles sobre `defstructure`.

Un paso de prueba se dirá **legal** si uno de sus elementos se obtiene aplicando el operador al otro elemento en el sentido indicado y dicho operador es aplicable al elemento en el que se lleva a cabo la reducción. Esto es precisamente lo que implementa la función `proof-step-p`:

```
(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))))))
```

Una vez tenemos la función que comprueba la legalidad de un paso de prueba, podemos definir de manera precisa qué entendemos por equivalencia entre dos objetos `x` e `y` (pertenecientes al dominio de definición) respecto de una reducción. Esta equivalencia

ha de justificarse mediante una prueba `p`, siendo `p` una secuencia concatenada de pasos de prueba que conectan `x` e `y` (siempre que estos pasos de prueba se realicen dentro del dominio de definición). La función `equiv-p` formaliza tal idea en la lógica de ACL2:

```
(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)))))
```

Si `(equiv-p x y p)` es igual a `t`, decimos que `p` es una prueba que **justifica** la equivalencia entre `x` e `y`. Si `p1` es otra prueba tal que `(equiv-p x y p1)`, decimos entonces que `p` y `p1` son **pruebas equivalentes**. De la definición de `equiv-p` se deduce que si `(equiv-p x y p)`, entonces se verifica `(q x)` y `(q y)`. Es decir, fuera del dominio de definición de la reducción, la función `equiv-p` siempre devuelve `nil`. Además los elementos que aparecen en los pasos de una prueba son siempre elementos del dominio de definición.

Como es de esperar, la definición de `proof-step-p` y de `equiv-p` depende de las funciones `q`, `legal` y `reduce-one-step`. Por tanto, cuando hablemos de una reducción en concreto, se tienen que definir las funciones concretas correspondientes a los pasos de prueba y a la relación de equivalencia.

Ejemplo 6.33 Para la reducción del ejemplo 6.32, la figura 6.6 muestra la definición de las correspondientes funciones `perm-proof-step-p` y `perm-equiv-p`. Estas funciones se definen de manera totalmente análoga a `proof-step-p` y `equiv-p`, respectivamente. Las funciones `natural-true-listp`, `perm-reduce-one-step` y `perm-legal` juegan el papel de `q`, `reduce-one-step` y `legal`, respectivamente.

Es fácil probar (véase subsección `1.2` del libro `confluence.lisp`) que `equiv-p`[1] es la menor relación de equivalencia, definida en en el conjunto de objetos que satisfacen `q`, que contiene a la reducción que representan `legal` y `reduce-one-step`. De esta manera nos aseguramos que estamos formalizando la relación deseada. Por ejemplo, se trata de una relación de equivalencia:

```
(defthm equiv-p-reflexive
  (implies (q x) (equiv-p x x nil)))

(defthm equiv-p-symmetric
  (implies (equiv-p x y p)
           (equiv-p y x (inverse-proof p))))

(defthm equiv-p-transitive
  (implies (and (equiv-p x y p1)
                (equiv-p y z p2))
           (equiv-p x z (append p1 p2))))
```

[1] Rigurosamente hablando, deberíamos decir "la relación *existe* `p` tal que `(equiv-p x y p)`"

```
(defun perm-proof-step-p (s)
   (let ((l1 (elt1 s)) (l2 (elt2 s))
         (operator (operator s)) (direct (direct s)))
     (and
       (r-step-p s)
       (implies direct
              (and (perm-legal l1 operator)
                   (equal (perm-reduce-one-step l1 operator)
                          l2)))
       (implies (not direct)
                (and (perm-legal l2 operator)
                     (equal (perm-reduce-one-step l2 operator)
                            l1))))))

(defun perm-equiv-p (x y p)
  (if (endp p)
      (and (equal x y) (natural-true-listp x))
    (and (natural-true-listp x)
         (perm-proof-step-p (car p))
         (equal x (elt1 (car p)))
         (perm-equiv-p (elt2 (car p)) y (cdr p)))))
```

Figura 6.6: Clausura de equivalencia en el ejemplo de las permutaciones

La manera en que enunciamos estas propiedades nos muestra la ventaja que tiene el hecho de tratar a las pruebas abstractas como objetos con entidad propia, susceptibles de ser transformados. Por ejemplo, en la prueba de la propiedad simétrica, la función `inverse-proof` obtiene la prueba abstracta inversa de una dada. Esta función viene definida de la siguiente manera:

```
(defun inverse-r-step (st)
   (make-r-step
    :direct (not (direct st))
    :elt1 (elt2 st) :elt2 (elt1 st) :operator (operator st)))

(defun inverse-proof (p)
   (if (atom p)
       p
     (append (inverse-proof (cdr p))
             (list (inverse-r-step (car p))))))
```

Por último, también podemos definir funciones que actúan como reconocedores de la forma particular que tienen determinadas pruebas, implementando así las definiciones de 6.4. Estas definiciones las usaremos más adelante para expresar propiedades como la de Church-Rosser o la confluencia local.

```
(defun steps-up (p)
   (if (endp p)
       t
     (and (not (direct (car p))) (steps-up (cdr p)))))

(defun steps-down (p)
   (if (endp p)
       t
     (and (direct (car p)) (steps-down (cdr p)))))

(defun steps-valley (p)
   (cond ((endp p) t)
         ((direct (car p)) (steps-valley (cdr p)))
         (t (steps-up (cdr p)))))

(defun steps-mountain (p)
   (cond ((endp p) t)
         ((direct (car p)) (steps-down (cdr p)))
         (t (steps-mountain (cdr p)))))

(defun local-peak-p (p)
   (and (consp p) (consp (cdr p)) (not (consp (cddr p)))
        (not (direct (car p))) (direct (cadr p))))
```

Las definiciones sobre la forma de las pruebas no dependen de la reducción concreta que se trate en cada caso, ya que sólo atienden a los sentidos en los que se dan los pasos de las pruebas.

6.2.3 Enunciado de algunas propiedades

En esta subsección veremos cómo expresar determinadas propiedades sobre reducciones, siguiendo con la formalización que estamos describiendo. En concreto, veremos cómo definir las propiedades de normalización, Church-Rosser, confluencia, confluencia local y noetherianidad para una reducción abstracta definida por las funciones `q`, `reduce-one-step` y `legal`.

Recordemos que para formalizar un teorema cualquiera sobre reducciones abstractas (como los que se muestran en las tres secciones siguientes) estas propiedades han de ser asumidas como ciertas, si la propiedad se encuentra entre las hipótesis del teorema, o bien han de ser demostradas, si se trata de la conclusión.

Propiedad de normalización

La propiedad de normalización, según la definición 6.7, significa la existencia de una forma normal equivalente para cualquier elemento dado. Puesto que en nuestra formalización toda equivalencia debe ser justificada por la correspondiente prueba, esto significa que existe una función `proof-irreducible`, tal que para cada objeto `x` del dominio de definición, devuelve una prueba que justifica la equivalencia de `x` con otro objeto en forma normal:

```
(defthm normalizing
  (implies (q x)
           (let* ((p-x-y (proof-irreducible x))
                  (y (last-of-proof x p-x-y)))
             (and (equiv-p x y p-x-y)
                  (not (legal y op))))))
```

La función `last-of-proof` que se ha usado en la fórmula anterior, es una función tal que actuando sobre un objeto `x` y una prueba `p`, obtiene el último elemento en la secuencia de elementos de los pasos de la prueba. En el caso particular de que `p` sea vacía, se devuelve `x` (la función `last-elt` devuelve el último elemento de una lista no vacía, en este caso el último paso):

```
(defun last-of-proof (x p)
   (if (endp p) x (elt2 (last-elt p))))
```

Propiedades de Church-Rosser, confluencia y confluencia local

La propiedad de Church-Rosser, tal y como se define en 6.8, puede ser reformulada mediante el concepto de prueba valle: una reducción tiene la propiedad de Church-Rosser si y sólo si para toda prueba existe una prueba valle equivalente. La falta de cuantificación existencial de la lógica de ACL2 se suple aquí suponiendo la existencia de una función, que llamamos `transform-to-valley`, que recibiendo una prueba `p`, devuelve una prueba valle equivalente, tal y como queda establecido por la siguiente fórmula:

```
(defthm Chuch-Rosser-property
  (let ((valley (transform-to-valley p)))
    (implies (equiv-p x y p)
             (and (steps-valley valley)
                  (equiv-p x y valley)))))
```

La misma idea se puede usar para expresar la confluencia de una reducción, ya que esto significa la existencia de una función `transform-mountain-to-valley` que recibiendo como entrada una prueba `p`, devuelve una prueba valle equivalente:

```
(defthm confluence
  (let ((valley (transform-mountain-to-valley p)))
    (implies (and (equiv-p x y p) (steps-mountain p))
             (and (steps-valley valley)
                  (equiv-p x y valley))))))
```

Y de la misma manera, a través de una función `transform-local-peak`, se expresa la confluencia local:

```
(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)))))
```

Demostrar alguna de estas propiedades requiere *definir* la correspondiente función de transformación de pruebas y *demostrar* las propiedades requeridas sobre equivalencia de pruebas y su forma.

Ejemplo 6.34 La reducción descrita por `perm-reduce-one-step` y `perm-legal` (ejemplo 6.32) es localmente confluente, tal y como muestran los eventos de la figura 6.7. En primer lugar, definimos la función `perm-transform-local-peak`, guiándonos por la prueba a mano presentada en el ejemplo 6.27. Una vez definida la función, la confluencia local de la reducción queda demostrada mediante el teorema `perm-local-confluence`, que afirma que la función `perm-transform-local-peak` devuelve una prueba valle equivalente para cualquier pico local.

Noetherianidad

Nuestra formalización de la noetherianidad está basada en el siguiente (meta) teorema (véase el teorema A.11): una reducción es noetheriana si y sólo si está contenida en la inversa de una relación bien fundamentada. Por tanto, la propiedad de noetherianidad significa que existe una relación bien fundamentada tal que al aplicar cualquier paso legal de reducción a un elemento (en el dominio de definición), se obtiene otro elemento menor respecto de tal relación bien fundamentada.

Como ya se vió en la sección 5.2.1, la buena fundamentación de una relación binaria `rel` queda expresada mediante la siguiente fórmula. Recuérdese que la función `fn` es la función de inmersión que justifica la buena fundamentación de `rel`. En este caso, estamos considerando que la propiedad de medida viene definida por la función `q`. Es decir, la siguiente propiedad establece que `rel` es una relación bien fundamentada en el dominio de definición de la reducción:

```
(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)
```

La relación bien fundamentada `rel` se puede usar para enunciar la noetherianidad de una reducción representada mediante las funciones `q`, `reduce-one-step` y `legal`. Simplemente hay que expresar que mediante la aplicación de un operador legal se obtiene un objeto menor respecto de `rel`:

```
(defthm noetherian
  (implies (and (q x) (legal x u) (q (reduce-one-step x u)))
           (rel (reduce-one-step x u) x)))
```

Por tanto, para probar la noetherianidad de una reducción concreta, debemos definir la correspondiente relación bien fundamentada y probar un teorema análogo a `noetherian`.

Ejemplo 6.35 La reducción del ejemplo 6.32 es noetheriana como muestran los eventos de la figura 6.8. La función `disorders` cuenta el número de desordenes relativos entre

```
(defun transform-disjoint-perm-local-peak (i j l1 l2)
  (list (make-r-step :direct t :operator j
                     :elt1 l1 :elt2 (perm-reduce-one-step l1 j))
        (make-r-step :direct nil :operator i
                     :elt1 (perm-reduce-one-step l2 i) :elt2 l2)))

(defun transform-contiguous-perm-local-peak
  (i j l1 l2)
  (list
   (make-r-step :direct t :operator j
                :elt1 l1 :elt2 (perm-reduce-one-step l1 j))
   (make-r-step :direct t :operator i
                :elt1 (perm-reduce-one-step l1 j)
                :elt2 (perm-reduce-one-step
                         (perm-reduce-one-step l1 j) i))
   (make-r-step :direct nil :operator j
                :elt1 (perm-reduce-one-step
                         (perm-reduce-one-step l2 i) j)
                :elt2 (perm-reduce-one-step l2 i))
   (make-r-step :direct nil :operator i
                :elt1 (perm-reduce-one-step l2 i) :elt2 l2)

(defun perm-transform-local-peak (p)
  (let* ((step1 (first p)) (step2 (second p))
         (i (operator step1)) (j (operator step2))
         (l1 (elt1 step1)) (l2 (elt2 step2)))
    (cond ((= i j) nil)
          ((or (< (+ i 1) j) (< (+ j 1) i))
           (transform-disjoint-perm-local-peak i j l1 l2))
          (t (transform-contiguous-perm-local-peak i j l1 l2)))))

(defthm perm-local-confluence
  (let ((valley (perm-transform-local-peak p)))
    (implies (and (perm-equiv-p x y p) (local-peak-p p))
             (and (steps-valley valley)
                  (perm-equiv-p x y valley)))))
```

Figura 6.7: Confluencia local en el ejemplo de las permutaciones

pares de elementos de una lista (como la función d en el ejemplo 6.19). Definimos la relación binaria `perm-rel` como aquella que compara dos listas atendiendo al valor que toma la función `disorders` en cada una de ellas. El teorema `perm-rel-well-founded` muestra la buena fundamentación de `perm-rel` (nótese que no es necesaria la hipótesis `(natural-true-listp l)`). El teorema `perm-noetherian` demuestra la noetherianidad de la reducción representada por las funciones `perm-reduce-one-step` y `perm-legal`.

```
(defun disorders-x (x l)
  (cond ((endp l) 0)
        ((< (car l) x) (+ 1 (disorders-x x (cdr l))))
        (t (disorders-x x (cdr l)))))

(defun disorders (l)
  (if (endp l)
      0
    (+ (disorders-x (car l) (cdr l)) (disorders (cdr l)))))

(defun perm-rel (l1 l2)
  (< (disorders l1) (disorders l2)))

(defthm perm-rel-well-founded
  (and (e0-ordinalp (disorders x))
       (implies (perm-rel x y)
                (e0-ord-< (disorders x) (disorders y))))
  :rule-classes :well-founded-relation )

(defthm perm-noetherian
  (implies (perm-legal l i)
           (perm-rel (perm-reduce-one-step l i) l)))
```

Figura 6.8: Noetherianidad en el ejemplo de las permutaciones

6.3 Reducciones Church-Rosser y normalizadoras

En esta sección presentamos una formalización del teorema 6.16, en el que se demuestra la existencia de un algoritmo de decisión para la relación de equivalencia inducida por una reducción normalizadora, con la propiedad de Church-Rosser. Este resultado se usará más adelante para probar un teorema de decidibilidad análogo para reducciones convergentes. Los eventos que conducen a los resultados en esta sección se encuentran en el libro ACL2 `confluence.lisp`. Este libro incluye el libro `abstract-proofs.lisp`, que contiene definiciones y resultados básicos sobre pruebas abstractas. Los nombres del fichero `confluence.lisp` se definen dentro del paquete `CNF`.

Para demostrar en ACL2 un teorema análogo al teorema 6.16, debemos en primer lugar asumir como hipótesis la existencia de una reducción normalizadora con la propiedad de Church-Rosser. Para ello, usamos un encapsulado, mediante el cual podemos asumir la existencia de una reducción abstracta con las dos propiedades mencionadas. Estas propiedades se expresan tal y como se ha explicado en la sección anterior. La figura 6.9 muestra el encapsulado completo que se ha usado para asumir que las funciones `q`, `reduce-one-step` y `legal` representan una reducción normalizadora y Church-Rosser (hemos omitido las definiciones locales). La función `equiv-p` (y por tanto `proof-step-p`) ha de ser definida dentro del ámbito del `encapsulate`, ya que es necesaria para expresar las propiedades.

El teorema 6.16 quedará demostrado si definimos una función, que llamaremos `r-`

```
(encapsulate
 ((q (x) boolean)
  (legal (x u) boolean)
  (reduce-one-step (x u) element)
  (transform-to-valley (x) valley-proof)
  (proof-irreducible (x) proof))
 ....
 (defun proof-step-p (s)
   (let ((elt1 (elt1 s)) (elt2 (elt2 s))
         (op (operator s)) (dt (direct s)))
     (and (r-step-p s)
          (implies dt (and (legal elt1  op)
                           (equal (reduce-one-step elt1 op)
                                  elt2)))
          (implies (not dt) (and (legal elt2 op)
                                 (equal (reduce-one-step elt2 op)
                                        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 Chuch-Rosser-property
   (let ((valley (transform-to-valley p)))
     (implies (equiv-p x y p)
              (and (steps-valley valley) (equiv-p x y valley)))))

 (defthm normalizing
   (implies (q x)
            (let* ((p-x-y (proof-irreducible x))
                   (y (last-of-proof x p-x-y)))
              (and (equiv-p x y p-x-y)
                   (not (legal y op)))))))
```

Figura 6.9: Reducción normalizadora y Church-Rosser

`-equiv` y probamos que implementa un algoritmo de decisión correcto y completo para la clausura de equivalencia generada por la reducción representada por la terna de funciones `q`, `reduce-one-step` y `legal`. Como se ha visto en la sección 6.1, este algoritmo debe comprobar la igualdad de las formas normales de los elementos que se reciben como entrada. Por tanto, previo a la definición del algoritmo de decisión, hemos de definir una

función que calcule la forma normal de un elemento dado `x`. Basta para ello tomar el último elemento de la prueba `(proof-irreducible x)`:

```
(defun normal-form (x)
  (last-of-proof x (proof-irreducible x)))
```

La función `normal-form` realmente implementa el concepto de forma normal de un elemento, tal y como demuestran las siguientes propiedades. La función `normal-form` se corresponde, por tanto, con la función n del enunciado del teorema 6.16.

```
(defthm irreducible-normal-form
  (implies (q x)
           (not (legal (normal-form x) op))))

(defthm equivalent-normal-form
  (implies (q x)
           (equiv-p x (normal-form x) (proof-irreducible x)))
```

Una vez se ha definido el concepto de forma normal, podemos definir la función `r-equiv`, como la función que comprueba si dos elementos tienen la misma forma normal:

```
(defun r-equiv (x y)
  (equal (normal-form x) (normal-form y)))
```

Para demostrar el teorema deseado, bastará ver que `r-equiv` es un algoritmo de decisión para la clausura de equivalencia y por tanto hemos de demostrar su corrección y completitud. Por completitud entendemos aquí la propiedad que establece que `(r-equiv x y)` nunca devuelve `nil` cuando existe una prueba que justifica la equivalencia de dos elementos del dominio de definición, `x` e `y`. Es decir:

```
(defthm r-equiv-complete
  (implies (equiv-p x y p) (r-equiv x y)))
```

La corrección del algoritmo significa que si `(r-equiv x y)` no toma el valor `nil` sobre dos elementos `x` e `y` del dominio de definición, entonces `x` e `y` son equivalentes. Es decir, si `x` e `y` tienen la misma forma normal, entonces existe una prueba que justifica su equivalencia. Tal prueba se obtiene mediante la función `make-proof-common-n-f`:

```
(defun make-proof-common-n-f (x y)
   (append (proof-irreducible x) (inverse-proof (proof-irreducible y))))
```

Con esta definición, el teorema de corrección del algoritmo `r-equiv` se formula de la siguiente manera:

```
(defthm r-equiv-sound
  (implies (and (q x) (q y) (r-equiv x y))
           (equiv-p x y (make-proof-common-n-f x y))))
```

En el apéndice C, subsección C.4.1, discutimos los aspectos más destacados de la prueba automática de los teoremas `r-equiv-complete` y `r-equiv-sound`.

6.4 El lema de Newman

En esta sección presentamos una demostración en la lógica de ACL2 del lema de Newman (teorema 6.24), que afirma la confluencia de toda reducción noetheriana y localmente confluente. Este resultado es importante para el estudio de la decidibilidad de ciertas teorías ecuacionales, como se verá en el capítulo 7. Además se trata, como veremos, de un interesante ejemplo del uso de las relaciones entre multiconjuntos para probar terminación de funciones.

Los eventos necesarios para la demostración del lema de Newman se encuentran en el libro `newman.lisp`. Este libro a su vez incluye el libro `defmul.lisp` (para poder usar la macro `defmul`) y el libro `abstract-proofs.lisp`. Los nombres del libro `newman.lisp` se definen dentro del paquete `NWM`.

6.4.1 Formalización

En la figura 6.10 aparecen los eventos necesarios para enunciar las hipótesis del lema de Newman. Como en la sección anterior, usamos `encapsulate` para asumir ciertas propiedades: en este caso, que las funciones `q`, `reduce-one-step` y `legal` representan una reducción abstracta noetheriana y localmente confluente[2]. Hemos omitido las definiciones locales de las funciones que se introducen en el encapsulado.

En primer lugar, se presenta una relación bien fundamentada `rel`, en el dominio de definición de la reducción, que va a servir para justificar la noetherianidad de la reducción, tal y como se explica en la subsección 6.2.3. Hemos asumido, además, que la relación `rel` posee la propiedad transitiva. Desde el punto de vista teórico, esto no supone ninguna restricción, ya que si una relación está bien fundamentada, también lo es su clausura transitiva. En la práctica, necesitamos que `rel` satisfaga esta propiedad para poder demostrar el teorema, como se verá en la subsección siguiente. Como en general no es posible definir la clausura transitiva de una relación cualquiera, asumiremos directamente que `rel` es transitiva.

Ejemplo 6.36 En bastante usual que la relación que justifica la noetherianidad de una reducción concreta cumpla la propiedad transitiva. Por ejemplo, la relación `perm-rel` que justifica la noetherianidad en el ejemplo de las permutaciones es transitiva:

```
(defthm perm-rel-transitive
  (implies (and (perm-rel l1 l2) (perm-rel l2 l3))
           (perm-rel l1 l3)))
```

La segunda parte del encapsulado presenta una reducción noetheriana y localmente confluente. La noetherianidad se justifica mediante la relación `rel`. La confluencia local queda definida asumiendo la existencia de una función `transform-local-peak` que asegura, a su vez, la existencia de una prueba valle equivalente para cada posible pico local, tal y como se describió en la subsección 6.2.3. Para una mayor claridad, hemos omitido las definiciones de `proof-step-p` y `equiv-p`, ya que éstas son exactamente iguales que las presentadas en la subsección 6.2.2. Nuevamente, estas definiciones deben estar dentro

[2] Recuérdese que para evitar conflictos con los nombres usados en la sección anterior, usamos un paquete de símbolos diferente.

```
(encapsulate
 ((rel (x y) booleanp) (fn (x) e0-ordinalp)
  (q (x) booleanp) (legal (x u) boolean)
  (reduce-one-step (x u) element)
  (transform-local-peak (x) proof))

 (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)

 (defthm rel-transitive
   (implies (and (q x) (q y) (q z) (rel x y) (rel y z))
            (rel x z)))
 ...
 (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 (legal x u) (q x) (q (reduce-one-step x u)))
            (rel (reduce-one-step x u) x))))
```

Figura 6.10: Hipótesis del lema de Newman

del ámbito del encapsulado, ya que se necesitan para expresar la propiedad de confluencia local.

Para probar el lema de Newman, debemos demostrar que la reducción que representan la terna de funciones `q`, `reduce-one-step` y `legal` es confluente. Sin embargo, probaremos una propiedad equivalente, como es la propiedad de Church-Rosser (teorema 6.22). En la demostración presentada en la sección 6.1, la noción de confluencia aparecía como un concepto más simple equivalente a la propiedad de Church-Rosser, que hacía más fácil la demostración del lema de Newman. En la demostración que presentaremos, nos será más fácil manejar esta última propiedad que la noción de confluencia. Además, de esta manera, no necesitaremos probar la equivalencia de ambos conceptos.

Según lo discutido en la subsección 6.2.3, para probar que la reducción tiene la propiedad de Church-Rosser, *debemos definir* una función `transform-to-valley` y *demostrar* que `(transform-to-valley p)` es una prueba valle equivalente a `p`. Así, el lema de Newman quedará probado si se demuestra el siguiente teorema:

```
(defthm Newman-lemma
  (let ((valley (transform-to-valley p)))
```

```
(implies (equiv-p x y p)
         (and (steps-valley valley)
              (equiv-p x y valley)))))
```

En lo que sigue, presentaremos una definición adecuada de `transform-to-valley` y describiremos una demostración de este teorema. La dificultad principal radica en mostrar la terminación de la definición, para lo cual usaremos una relación bien fundamentada entre multiconjuntos.

6.4.2 Demostración del lema de Newman

La demostración del lema de Newman que usualmente se encuentra en la literatura sobre reducciones abstractas es la que se ha presentado en 6.24. En esta demostración, se emplea inducción noetheriana basada en la noetherianidad de la reducción. Sin embargo, la formulación en ACL2 es distinta y por tanto también será distinta la demostración, ya que debemos definir definir una función `transform-to-valley` y probar que verifica determinadas propiedades. La demostración formal que hemos realizado en ACL2 está basada en una prueba del lema de Newman presentada por Klop en [43]. Describamos a continuación los puntos principales de la misma.

Centrémonos en primer lugar en la definición de la función `transform-to-valley`. Para ello podemos usar la función `transform-local-peak`, la cual hemos asumido que transforma picos locales en pruebas valle equivalentes. Dada una prueba que no sea un valle, podemos reemplazar uno de sus picos locales por una subprueba valle equivalente. Este proceso se puede aplicar iterativamente hasta que no existan más picos locales. Formalmente:

```
(defun exists-local-peak (p)
  (cond ((or (atom p) (atom (cdr p))) nil)
        ((and (not (direct (car p)))
              (direct (cadr p)))
          (and (proof-step-p (car p))
               (proof-step-p (cadr p))
               (q (elt1 (car p)))
               (q (elt2 (car p))) (q (elt2 (cadr p)))
               (equal (elt2 (car p)) (elt1 (cadr p)))))
        (t (exists-local-peak (cdr p)))))

(defun replace-local-peak (p)
  (cond ((or (atom p) (atom (cdr p))) nil)
        ((and (not (direct (car p))) (direct (cadr p)))
         (append (transform-local-peak (list (car p) (cadr p)))
                 (cddr p)))
        (t (cons (car p) (replace-local-peak (cdr p))))))

;(defun transform-to-valley (p)
;  (if (exists-local-peak p)
;      (transform-to-valley (replace-local-peak p))
;      p))
```

Esta definición de `transform-to-valley` no se admite en la lógica de ACL2 sin ayuda por parte del usuario, ya que el probar su terminación no es en absoluto trivial. La razón es que cuando un pico local de una prueba se sustituye por una subprueba valle equivalente, la longitud de la prueba que se obtiene puede ser mayor que la longitud de la prueba original.

Sin embargo, si todos los elementos de la prueba verifican `q`, cada elemento de la nueva subprueba es menor, respecto de la relación bien fundamentada `rel`, que el mayor elemento del pico local reemplazado. Si medimos una prueba por el multiconjunto de los elementos involucrados en la misma, entonces al reemplazar un pico local por una subprueba valle equivalente, obtenemos una prueba cuya medida asociada es un multiconjunto menor, respecto de la relación entre multiconjuntos inducida por `rel`. Puesto que `rel` está bien fundamentada sobre el conjunto definido por el predicado `q`, la relación entre multiconjuntos está bien fundamentada sobre los multiconjuntos cuyos elementos satisfacen `q` (según lo visto en el capítulo 5). Esto justifica la terminación de `transform-to-valley`. A continuación formalizamos este razonamiento.

La función `proof-measure` devuelve la medida asociada a una prueba, el multiconjunto de los elementos almacenados en el campo `elt1` de cada uno de los pasos de la prueba:

```
(defun proof-measure (p)
  (if (endp p)
      nil
    (cons (elt1 (car p)) (proof-measure (cdr p)))))
```

Sólo en el caso de que `p` represente una secuencia de pasos cuyos elementos satisfacen `q`, podremos asegurar que `(proof-measure p)` devuelve un multiconjunto de elementos que satisfacen `q`. Es por esto que definimos la siguiente función que implementa dicho concepto, que más adelante necesitaremos:

```
(defun steps-q (p)
  (if (endp p)
      t
    (and (r-step-p (car p))
         (q (elt1 (car p))) (q (elt2 (car p)))
         (steps-q (cdr p)))))
```

Usando la herramienta `defmul` descrita en el capítulo 5, podemos definir la relación bien fundamentada entre multiconjuntos `mul-rel`, inducida por la relación `rel`:

```
(acl2::defmul (rel rel-well-founded-relation-on-q q fn x y))
```

Como ya se comentó en el capítulo 5, esta demostración del lema de Newman constituye la principal aplicación de nuestra herramienta `defmul` y motivó originalmente su desarrollo. Como se observa, permite definir de manera sencilla la relación entre multiconjuntos `mul-rel`, inducida por la relación `rel` y probar que se trata de una relación bien fundamentada en el conjunto de las listas cuyos elementos satisfacen `q`.

El lema que se presenta a continuación, es el resultado principal que hemos necesitado en la demostración del lema de Newman. Establece que la medida asignada a una prueba decrece (respecto de la relación `mul-rel`) si un pico local se reemplaza por una subprueba valle equivalente:

```
(defthm transform-to-valley-admission
  (implies (exists-local-peak p)
           (mul-rel (proof-measure (replace-local-peak p))
                    (proof-measure p))))
```

Con este resultado la admisión de la función `transform-to-valley` es ahora posible, sin más que proporcionar el consejo adecuado:

```
(defun transform-to-valley (p)
  (declare (xargs :measure (if (steps-q p) (proof-measure p) nil)
                  :well-founded-relation mul-rel))
  (if (and (steps-q p) (exists-local-peak p))
      (transform-to-valley (replace-local-peak p))
      p))
```

Merece la pena comentar dos detalles técnicos relativos a la admisión de esta función. Nótese en primer lugar que la medida viene dada por la expresión `(if (steps-q p) (proof-measure p) nil)`, en lugar de usar simplemente `(proof-measure p)`. La razón es que debemos proporcionar una medida tal que a todo objeto `p` se le asigne un objeto que se encuentre dentro del conjunto en el que `mul-rel` está bien fundamentada: es decir, la medida debe ser un multiconjunto de elementos que satisfagan `q`. En segundo lugar y como consecuencia de lo anterior, la condición en la que se produce la llamada recursiva debe incluir ahora `(steps-q p)`, ya que sólo en ese caso la medida asociada es `(proof-measure p)`, que es justamente lo que decrece respecto a la relación bien fundamentada `mul-rel`, (como afirma el teorema `transform-to-valley-admission`). El añadir esta condición no supone, sin embargo, ninguna restricción ya que sólo nos interesa el comportamiento de `transform-to-valley` sobre objetos que representen pruebas y éstas por definición (véase `equiv-p`) verifican la propiedad `steps-q` y además la función `replace-local--peak` conserva tal propiedad.

Una vez que se ha admitido la función `transform-to-valley`, para completar la demostración del lema de Newman queda por demostrar sus dos propiedades fundamentales. Es decir, que la prueba transformada es una prueba valle y que es equivalente a la original. Los dos teoremas siguientes muestran estas dos propiedades e implican trivialmente el resultado `Newman-lemma`, tal y como se ha presentado en la subsección anterior.

```
(defthm equiv-p-x-y-transform-to-valley
  (implies (equiv-p x y p)
           (equiv-p x y (transform-to-valley p))))

(defthm valley-transform-to-valley
   (implies (equiv-p x y p)
            (steps-valley (transform-to-valley p))))
```

Es importante destacar que la prueba de estos dos teoremas se realiza por inducción en la relación multiconjuntista `mul-rel`, frente a la demostración estándar presentada en 6.24, que usa inducción en la reducción noetheriana (es decir, inducción en la relación `rel`).

La demostración automática del lema de Newman es la más laboriosa de las presentadas en este capítulo (en particular el lema `transform-to-valley-admission`), debido a la cantidad de lemas previos que es necesario probar. En el apéndice C, subsección C.4.2 se describe con detalle la interacción con el demostrador, para llegar a la prueba automática del resultado.

6.5 Reducciones convergentes: decidibilidad

En esta sección presentamos la formalización en ACL2 del teorema 6.29, en el que se demuestra la decidibilidad de la relación de equivalencia descrita por una reducción convergente (siempre que se disponga de un test de reducibilidad). Este teorema se obtiene como consecuencia del lema de Newman y de la decidibilidad de la relación de equivalencia descrita por una reducción normalizadora y Church-Rosser, resultados cuya formalización y prueba se ha descrito en las dos secciones anteriores. Es por esto que este resultado constituye un buen ejemplo del uso de instanciación funcional en ACL2. Los eventos que se necesitan para la demostración de los teoremas presentados se encuentran en el libro ACL2 `convergent.lisp`. Los nombres de este libro se definen dentro del paquete `CNV`.

6.5.1 Formalización del resultado

La hipótesis del teorema 6.29 asume la existencia de una reducción convergente (es decir, noetheriana y localmente confluente) y de un test de reducibilidad para la misma. En la figura 6.11 se muestran los eventos necesarios para describir tales propiedades. Como en las secciones anteriores, usamos un encapsulado para asumir que las funciones `q`, `reduce-one-step` y `legal` representan una reducción convergente para la cual existe un test de reducibilidad[3].

Por lo que respecta a las propiedades de noetherianidad y confluencia local, la formalización es análoga a la de la sección anterior, por lo que son validos aquí todos los comentarios de la subsección 6.4.1. Sin embargo existen dos detalles nuevos en este caso.

En primer lugar, hemos asumido la existencia de un elemento en el dominio de definición de la reducción, que hemos llamado `(q-w)`. Más adelante comentaremos por qué necesitamos suponer explícitamente la existencia de al menos un elemento en el conjunto definido por `q`. Esto no supone ninguna restricción a la generalidad del resultado, ya que si el dominio de definición de la reducción fuera vacío los resultados presentados serían trivialmente ciertos.

Por otro lado, para probar la propiedad de normalización, debemos definir una función `proof-irreducible`, con propiedades análogas a las que se asumieron para tal función en la sección 6.3. Como se verá más adelante, tal función será obtenida aplicando pasos de reducción hasta que se llega a un elemento irreducible. Para que la secuencia de tales pasos constituya una prueba, debemos suponer que todo operador legal, al aplicarse sobre un elemento del dominio de definición de la reducción, obtiene un elemento en el mismo dominio de definición:

```
(defthm legal-reduce-one-step-closure
   (implies (and (q x) (legal x op))
            (q (reduce-one-step x op))))
```

[3] Nuevamente usamos distintos paquetes para evitar conflictos entre nombres.

```
(encapsulate
 ((rel (x y) boolean) (fn (x) e0-ordinalp)
  (q (x) boolean) (q-w () elemement)
  (legal (x u) boolean) (reducible (x) boolean)
  (reduce-one-step (x u) element)
  (transform-local-peak (x) proof))
 ...
 (defthm one-element-of-q (q (q-w)))

 (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)

 (defthm rel-transitive
   (implies (and (q x) (q y) (q z) (rel x y) (rel y z))
            (rel x z)))
 ...
 (defthm legal-reduce-one-step-closure
   (implies (and (q x) (legal x op))
            (q (reduce-one-step x op))))

 (defthm reducible-implies-legal
   (implies (and (q x) (reducible x))
            (legal x (reducible x))))

 (defthm not-reducible-nothing-legal
   (implies (and (q x) (not (reducible x)))
            (not (legal x u))))
 ...
 (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))))
```

Figura 6.11: Reducción convergente con test de reducibilidad

Tal propiedad tampoco supone ninguna restricción en la práctica, ya que parece bastante obvio que toda reducción concreta debe verificarla (sin embargo, esta propiedad no se necesitaba para probar el lema de Newman).

Por lo que respecta a la existencia de un test de reducibilidad, asumiremos que existe una función llamada `reducible` con la siguiente propiedad: para elementos `x` del dominio de definición, `(reducible x)` devuelve un operador aplicable a `x` siempre que `x` no esté en forma normal y `nil` en caso contrario[4]. Formalmente:

```
(defthm reducible-implies-legal
  (implies (and (q x) (reducible x))
           (legal x (reducible x))))

(defthm not-reducible-nothing-legal
  (implies (and (q x) (not (reducible x)))
           (not (legal x op))))
```

Esta función `reducible` no es exactamente un test de reducibilidad tal y como se define en 6.28, pero junto con la función `reduce-one-step` permitiría definir trivialmente un test de reducibilidad en el sentido de la definición 6.28: si `(q x)` y `(reducible x)`, entonces devolver `(reduce-one-step x (reducible x))`, en caso contrario devolver `nil`.

Ejemplo 6.37 En la figura 6.12 se presenta un test de reducibilidad para el ejemplo de las permutaciones, junto con los teoremas establecen que efectivamente lo es.

```
(defun perm-reducible-aux (l i)
  (cond ((or (endp l) (endp (cdr l))) nil)
        ((> (car l) (cadr l)) i)
        (t (perm-reducible-aux (cdr l) (+ i 1)))))

(defun perm-reducible (l)
  (and (natural-true-listp l)
       (perm-reducible-aux l 0)))

(defthm perm-reducible-implies-perm-legal
  (implies (perm-reducible l)
           (perm-legal l (perm-reducible l))))

(defthm not-perm-reducible-nothing-perm-legal
  (implies (not (perm-reducible l))
           (not (perm-legal l op))))
```

Figura 6.12: Un test de reducibilidad en el ejemplo de las permutaciones

Para probar la decidibilidad de la relación de equivalencia descrita por la reducción correspondiente a la terna de funciones `q`, `reduce-one-step` y `legal`, debemos definir un algoritmo de decisión para la misma, demostrando que es correcto y completo. Como en

[4]Esto nos obliga a representar los operadores de cualquier reducción concreta de manera que `nil` no sea la representación de ningún operador.

la sección 6.3, este algoritmo se limita a comprobar la igualdad de las formas normales de los dos elementos que se reciben como entrada. Debemos definir por tanto una función que calcule formas normales, usando para ello la función `reducible`:

```
(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))
```

En la prueba de terminación de esta función se necesita la buena fundamentación de la relación `rel`. Puesto que asumimos que `rel` está bien fundamentada en el conjunto definido por el predicado `q`, la medida que usamos en la prueba de terminación viene dada por la expresión `(if (q x) x (q-w))`. Es aquí donde es necesario disponer explícitamente de un elemento que satisfaga la propiedad `q`, en nuestro caso `(q-w)`: este elemento será asignado por esta medida a aquellos elementos que no verifiquen `q`.

Otro punto importante que debemos señalar es la inclusión de la comprobación `(q x)` en la definición de la función `normal-form`. Esta comprobación es necesaria, ya que (obviamente) sólo hemos asumido la noetherianidad de la reducción en su dominio de definición. Si no incluyéramos tal comprobación, no sería posible la prueba de terminación de la función `normal-form` en la lógica de ACL2 y por tanto la función no sería admisible. Desgraciadamente, esto tiene un efecto negativo sobre la eficiencia de la función[5]: en cada llamada recursiva se realiza la comprobación de que el argumento de entrada está en el dominio de definición, aún cuando bastaría con comprobarlo sólo en la primera vez. Comentaremos más sobre este aspecto en la sección 7.3.4, al instanciar este marco abstracto para el caso concreto de los sistemas de reescritura de términos.

Una vez definida la función `normal-form`, el algoritmo de decisión viene dado por la función `r-equivalent`:

```
(defun r-equivalent (x y)
  (equal (normal-form x) (normal-form y)))
```

Veamos ahora los principales resultados de esta sección, que demuestran la decidibilidad de la relación de equivalencia descrita por una reducción noetheriana y localmente confluente. En primer lugar, la completitud del algoritmo `r-equivalent` se expresa mediante el siguiente teorema, que establece que si existe una prueba que justifica la equivalencia de dos objetos, entonces `r-equivalent` devuelve `t` actuando sobre tales elementos:

```
(defthm r-equivalent-complete
  (implies (equiv-p x y p)
           (r-equivalent x y)))
```

[5] En realidad, sobre posibles instancias concretas y ejecutables de esta función.

La corrección del algoritmo queda establecida por el siguiente teorema: si `r-equivalent` aplicado a un par de objetos del dominio de definición devuelve `t`, entonces existe una prueba abstracta que justifica la equivalencia de tales objetos. Tal prueba viene dada por una función `make-proof-common-n-f` (cuya definición daremos en la subsección siguiente):

```
(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))))
```

La demostración de ambos teoremas prueba la existencia de un algoritmo de decisión para la relación de equivalencia descrita por la reducción. Tal demostración se obtiene de manera fácil a partir de los resultados de las dos secciones anteriores, haciendo uso de la regla de inferencia de instanciación funcional. En la siguiente sección lo explicamos.

6.5.2 Descripción de la demostración

La corrección y completitud del algoritmo descrito por `r-equivalent` se va a obtener como consecuencia del resultado, descrito en la sección 6.3, que afirma la decidibilidad de la relación de equivalencia descrita por una reducción normalizadora con la propiedad de Church-Rosser. Para poder usar este resultado, debemos probar que se cumplen las hipótesis de dicho teorema: es decir, debemos probar que tenemos una reducción normalizadora con la propiedad de Church-Rosser.

Para probar que la reducción es normalizadora, hemos de definir la siguiente función `proof-irreducible`, que obtiene una prueba con los sucesivos pasos de prueba realizados hasta obtener una forma normal:

```
(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))
```

La admisión de esta función es análoga a la de la función `normal-form`. Nótese nuevamente que es necesario incluir la comprobación `(q x)` en su definición. Probar que la reducción es normalizadora consiste en probar que `(proof-irreducible x)` es una prueba que justifica la equivalencia de `x` con un elemento irreducible, para cada `x` en el dominio de definición (subsección 6.2.3). Es lo que establece el siguiente teorema:

```
(defthm normalizing
  (implies (q x)
```

```
          (let* ((p-x-y (proof-irreducible x))
                 (y (last-of-proof x p-x-y)))
            (and (equiv-p x y p-x-y)
                 (not (legal y op))))))
```

En cuanto a la propiedad de Church-Rosser, podemos usar el lema de Newman, ya que estamos asumiendo que tenemos una reducción convergente. Por tanto es posible definir una función `transform-to-valley` exactamente igual que en la subsección 6.4.2 y demostrar (ahora directamente mediante instanciación funcional) que verifica las propiedades que establecen la propiedad de Church-Rosser de la reducción, expresadas por el siguiente teorema:

```
(defthm Chuch-Rosser-property
  (let ((valley (transform-to-valley p)))
    (implies (equiv-p x y p)
             (and (steps-valley valley)
                  (equiv-p x y valley)))))
```

Hemos probado, pues, que se tienen las condiciones del teorema de decidibilidad descrito en la sección 6.3. Esto nos permite concluir, nuevamente mediante instanciación funcional, la corrección y completitud de un algoritmo que decide la clausura de equivalencia. Sin embargo, el algoritmo de decisión que se daba en la sección 6.3 no era exactamente el descrito por `r-equivalent`, ya que la forma normal se definía allí como el último elemento de la prueba devuelta por `proof-irreducible`. Es decir, el resultado descrito en la sección 6.3, aplicado a este caso, lo que afirma es que el siguiente algoritmo `r-equiv` es correcto y completo.

```
(defun normal-form-aux (x) (last-of-proof x (proof-irreducible x)))

(defun r-equiv (x y) (equal (normal-form-aux x) (normal-form-aux y)))
```

La corrección y completitud del algoritmo `r-equiv` vendrían expresadas de igual manera que los teoremas `r-equiv-complete` y `r-equiv-sound` de la sección 6.3, pero ahora se demuestran mediante instanciación funcional. Recuérdese que la prueba que justifica la corrección del algoritmo viene definida por:

```
(defun make-proof-common-n-f (x y)
   (append (proof-irreducible x) (inverse-proof (proof-irreducible y))))
```

Es evidente que `r-equivalent` es más "eficiente" que `r-equiv`[6], ya que no es necesario generar previamente una prueba abstracta para obtener una forma normal. Teniendo la corrección y completitud de `r-equiv` por instanciación funcional, las mismas propiedades se tendrán para `r-equivalent` si probamos que se trata de la misma función. Pero esto es evidente por el siguiente teorema, que afirma que ambas formas de calcular una forma normal son iguales:

```
(defthm normal-form-aux-normal-form
   (equal (normal-form x) (normal-form-aux x)))
```

[6] Aunque, en realidad, ninguna de ellas es ejecutable.

De esta manera llegamos a la demostración de los teoremas `r-equivalent-sound` y `r-equivalent-complete`, formalizando así un teorema análogo al 6.29.

En el apéndice C, subsección C.4.3 se describe con detalle el uso de instanciación funcional para obtener estos resultados.

6.5.3 Un ejemplo

Como conclusión a los resultados vistos hasta ahora, usaremos el resultado descrito en esta sección para probar la decidibilidad de la relación de equivalencia descrita por la reducción $\rightarrow_P$. Recuérdese que tal relación de equivalencia es la relación "ser permutación de". A lo largo de los ejemplos anteriores, se ha visto que `natural-true-listp`, `perm-reduce--one-step` y `perm-legal` describen la reducción $\rightarrow_P$ y que tal reducción es localmente confluente (justificado por la función `perm-transform-local-peak`), noetheriana (justificado por el orden bien fundamentada `perm-rel`) y con un test de reducibilidad (dado por la función `perm-reducible`).

Estamos por tanto en las condiciones del resultado descrito en esta sección. Podemos, mediante instanciación funcional, mostrar la decidibilidad de la relación $\overset{*}{\leftrightarrow}_P$. El algoritmo de decisión consiste en ordenar las dos listas (es decir, calcular las formas normales) y comparar las versiones ordenadas. Este algoritmo de decisión viene definido por la función `perm-equivalent`. En la figura 6.13 se presenta la definición de `perm-equivalent` y las propiedades de corrección y completitud de dicho algoritmo de decisión, obtenidas ambas mediante instanciación funcional de los teoremas `r-equivalent-sound` y `r-equivalent--complete`. Para más detalles sobre este ejemplo, remitimos al lector al libro `perm.lisp`.

Sumario

En este capítulo:

- Hemos explicado cómo representamos en la lógica de ACL2 a las reducciones abstractas y cómo podemos enunciar diversas propiedades relativas a las mismas.
- Hemos demostrado que toda reducción normalizadora con la propiedad de Church-Rosser describe una relación de equivalencia decidible.
- Hemos demostrado que toda reducción noetheriana y localmente confluente es confluente, resultado conocido como lema de Newman.
- A partir de los dos resultados anteriores, mediante instanciación funcional, hemos demostrado que toda reducción convergente describe una relación de equivalencia decidible.

```
(defun perm-normal-form (l)
  (declare (xargs :measure l :well-founded-relation perm-rel))
  (if (natural-true-listp l)
      (let ((red (perm-reducible l)))
        (if (not red)
            l
            (perm-normal-form (perm-reduce-one-step l red))))
      l))

(defun perm-equivalent (l1 l2)
  (equal (perm-normal-form l1)
         (perm-normal-form l2)))

(defun perm-proof-irreducible (l)
  (declare (xargs :measure l :well-founded-relation perm-rel))
  (and (natural-true-listp l)
       (let ((red (perm-reducible l)))
         (if (not red)
             nil
             (cons (make-r-step
                    :elt1 l :elt2 (perm-reduce-one-step l red)
                    :operator red :direct t)
                   (perm-proof-irreducible
                      (perm-reduce-one-step l red)))))))

(defun perm-make-proof-common-normal-form (l1 l2)
   (append (perm-proof-irreducible l1)
           (inverse-proof (perm-proof-irreducible l2))))

(defthm perm-equivalent-complete
  (implies (perm-equiv-p l1 l2 p)
           (perm-equivalent l1 l2)))

(defthm perm-equivalent-sound
  (implies (and (natural-true-listp l1)
                (natural-true-listp l2)
                (perm-equivalent l1 l2))
           (perm-equiv-p
              l1 l2
              (perm-make-proof-common-normal-form l1 l2))))
```

Figura 6.13: Decidibilidad en el ejemplo de las permutaciones

Capítulo 7

Teorías ecuacionales y sistemas de reescritura

El objetivo de este capítulo es definir y formalizar en la lógica de ACL2 propiedades relativas a las teorías ecuacionales y su relación con la reescritura de términos. Dado un sistema de ecuaciones E (*axiomas*), la teoría ecuacional de E es el conjunto de ecuaciones (*teoremas*) que son consecuencia lógica de E. Alternativamente, una teoría ecuacional se puede ver como la relación de equivalencia descrita por una reducción: el reemplazamiento o *reescritura* de iguales por iguales, usando los axiomas de la teoría. Utilizaremos este punto de vista basado en reducciones, ya que nos permitirá emplear, mediante instanciación funcional, los resultados que en el capítulo 6 se han demostrado en un marco abstracto. Es interesante destacar que vamos a usar la lógica de ACL2 como *meta-lenguaje* para formalizar y razonar sobre otro sistema lógico.

En la primera sección, definimos formalmente qué significa que una ecuación sea un teorema derivable a partir de un sistema de ecuaciones. Se define así una relación binaria en el conjunto de términos de primer orden de una signatura, que puede ser vista como la relación de equivalencia descrita por una reducción definida en dicho conjunto. Informalmente, esta reducción consiste es sustituir un subtérmino que es instancia del lado izquierdo de una ecuación, por la correspondiente instancia del lado derecho. Para enfatizar este punto de vista de las ecuaciones como reglas que permiten reemplazar unos términos por otros, hablaremos de reglas de reescritura y de sistemas de reescritura de términos.

Siguiendo el esquema general mostrado en nuestra formalización de las reducciones abstractas, definimos los operadores ecuacionales, la aplicación de un paso de reducción ecuacional y un test de aplicabilidad para operadores ecuacionales. En concreto, un operador ecuacional se definirá como una estructura que contiene la posición del subtérmino que se va a sustituir, la sustitución que se va a aplicar y la ecuación que se usa para el reemplazamiento. También, como en el caso abstracto, la relación de equivalencia descrita por dicha reducción vendrá definida formalmente en ACL2 a partir del concepto de prueba (ecuacional en este caso). Para verificar que efectivamente esta relación de equivalencia formaliza el concepto pretendido de teoría ecuacional, demostraremos formalmente que se trata de la menor congruencia que contiene a los axiomas de la teoría.

En la segunda sección definimos un test de reducibilidad para la reducción asociada a un conjunto de ecuaciones. Es decir, se trata de definir y verificar una función tal que

dado un término y un sistema de ecuaciones, encuentra, si existe, una posición del término en la que ocurre un subtérmino que es instancia del lado izquierdo de una ecuación del sistema.

En la siguiente sección se discuten aspectos relativos a la terminación de la reducción de reescritura. Definiremos el concepto de orden de reducción, que nos servirá para establecer la noetherianidad de la reducción asociada a un conjunto de ecuaciones. Una vez que disponemos de un test de reducibilidad y que hemos formalizado el concepto de sistema de reescritura noetheriano, podemos también definir y verificar un algoritmo para el cálculo de formas normales.

Otro aspecto importante en toda reducción y en particular en las reducciones ecuacionales, es su confluencia. El concepto de par crítico permite decidir la confluencia local de la reducción asociada a un sistema de ecuaciones, comprobando la convergencia de un conjunto finito de pares de términos llamados pares críticos, obtenidos mediante unificación de máxima generalidad entre los lados izquierdos de las ecuaciones del sistema. Este es uno de los resultados básicos en la teoría de los sistemas de reescritura y se conoce como el teorema de pares críticos de Knuth y Bendix. En la cuarta sección, describimos la demostración formal de dicho teorema usando ACL2. La demostración automática del teorema de pares críticos de Knuth y Bendix constituye el mayor esfuerzo de prueba de los presentados en esta memoria. En la quinta sección, definimos y verificamos una función que calcula todos los pares críticos de un sistema de ecuaciones dado. Esta función, en conjunción con el teorema de pares críticos, permite expresar de manera compacta la confluencia local de un sistema de ecuaciones.

Como consecuencia de los resultados demostrados en este capítulo y en el capítulo 6 sobre reducciones abstractas, podemos finalmente demostrar la decidibilidad de las teorías ecuacionales descritas por sistemas de reescritura noetherianos cuyos pares críticos convergen. Es lo que demostramos en la sección sexta. El resultado descrito en esta sección se obtiene, usando el teorema de pares críticos, como instancia funcional de un resultado análogo para reducciones abstractas. Constituye, a nuestro juicio, un interesante colofón a toda la teoría desarrollada y descrita en esta memoria.

7.1 Teorías ecuacionales

En esta sección presentamos la formalización en la lógica ACL2 del concepto de teoría ecuacional asociada a un conjunto de ecuaciones. Es decir, definiremos qué entendemos al afirmar que una ecuación es consecuencia lógica de un conjunto de ecuaciones. Los eventos ACL2 que conducen hacia los resultados presentados en esta sección se encuentran en el libro `equational-theories.lisp`.

7.1.1 Preliminares

Las definiciones de esta subsección están tomadas, esencialmente, de [1]. Puesto que vamos a definir una lógica (la lógica ecuacional), hemos de definir el lenguaje de la misma (su sintaxis) y el significado de sus expresiones (su semántica). En este caso, la sintaxis viene definida por una signatura y los términos de primer orden en dicha signatura, ya definidos en el capítulo 3. En la lógica ecuacional, las sentencias son ecuaciones entre términos de una signatura dada. A continuación, presentamos las definiciones pertinentes para dotar de significado a los términos de primer orden y las ecuaciones.

Definición 7.1 *Una* **Σ-álgebra** $\mathbb{A}$ *consiste en un conjunto* A *(su* **dominio***) junto con una correspondencia que asocia a cada* $f \in \Sigma_n$ *una función* $f^{\mathbb{A}} : A^n \longrightarrow A$*, para todo* $n \geq 0$*. El conjunto de todas las* Σ*-álgebras lo notaremos por* $Alg(\Sigma)$*.*

Ejemplo 7.2 Si Σ_G es la signatura del ejemplo 3.2, podemos definir una Σ_G-álgebra cuyo dominio es el conjunto $\mathbb{Z}$ de los enteros y tal que al símbolo $*$ le asigna la suma de enteros, al símbolo i le asigna la operación de cálculo del opuesto de un entero y a e le asigna 0.

Definición 7.3 *Dada* $\mathbb{A} \in Alg(\Sigma)$*, una* **asignación** *en* $\mathbb{A}$ *es una función* $\alpha : X \to A$*.*

Toda asignación α en una Σ-álgebra $\mathbb{A}$, se puede extender de manera única a una función que asigna un elemento de A a cada término de $T(\Sigma, X)$, de manera similar a como se define la aplicación de una sustitución a un término (definición 3.10).

Definición 7.4 *Sea* $\mathbb{A}$ *una* Σ*-álgebra y* α *una* **asignación** *en* $\mathbb{A}$*. Definimos (por recursión estructural) el* **valor de un término** $t \in T(\Sigma, X)$ **en** $\mathbb{A}$ **respecto de la asignación** α*, notado* $\widehat{\alpha}(t)$*, de la siguiente manera:*

- $\widehat{\alpha}(t) = \alpha(x)$*, si* $t = x \in X$
- $\widehat{\alpha}(t) = f^{\mathbb{A}}(\widehat{\alpha}(t_1), \ldots, \widehat{\alpha}(t_n))$*, si* $t = f(t_1, \ldots, t_n)$

Definición 7.5 *Dada* $s \approx t$ *una ecuación en* $T(\Sigma, X)$*, decimos que es* **válida** *en* $\mathbb{A} \in Alg(\Sigma)$*, y lo notamos* $\mathbb{A} \models s \approx t$*, si para cualquier asignación en* $\mathbb{A}$*, se tiene que* $\widehat{\alpha}(s) = \widehat{\alpha}(t)$*. En ese caso, decimos que* $\mathbb{A}$ *es un* **modelo** *de* $s \approx t$*. Si* E *es un sistema de ecuaciones en* $T(\Sigma, X)$*, decimos que* $\mathbb{A}$ *es un* **modelo** *de* E*, y lo notamos* $\mathbb{A} \models E$*, si para toda ecuación* $s \approx t \in E$*, se verifica* $\mathbb{A} \models s \approx t$*.*

Definición 7.6 *Sean* E *y* $s \approx t$*, respectivamente, un sistema de ecuaciones y una ecuación en* $T(\Sigma, X)$*. Decimos que* $s \approx t$ *es* **consecuencia lógica** *de* E*, y lo notamos* $E \models s \approx t$ *si para toda* Σ*-álgebra* $\mathbb{A}$ *tal que* $\mathbb{A} \models E$*, se verifica que* $\mathbb{A} \models s \approx t$*. Alternativamente, lo notaremos por* $s =_E t$ *y llamaremos a la relación* $=_E$ *la* **teoría ecuacional** *de* E*.*

Ejemplo 7.7 Si G son los axiomas de la teoría de grupos libres definidos en el ejemplo 3.18, es posible comprobar que $x * i(x) =_G e$ y que $x * (i(x) * z) =_G z$.

El concepto de consecuencia lógica dota de una semántica a las ecuaciones. Como es habitual a la hora de tratar cualquier sistema lógico, vamos a definir un cálculo que permitirá, de manera correcta y completa, derivar teoremas ecuacionales a partir de un conjunto de ecuaciones. Este cálculo se definirá a partir de la siguiente relación de reducción:

Definición 7.8 *Dado sistema de ecuaciones* E *en* $T(\Sigma, X)$*, la* **reducción ecuacional en** $T(\Sigma, X)$ **asociada a** E*, notada por* $\to_E$*, se define de la siguiente manera:* $s \to_E t$ *si y sólo si existe* $l = r \in E$*,* $p \in \mathcal{P}(s)$ *y* σ *una sustitución en* $T(\Sigma, X)$*, tales que*

$$s/p = \sigma(l) \text{ y } t = s[p \leftarrow \sigma(r)]$$

En ese caso, decimos que p *es una ocurrencia* **redex** *de* s *(respecto a* E*) y que* s **se reduce** *a* t *o que* s **se reescribe** *a* t *(respecto de* E*).*

Puesto que $\rightarrow_E$ es una reducción definida en $T(\Sigma, X)$, le es aplicable la terminología y notaciones que se han definido en el capítulo anterior para reducciones abstractas. En particular, notaremos por $\overset{*}{\leftrightarrow}_E$ la clausura de equivalencia de la reducción $\rightarrow_E$. El siguiente teorema, (véase en [1] una demostración), establece que la relación $\overset{*}{\leftrightarrow}_E$ define un cálculo correcto y completo para la lógica ecuacional:

Teorema 7.9 (Teorema de corrección y completitud (Birkhoff)). *Sean $s, t \in T(\Sigma, X)$ y E sistema de ecuaciones en $T(\Sigma, X)$. Entonces $s =_E t$ si y sólo si $s \overset{*}{\leftrightarrow}_E t$.*

Por último, veremos una caracterización alternativa de la relación $\overset{*}{\leftrightarrow}_E$, resultado que usaremos para verificar nuestra definición ACL2 del concepto de teoría ecuacional.

Definición 7.10 *Sea $\rightarrow$ una relación en $T(\Sigma, X)$. Decimos que $\rightarrow$ es*

- **estable***: si para toda sustitución σ y para cualquier par de términos s y t tal que $s \rightarrow t$, se verifica que $\sigma(s) \rightarrow \sigma(t)$.*
- **compatible***: si para cualesquiera términos u,s,t y $p \in \mathcal{P}(u)$, se verifica que $s \rightarrow t$ implica $u[p \leftarrow s] \rightarrow u[p \leftarrow t]$.*

Definición 7.11 *Una* **relación de reescritura** *es una relación en $T(\Sigma, X)$, estable y compatible.*

Teorema 7.12 *Sea E un sistema de ecuaciones en $T(\Sigma, X)$. Entonces $\rightarrow_E$ es la menor relación de reescritura que contiene a E y $\overset{*}{\leftrightarrow}_E$ es la menor relación de equivalencia y de reescritura que contiene a E.*

Demostración:
Probemos en primer lugar que la menor relación de reescritura en $T(\Sigma, X)$ conteniendo a E es $\rightarrow_E$. Es claro que $\rightarrow_E$ contiene a E. Veamos que es estable y compatible. Supongamos que $s \rightarrow_E t$ mediante la ecuación $l \approx r \in E$, en la posición $p \in \mathcal{P}(s)$ y sea μ una sustitución tal que $s/p = \mu(l)$. Por tanto $t = s[p \leftarrow \mu(r)]$.

– Estable: Sea σ una sustitución cualquiera. Por el teorema 3.25, $p \in \mathcal{P}(\sigma(s))$ y $\sigma(s)/p = \sigma(s/p)$. Luego $\sigma(s)/p = \sigma(\mu(l))$. Además, también por 3.25,

$$\sigma(t) = \sigma(s[p \leftarrow \mu(r)]) = \sigma(s)[p \leftarrow \sigma(\mu(r))].$$

Es decir, $\sigma(s) \rightarrow_E \sigma(t)$.

– Compatible: Sea u un término cualquiera y $q \in \mathcal{P}(u)$. Según 3.25, $q \cdot p \in \mathcal{P}(u[q \leftarrow s])$, $u[q \leftarrow s]/q \cdot p = s/p = \mu(l)$ y

$$u[q \leftarrow s][q \cdot p \leftarrow \mu(r)] = u[q \leftarrow s[p \leftarrow \mu(r)]] = u[q \leftarrow t].$$

Es decir, $u[q \leftarrow s] \rightarrow_E u[q \leftarrow t]$.

Supongamos que $E \subseteq \rightarrow$ y que $\rightarrow$ es una relación de reescritura. Si $s \rightarrow_E t$, entonces $t = s[p \leftarrow \mu(r)]$, para ciertas $p \in \mathcal{P}(s)$, $l \approx r \in E$ y μ sustitución tal que $s/p = \mu(l)$. Como $l \rightarrow r$ (pues $E \subseteq \rightarrow$), por estabilidad se tiene que $\mu(l) \rightarrow \mu(r)$ y por compatibilidad

$s[p \leftarrow \mu(l)] \rightarrow s[q \leftarrow \mu(r)]$, o lo que es lo mismo, $s \rightarrow t$. Hemos probado, por tanto que $\rightarrow_E \subseteq \rightarrow$.

Por un sencillo razonamiento por inducción en la longitud de la $\leftrightarrow_E$-derivación, se demuestra que $\overset{*}{\leftrightarrow}_E$ es una relación de reescritura, ya que $\rightarrow_E$ lo es. Luego si $\sim$ es una relación de reescritura y de equivalencia en $T(\Sigma, X)$ que contiene a E, se tiene que $\rightarrow_E \subseteq \sim$ (por ser $\sim$ de reescritura) y de esto se sigue que $\overset{*}{\leftrightarrow}_E \subseteq \sim$ (pues $\sim$ es de equivalencia). □

7.1.2 Formalización de las teorías ecuacionales

En esta subsección vamos a definir un predicado ACL2 que formaliza el concepto de teorema ecuacional. Las limitaciones del lenguaje de la lógica ACL2 hace difícil la formalización de tal concepto a través de la definición de consecuencia lógica, definido en 7.6. Por ejemplo, parece difícil representar en la lógica de ACL2 a las Σ-álgebras con dominio infinito (véase [51]). Basándonos en el teorema de Birkhoff, implementaremos el concepto a través de una definición ACL2 de la relación $\overset{*}{\leftrightarrow}_E$. El teorema de Birkhoff nos asegura que $\overset{*}{\leftrightarrow}_E$ y $=_E$ son la misma relación.

La relación $\overset{*}{\leftrightarrow}_E$ es justamente la relación de equivalencia descrita por la reducción $\rightarrow_E$. Por tanto, para formalizar la teoría ecuacional definida por un conjunto finito de ecuaciones, seguiremos el marco abstracto en el que hemos definido las reducciones abstractas, pero ahora para definir una reducción concreta: la definida por un conjunto finito de axiomas ecuacionales en el conjunto de los términos de primer orden en una signatura. Una vez definida la reducción ecuacional asociada a un sistema de ecuaciones, la teoría ecuacional definida por dicho sistema será simplemente la clausura de equivalencia de dicha reducción.

Reducciones ecuacionales en ACL2

Recordamos (sección 6.2) que en la formalización de una reducción intervienen varios elementos:

- los objetos del dominio de definición,
- los operadores que se aplican a tales objetos,
- la función que define el dominio de definición,
- la función que implementa la aplicación de un paso de reducción y
- la función que implementa el test de aplicabilidad.

Por tanto, para formalizar en ACL2 la reducción $\rightarrow_E$ definida por un conjunto de ecuaciones E, describimos a continuación cada uno de los elementos anteriores para el caso de las reducciones ecuacionales.

Es evidente que el dominio de definición de una reducción ecuacional es el conjunto de los términos en una signatura dada. En el capítulo 3 ya se discutió ampliamente la representación escogida en ACL2 para los términos de primer orden. Por tanto, la función que implementa el dominio de definición es `term-s-p` (véase la subsección 3.1.2).

Discutimos a continuación la representación de los operadores correspondientes a una reducción ecuacional, que llamaremos **operadores ecuacionales**. Un operador debe contener toda la información necesaria para poder llevar a cabo la transformación que define. En este caso, y analizando la definición de $\rightarrow_E$, un paso de reducción ecuacional aplicado a un término viene caracterizado por la posición del término donde ocurre el reemplazamiento, junto con la ecuación y la sustitución empleada. Basándonos en esta idea, representamos un operador ecuacional mediante una estructura[1] `eq-operator` con tres campos: `pos` (para almacenar la posición del término donde ocurre el reemplazamiento), `rule` (conteniendo la ecuación o regla usada) y `match` (con la sustitución empleada).

```
(defstructure eq-operator pos rule match
              (:options (:conc-name op-)))
```

La opción `:conc-name` permite especificar un prefijo para las funciones de acceso a la estructura. En este caso, las funciones de acceso a los campos de una estructura `eq-operator` serán `op-pos`, `op-rule` y `op-match`. Además, la función reconocedora de estructuras de este tipo se denomina `eq-operator-p` y la función constructora `make-eq--operator`. Todas estas funciones se definen automáticamente mediante esta llamada a `defstructure`.

Una vez establecida la representación de los objetos sobre los que va a estar definida una reducción ecuacional y de los operadores que se aplican a esos objetos, definamos el test de aplicabilidad ecuacional. En consonancia con el marco abstracto definido en el capítulo 6, un test de aplicabilidad debe ser una función tal que recibiendo un elemento del dominio de definición y un operador, comprueba si el operador puede ser aplicado al elemento.

En este caso, debemos definir una función que compruebe si un operador ecuacional puede ser aplicado a un término. La función `eq-legal` formaliza el test de aplicabilidad para las reducciones ecuacionales:

```
(defun eq-legal (term op E)
  (let ((pos (op-pos op))
        (rule (op-rule op))
        (sigma (op-match op)))
    (and (eq-operator-p op)
         (member rule E)
         (substitution-s-p sigma)
         (position-p pos term)
         (equal (instance (lhs rule) sigma)
                (occurrence term pos)))))
```

Es decir, un operador `op` es aplicable a un término `term` respecto de un conjunto de ecuaciones `E` si es una estructura `eq-operator`, cuya campo `rule` contiene una ecuación de `E`, el campo `match` contiene una sustitución en la signatura, el campo `pos` contiene una posición de `term` y el subtérmino en dicha posición es instancia (dada por la sustitución `sigma`) del lado izquierdo de la ecuación `rule`.

Finalmente, definimos la aplicación de un operador ecuacional (que ha de ser aplicable) a un término. La función `eq-reduce-one-step` formaliza este concepto:

[1] Usando nuevamente la macro `defstructure` [8].

```
(defun eq-reduce-one-step (term op)
  (replace-term term
                (op-pos op)
                (instance (rhs (op-rule op))
                          (op-match op))))
```

Como se observa, el subtérmino que ocurre en la posición indicada por el operador es sustituido por la instancia (dada por la sustitución del operador) del lado derecho de la regla del operador.

Pruebas ecuacionales: teoría ecuacional

Una vez definida la reducción $\rightarrow_E$, para definir la teoría ecuacional definida por E, hemos de definir su clausura de equivalencia $\overset{*}{\leftrightarrow}_E$. Como en el caso de las reducciones abstractas, y debido a la naturaleza constructiva de la lógica de ACL2, al definir la relación $s \overset{*}{\leftrightarrow}_E t$, es necesario incluir un argumento con la secuencia de pasos $s = u_0 \leftrightarrow_E u_1 \leftrightarrow_E u_2 \ldots \leftrightarrow_E u_n = t$. Tal secuencia de pasos la llamaremos una **prueba ecuacional** y supone el análogo ecuacional de las pruebas abstractas definidas en 6.2.2.

Debemos definir, por tanto, el concepto de paso de prueba ecuacional. Recuérdese que en la sección 6.2.2 definimos los pasos de prueba mediante una estructura `r-step`, con cuatro campos: `direct` (un campo booleano que indica el sentido en el que se da el paso), `operator` (el operador que se aplica) y `elt1`, `elt2` (los elementos que se conectan). No es necesario definir una nueva estructura para los pasos de pruebas ecuacionales; simplemente en el caso ecuacional los campos `elt1` y `elt2` deben contener términos en una signatura dada y el campo `operator` debe contener operadores ecuacionales.

Sí que debemos definir, en cambio, el concepto de paso de prueba ecuacional **legal**. Como en el caso de las reducciones abstractas, un paso de prueba ecuacional se dirá legal si uno de sus términos se obtiene aplicando el operador al otro término en el sentido indicado y dicho operador ecuacional es aplicable al término en el que se lleva a cabo la reducción. Esto es precisamente lo que define la función `eq-proof-step-p`, que implementa el análogo ecuacional de la función `proof-step-p` de la sección 6.2.2:

```
(defun eq-proof-step-p (s E)
   (let ((t1 (elt1 s)) (t2 (elt2 s))
         (operator (operator s)) (direct (direct s)))
     (and (r-step-p s)
          (implies direct
                   (and (eq-legal t1 operator E)
                        (equal (eq-reduce-one-step t1 operator) t2)))
          (implies (not direct)
                   (and (eq-legal t2 operator E)
                        (equal (eq-reduce-one-step t2 operator) t1))))))
```

Finalmente, podemos definir la relación $\overset{*}{\leftrightarrow}_E$ a través del concepto de prueba ecuacional. Una prueba ecuacional es una secuencia concatenada de pasos de prueba ecuacionales legales, que conecta dos términos. Así, la función `eq-equiv-s-p` es la contrapartida ecuacional de la función `equiv-p` definida en la sección 6.2.2 y define la clausura de equivalencia de la reducción ecuacional correspondiente a un sistema de ecuaciones `E`:

```
(defun eq-equiv-s-p (t1 t2 p E)
  (if (endp p)
      (and (equal t1 t2) (term-s-p t1))
    (and (term-s-p t1)
         (eq-proof-step-p (car p) E)
         (equal t1 (elt1 (car p)))
         (eq-equiv-s-p (elt2 (car p)) t2 (cdr p) E))))
```

Fijado un sistema de ecuaciones `E` en una signatura, la función `eq-equiv-s-p` implementa la definición en ACL2 de la teoría ecuacional correspondiente a `E`. En concreto, dados dos términos `t1` y `t2`, diremos que `t1` y `t2` son equivalentes en la teoría ecuacional que define `E` si existe una prueba ecuacional `p` tal que se verifica `(eq-equiv-s-p t1 t2 p E)`.

Ejemplo 7.13 Supongamos dados los términos $t_1 = k(h(v))$, $t_2 = k(f(v))$, $t_3 = v$ (en una signatura adecuada) y el sistema de ecuaciones $E = \{f(x) \approx h(x), k(f(x)) \approx x\}$. Podemos definir en ACL2 las siguientes constantes para almacenar la representación de tales elementos:

```
(defconst *t1* '(k (h v)))
(defconst *t2* '(k (f v)))
(defconst *t3* 'v)
(defconst *E* '(((f x) . (h x) ) ((k (f x)) . x)))
```

Es posible probar que $t_1 \overset{*}{\leftrightarrow}_E t_3$, ya que $t_1 \leftarrow_E t_2 \rightarrow_E t_3$. Nótese que t_2 se reduce a t_1 sustituyendo el subtérmino $f(v)$ por $h(v)$, aplicando la primera ecuación de E. También t_2 se reduce a t_3 aplicando la segunda ecuación para sustituir $k(f(v))$ por v. Esta justificación de la equivalencia entre t_1 y t_3 se puede representar en ACL2 mediante la siguiente prueba `*p*` que consta de los pasos `*s1*` y `*s2*`. Es decir, el valor de `(eq-equiv-s-p *t1* *t3* *p* *E*)`, es `t`[2].

```
(defconst *s1* (make-r-step :direct nil
                            :operator (make-eq-operator
                                       :pos '(1)
                                       :rule '((f x) . (h x))
                                       :match '((x . v)))
                            :elt1 '(k (h v))
                            :elt2 '(k (f v))))

(defconst *s2* (make-r-step :direct t
                            :operator (make-eq-operator
                                       :pos nil
                                       :rule '((k (f x)) . x)
                                       :match '((x . v)))
                            :elt1 '(k (f v))
                            :elt2 'v))

(defconst *p* (list *s1* *s2*))
```

[2]Previa definición de una signatura adecuada, véase sección `2.2` del libro `equational-theories.lisp`.

En lo que sigue, verificaremos que efectivamente la función `eq-equiv-s-p` formaliza el concepto de teoría ecuacional definida por un conjunto de ecuaciones.

7.1.3 Álgebra de pruebas ecuacionales

Antes de verificar las propiedades principales de la función `eq-equiv-s-p` definiremos una serie de funciones que transforman pruebas ecuacionales en otras pruebas ecuacionales. Estas funciones pueden ser vistas como un álgebra definida sobre el conjunto de las pruebas ecuacionales y juegan un papel fundamental en la formalización de las propiedades que se verificarán a continuación.

Las funciones `append` e `inverse-proof` (definida en la sección 6.2.2) definen, respectivamente, la concatenación y la inversión de pruebas. No es necesario redefinir estas operaciones específicamente para pruebas ecuacionales.

Dada una sustitución `sigma` y un paso de prueba ecuacional `s`, podemos definir la instancia de `s` mediante `sigma`, tal y como lo hace la función `eq-step-instance`:

```
(defun eq-step-instance (s sigma)
  (make-r-step
   :elt1 (instance (elt1 s) sigma)
   :elt2 (instance (elt2 s) sigma)
   :direct (direct s)
   :operator (make-eq-operator
              :pos (op-pos (operator s))
              :rule (op-rule (operator s))
              :match (composition sigma (op-match (operator s))))))
```

La instanciación de una prueba `p` mediante una sustitución `sigma` se obtiene instanciando cada uno de sus pasos de prueba, como hace la función `eq-proof-instance`:

```
(defun eq-proof-instance (p sigma)
  (if (endp p)
      p
    (cons (eq-step-instance (car p) sigma)
          (eq-proof-instance (cdr p) sigma))))
```

Dado un término `term`, una posición `pos` de `term` y un paso de prueba ecuacional `s`, podemos definir el paso de prueba resultante de incluir el paso de prueba `s` en el contexto definido por `term` y `pos`. Es lo que hace la función `eq-step-context`:

```
(defun eq-step-context (s term pos)
  (make-r-step
   :elt1 (replace-term term pos (elt1 s))
   :elt2 (replace-term term pos (elt2 s))
   :direct (direct s)
   :operator (make-eq-operator
              :pos (append pos (op-pos (operator s)))
              :rule (op-rule (operator s))
              :match (op-match (operator s)))))
```

La inclusión de una prueba `p` en el contexto definido por `term` y `pos` se obtiene incluyendo cada paso en dicho contexto, como hace la función `eq-proof-context`:

```
(defun eq-proof-context (p term pos)
  (if (endp p)
      p
    (cons (eq-step-context (car p) term pos)
          (eq-proof-context (cdr p) term pos))))
```

Para acabar con el conjunto de operaciones que construyen pruebas ecuacionales, dado un axioma `equ` podemos construir la prueba ecuacional que justifica la equivalencia del lado izquierdo del axioma con su lado derecho, como hace la siguiente función `eq-proof--axiom`:

```
(defun eq-proof-axiom (equ)
  (list
   (make-r-step
    :elt1 (lhs equ) :elt2 (rhs equ) :direct t
    :operator (make-eq-operator :pos nil :rule equ :match nil))))
```

7.1.4 Propiedades principales

Para verificar que efectivamente la función `eq-equiv-s-p` implementa la relación $\overset{*}{\leftrightarrow}_E$, demostraremos formalmente el teorema 7.12. Por tanto, hemos de probar que, fijado un sistema de ecuaciones `E`, `eq-equiv-s-p`[3] define la menor relación de equivalencia y de reescritura que contiene a `E`. Veamos cómo expresamos este resultado en la lógica de ACL2.

Los siguientes teoremas demuestran que `eq-equiv-s-p` es una relación de equivalencia (propiedades reflexiva, simétrica y transitiva):

```
(defthm eq-equiv-s-p-reflexive
  (implies (term-s-p term)
           (eq-equiv-s-p term term nil E)))

(defthm eq-equiv-s-p-symmetric
  (implies (eq-equiv-s-p t1 t2 p E)
           (eq-equiv-s-p t2 t1 (inverse-proof p) E)))

(defthm eq-equiv-s-p-transitive
  (implies (and (eq-equiv-s-p t1 t2 p E)
                (eq-equiv-s-p t2 t3 q E))
           (eq-equiv-s-p t1 t3 (append p q) E)))
```

Cada vez que se quiere establecer la equivalencia de dos términos se ha de *construir* la prueba ecuacional que justifica tal equivalencia. Por ejemplo, la propiedad de simetría consiste en probar la equivalencia entre dos términos `t2` y `t1`, suponiendo que existe una prueba `p` que justifica la equivalencia entre `t1` y `t2`. Entonces la equivalencia entre `t2`

[3] Rigurosamente, deberíamos decir "la relación *existe* `p` tal que `(equiv-s-p t1 t2 p E)`"

y `t1` queda justificada por la prueba `(inverse-proof p)`. Del mismo modo la prueba vacía `nil` sirve para demostrar la propiedad reflexiva y la concatenación de pruebas para demostrar la propiedad transitiva.

Los siguientes teoremas establecen, respectivamente, la estabilidad y compatibilidad de la relación definida por `eq-equiv-s-p`:

```
(defthm eq-equiv-s-p-stable
  (implies (and (eq-equiv-s-p t1 t2 p E)
                (substitution-s-p sigma))
           (eq-equiv-s-p (instance t1 sigma)
                         (instance t2 sigma)
                         (eq-proof-instance p sigma) E)))

(defthm eq-equiv-s-p-compatible
    (implies (and (eq-equiv-s-p t1 t2 p E)
                  (term-s-p term)
                  (position-p pos term))
             (eq-equiv-s-p (replace-term term pos t1)
                           (replace-term term pos t2)
                           (eq-proof-context p term pos) E)))
```

En este caso, las funciones `eq-proof-instance` y `eq-proof-context` sirven para construir las pruebas que justifican las equivalencias necesarias para demostrar las propiedades de estabilidad y compatibilidad, respectivamente.

Por último el siguiente teorema muestra que `eq-equiv-s-p` contiene a `E`. En este caso la función `eq-proof-axiom` proporciona la prueba ecuacional correspondiente:

```
(defthm eq-equiv-s-p-contains-E
  (implies (and (system-s-p E) (member equ E))
           (eq-equiv-s-p (lhs equ) (rhs equ)
                         (eq-proof-axiom equ) E)))
```

Los teoremas anteriores establecen que, fijado un sistema de ecuaciones `E`, el predicado `eq-equiv-s-p` define una relación de equivalencia y de reescritura que contiene a las ecuaciones de `E`. En este punto merece la pena destacar la ventaja que supone el hecho de representar las pruebas ecuacionales como objetos ACL2 que pueden ser manipulados para obtener nuevas pruebas. En concreto, cada una de las propiedades demostradas se corresponde con una de las funciones definidas en el álgebra de pruebas presentada en la subsección anterior[4].

Para demostrar formalmente el teorema 7.12, resta por probar que `eq-equiv-s-p` es la menor relación definida en el conjunto de términos de la signatura, que tiene las propiedades anteriores. Para formalizar este resultado en ACL2, usaremos `encapsulate` para definir una relación `rel` y un sistema de ecuaciones `(E)`, asumiendo únicamente que `rel` es una relación de equivalencia y de reescritura que contiene a `(E)`:

[4]Incluso podemos ver la prueba `nil` como una constante en el álgebra de pruebas.

```
(encapsulate
 ((rel (t1 t2) boolean)
  (E () axioms))
 ...
 (defthm E-is-system-s-p
   (system-s-p (E)))

 (defthm rel-contains-axioms
   (implies (member e (E))
            (rel (lhs e) (rhs e))))

 (defthm rel-reflexive
   (implies (term-s-p term) (rel term term)))

 (defthm rel-symmetric
   (implies (and (term-s-p t1) (term-s-p t2) (rel t1 t2))
            (rel t2 t1)))

 (defthm rel-transitive
   (implies (and (term-s-p t1) (term-s-p t2) (term-s-p t3)
                 (rel t1 t2) (rel t2 t3))
            (rel t1 t3)))

 (defthm rel-stable
   (implies (and (term-s-p t1) (term-s-p t2)
                 (substitution-s-p sigma)
                 (rel t1 t2))
            (rel (instance t1 sigma)
                 (instance t2 sigma))))

 (defthm rel-compatible
   (implies (and (term-s-p t1) (term-s-p t2) (term-s-p term)
                 (rel t1 t2) (position-p pos term))
            (rel (replace-term term p t1)
                 (replace-term term p t2)))))
```

El siguiente teorema establece que la relación `rel` contiene a la relación que describe `eq-equiv-s-p` y por tanto ésta es la menor relación que verifica las propiedades asumidas para `rel`.

```
(defthm eq-equiv-s-p-the-least-congruence-containing-E
  (implies (eq-equiv-s-p t1 t2 p (E))
           (rel t1 t2)))
```

El conjunto de los teoremas presentados en esta subsección sirven para demostrar formalmente en ACL2 el teorema 7.12. Estos teoremas, junto con el (meta) teorema de Birkhoff hacen que aumente nuestra confianza en que la función `eq-equiv-s-p` implementa formalmente en ACL2 el concepto pretendido: la teoría ecuacional definida por un sistema

de ecuaciones. En el apéndice C, subsección C.5.1, hacemos algunos comentarios sobre la prueba automática de los mismos.

7.2 Reducibilidad

Ya vimos en el tema dedicado a las reducciones abstractas, que un componente fundamental a la hora de definir el cálculo de formas normales era lo que denominamos test de reducibilidad. Aplicado al caso ecuacional, se trata de definir una función que recibiendo un término y un sistema de ecuaciones, encuentre, si existe, un operador aplicable al término respecto de la reducción descrita por el sistema de ecuaciones. En esta sección definimos y verificamos tal función. Los eventos que conducen a los resultados aquí presentados se encuentran en el libro `rewriting.lisp`.

7.2.1 Definición y propiedades principales

A continuación, definimos la función `eq-reducible` que implementa un test de reducibilidad para las reducciones ecuacionales. Como primera función auxiliar, necesitaremos la función `eq-reducible-top`, que comprueba, dado un término y un conjunto de ecuaciones, si el término es instancia del lado izquierdo de una de las ecuaciones del conjunto. En tal caso, devuelve un multivalor consistente en la ecuación encontrada, la sustitución correspondiente y el valor booleano `t`. En caso contrario, se devuelve `(mv nil nil nil)`. Nótese el uso del algoritmo de equiparación en esta definición:

```
(defun eq-reducible-top (term E)
  (if (endp E)
      (mv nil nil nil)
    (let ((equ (car E)))
      (mv-let (matching subsumption)
              (subs-mv (lhs equ) term)
              (if subsumption
                  (mv equ matching t)
                (eq-reducible-top term (cdr E)))))))
```

A continuación definimos la función `eq-reducible-aux`, definida tanto para términos como para listas de términos, que recorre la estructura de un término en busca de una posición del mismo en la que ocurra una instancia del lado izquierdo de una ecuación. En tal caso, devuelve el operador ecuacional correspondiente; en caso contrario, devuelve `nil`. Además de los argumentos de entrada `flg` (indicador de término o lista de términos), `term` (el término) y `E` (el sistema de ecuaciones), la función tiene dos argumentos auxiliares `posr` y `arg` que, como explicaremos más adelante, van a servir para saber en cada momento la posición del subtérmino que se está analizando.

```
(defun eq-reducible-aux (flg term posr arg E)
  (if flg
      (mv-let (equ match top-reducible)
              (eq-reducible-top term E)
              (if top-reducible
                  (make-eq-operator :pos (revlist posr)
```

```
                                    :rule equ
                                    :match match)
                (if (variable-p term)
                    nil
                  (eq-reducible-aux nil (cdr term)  posr 0 E))))
    (if (endp term)
        nil
      (let ((first-arg-red
             (eq-reducible-aux t (car term) (cons (1+ arg) posr) 0 E)))
        (if first-arg-red
            first-arg-red
          (eq-reducible-aux nil (cdr term) posr (1+ arg) E))))))
```

Finalmente, la función `eq-reducible` consiste en la siguiente llamada inicial a `eq--reducible-aux`:

```
(defun eq-reducible (term E)
  (eq-reducible-aux t term nil 0 E))
```

Una llamada a `(eq-reducible t1 E)`, realiza una búsqueda (en profundidad) sobre la estructura del término `t1` para encontrar, si existe, un operador ecuacional legal respecto a `t1`. La función `eq-reducible-aux` implementa esta búsqueda recursiva. En este caso, se produciría inicialmente una llamada a `(eq-reducible-aux t t1 nil 0 E)`. Durante el proceso recursivo, una llamada a `(eq-reducible-aux flg term posr arg E)` debe ser interpretada de la siguiente manera:

- El argumento `posr` contiene la posición correspondiente al subtérmino del término `t1` que en ese momento se está analizando.
- Si `flg` es distinto de `nil`, `term` es el subtérmino que ocurre en `t1` en la posición indicada por `posr` (en orden inverso). En ese momento se está analizando `term` para comprobar si alguno de sus subtérminos es instancia del lado izquierdo de alguna ecuación de `E`.
- Si `flg` es `nil`, `term` es una lista de términos, lista parcial de los argumentos del subtérmino que ocurre en `t1` en la posición indicada por `posr` (en orden inverso). Intuitivamente, `arg` es el número de argumentos previos ya analizados y `term` la lista de argumentos por analizar.
- En el caso de que se recorra el término completamente sin encontrar un subtérmino que sea instancia del lado izquierdo de una ecuación, se devuelve `nil`. Si se encuentra tal subtérmino, se devuelve el operador correspondiente. En ese caso, la sustitución y ecuación del operador ecuacional se obtiene mediante la función previamente definida `eq-reducible-top`. El uso de `posr` y `arg` como argumentos adicionales de `eq--reducible-aux` permite obtener la posición del operador. La función `revlist`, cuya definición omitimos, calcula la inversa de una lista dada.

Los siguientes teoremas establecen las dos propiedades fundamentales de la función `eq-reducible` y verifican formalmente que tal función implementa un test de reducibilidad para la reducción ecuacional:

```
(defthm eq-reducible-implies-eq-legal
  (implies (and (system-s-p E)
                (term-s-p term)
                (eq-reducible term E))
           (eq-legal term (eq-reducible term E) E)))

(defthm not-eq-reducible-nothing-eq-legal
  (implies (not (eq-reducible term E))
           (not (eq-legal term op E))))
```

El teorema `eq-reducible-implies-eq-legal` afirma que si `eq-reducible` no devuelve `nil`, entonces devuelve un operador ecuacional aplicable al término que recibe como entrada. El teorema `not-eq-reducible-nothing-eq-legal` establece que en caso contrario, no existe ningún operador ecuacional aplicable al término. Estos dos teoremas son los análogos ecuacionales a los teoremas `reducible-implies-legal` y `not-reducible--nothing-legal` presentados en la figura 6.11 del capítulo 6, asumidos entonces como propiedades de un test de reducibilidad abstracto.

7.2.2 Descripción de la demostración

A continuación describimos los principales lemas que permiten obtener una prueba en ACL2 de las propiedades de `eq-reducible` presentadas anteriormente.

Revisión de la estructura de árbol de un término

Antes de pasar a describir la demostración de las propiedades anteriores, veamos unas definiciones alternativas de las funciones `position-p`, `occurrence` y `replace-term` que serán de ayuda en la formulación de los lemas previos necesarios para demostrar las propiedades del test de reducibilidad. Las definiciones y teoremas de este apartado se encuentran en el libro `terms.lisp` que acompaña a esta memoria. Como se vió en la sección 3.2, tales funciones estaban definidas usando la función `nth`, formalizando así los conceptos de posición, ocurrencia y reemplazamiento de términos tal y como se definen usualmente en la literatura relacionada.

Estas funciones estaban definidas *sólo para términos*. Necesitaremos ahora, como veremos, formular determinadas propiedades (en las que intervienen los conceptos de posición, ocurrencia y reemplazamiento) tanto para términos como para listas de términos. Extenderemos, pues, tales conceptos a las listas de términos, usando para ello nuestro estilo estándar de definir funciones en la estructura recursiva de los términos. La siguiente función `position-p-rec` implementa la versión recursiva en la estructura de los términos de la función `position-p`:

```
(defun position-p-rec (flg pos term)
  (if flg
      (cond ((atom pos) (equal pos nil))
            ((variable-p term) nil)
            (t (position-p-rec nil pos (cdr term))))
    (cond ((or (atom term) (atom pos)) nil)
          ((equal (car pos) 1)
```

```
          (position-p-rec t (cdr pos) (car term)))
         ((and (integerp (car pos)) (<= 2 (car pos)))
          (position-p-rec nil
                          (cons (- (car pos ) 1) (cdr pos))
                          (cdr term)))
         (t nil))))
```

Según esta definición, el concepto de posición de un término es el mismo que el implementado por `position-p`, como demostraremos más adelante. Para listas de términos, el primer elemento de la posición indica el término del cual es posición el resto. Es decir, si $l = (t_1, \ldots, t_n)$ es una lista de términos, la posición $i \cdot p$ es posición de l si p es posición del término t_i. Siguiendo esta idea intuitiva, podemos definir también `occurrence-rec` y `replace-term-rec`, que implementan la versión recursiva en la estructura de los términos de las funciones `occurrence` y `replace-term` respectivamente:

```
(defun occurrence-rec (flg term pos)
  (if flg
      (cond ((endp pos) term)
            ((variable-p term) nil)
            (t (occurrence-rec nil (cdr term) pos)))
    (cond ((or (endp term) (endp pos)) nil)
          ((equal (car pos) 1)
           (occurrence-rec t (car term) (cdr pos)))
          ((and (integerp (car pos)) (<= 2 (car pos)))
           (occurrence-rec nil (cdr term)
                           (cons (- (car pos ) 1) (cdr pos))))
          (t nil))))

(defun replace-term-rec (flg term1 pos term2)
  (if flg
      (cond ((endp pos) term2)
            ((variable-p term1) nil)
            (t (cons (car term1)
                     (replace-term-rec
                        nil (cdr term1) pos term2))))
    (cond ((or (endp term1) (endp pos)) nil)
          ((equal (car pos) 1)
           (cons (replace-term-rec
                    t (car term1) (cdr pos) term2)
                 (cdr term1)))
          ((and (integerp (car pos)) (<= 2 (car pos)))
           (cons (car term1)
                 (replace-term-rec
                    nil (cdr term1)
                    (cons (- (car pos ) 1) (cdr pos)) term2)))
          (t nil))))
```

Los siguientes teoremas establecen la equivalencia entre las versiones originales de las

funciones y las versiones recursivas en la estructura que acabamos de presentar:

```
(defthm equal-position-p-position-p-rec
   (equal (position-p pos term)
          (position-p-rec t pos term)))

(defthm equal-occurrence-occurrence-rec
  (implies (position-p-rec t pos term)
           (equal (occurrence term pos)
                  (occurrence-rec t term pos))))

(defthm equal-replace-term-replace-term-rec
  (implies (position-p-rec t pos term1)
           (equal (replace-term term1 pos term2)
                  (replace-term-rec t term1 pos term2))))
```

Además de extender los conceptos relativos a la estructura de árbol de los términos, también a las listas de términos, el hecho de que estas versiones tengan un esquema recursivo en la estructura de los términos, hace que la automatización de las demostraciones por inducción se facilite, como explicamos más adelante.

Propiedades de `eq-reducible-top`

Recordemos que la función `eq-reducible-top` comprueba si un término dado es instancia del lado izquierdo de una de las ecuaciones de un conjunto dado. Los siguientes teoremas establecen sus propiedades principales:

```
(defthm eq-reducible-top-subs-lhs
  (let ((eq-reducible-top (eq-reducible-top term E)))
    (implies (third eq-reducible-top)
             (and (member (first eq-reducible-top)  E)
                  (equal (instance (lhs (first eq-reducible-top))
                                   (second eq-reducible-top))
                         term)))))

(defthm not-eq-reducible-top-member
  (implies (and (not (third (eq-reducible-top term E)))
                (member equ E))
           (not (subs (lhs equ) term))))
```

Es decir, si `(eq-reducible-top term E)` no es `nil`, entonces devuelve una ecuación de `E` y una sustitución que equipara su lado izquierdo con `term`. En caso contrario, no existe ninguna ecuación en `E` cuyo lado izquierdo subsuma a `term`.

Demostración del teorema `eq-reducible-implies-eq-legal`

El siguiente es el lema principal que necesitamos para demostrar el teorema `eq-reducible-implies-eq-legal` presentado anteriormente:

```
(defthm eq-reducible-aux-main-lemma
  (let* ((red (eq-reducible-aux flg term posr arg E))
         (pos (op-pos red))
         (rule (op-rule red))
         (matching (op-match red))
         (q (difference-pos (revlist posr) pos))
         (ql (cons (- (car q) arg) (cdr q))))
    (implies (and red (integerp arg) (>= arg 0))
             (if flg
                 (and (position-p-rec flg q term)
                      (equal (instance (lhs rule) matching)
                             (occurrence-rec flg term q)))
                (and (position-p-rec flg ql term)
                     (equal (instance (lhs rule) matching)
                            (occurrence-rec flg term ql)))))))
```

Este lema viene a especificar las propiedades fundamentales del proceso recursivo que define la función `eq-reducible-aux`, en el caso en el que no devuelve fallo. En cierta manera, viene a establecer los invariantes en dicho proceso. Expliquemos el lema con más detalle.

Supongamos que `(eq-reducible-aux flg term posr arg E)` no es `nil` y por tanto es un operador ecuacional. En el lema, las variables locales `pos`, `rule` y `matching` representan respectivamente la posición, ecuación y sustitución de tal operador. Además, `q` representa la diferencia entre las posiciones `posr` (en orden inverso) y `pos`. La función `difference-pos` (cuya definición omitimos) implementa la diferencia de posiciones formalizando la definición 3.22.

El lema divide su enunciado en dos casos, dependiendo del valor de `flg`. Si `flg` es distinto de `nil`, entonces `q` es una posición del término `term`. En el caso en el que `flg` sea `nil`, la posición obtenida restando `arg` al primer elemento de `q` y dejando el resto sin cambiar (que hemos llamado `ql`), es una posición de la lista de términos `term`. En ambos casos, en tal posición ocurre un subtérmino que la sustitución `matching` equipara al lado izquierdo de `rule`.

Como se observa, este lema viene a formalizar la descripción informal que se hizo del comportamiento de `eq-reducible-aux` en la subsección anterior. Nótese la utilidad de disponer de las versiones alternativas `position-p-rec` y `occurrence-rec`, que nos permiten formular la propiedad sobre `eq-reducible-aux` en lugar de para `eq-reducible`. Es decir, conjuntamente para términos y para listas de términos.

La demostración de esta propiedad se hace por inducción en la estructura de los términos, siguiendo el esquema sugerido por `eq-reducible-aux`. El lector interesado puede consultar los lemas previos que se necesitan para completar la demostración en la sección `1.2.1` del libro `rewriting.lisp`. El hecho de expresar las posiciones y ocurrencias de un término de manera recursiva en la estructura, similar a la de `eq-reducible-aux`, facilita la demostración.

El teorema `eq-reducible-implies-eq-legal` se tiene ahora simplemente por definición de `eq-legal`, sin más que instanciar el lema anterior con `flg`, `posr` y `arg` igual a `t`, `nil` y `0`, respectivamente. Nótese además que la definición de `eq-legal` usa las funciones `position-p` y `occurrence` en lugar de la versiones alternativas. Por tanto se necesitan

también los teoremas de equivalencia entre ambas versiones, mostrados anteriormente.

Demostración del teorema `not-eq-reducible-nothing-eq-legal`

Para demostrar el teorema `not-eq-reducible-nothing-eq-legal`, presentado anteriormente, necesitamos el siguiente lema sobre `eq-reducible-aux`:

```
(defthm not-eq-reducible-aux-not-subs-positions
  (implies (and (not (eq-reducible-aux flg term posr arg E))
                (position-p-rec flg q term)
                (member rule E))
           (not (equal (instance (lhs rule) sigma)
                       (occurrence-rec flg term q)))))
```

Este lema establece que cuando `(eq-reducible-aux flg term posr arg E)` es `nil`, no existe ningún subtérmino de `term` que sea instancia del lado izquierdo de una regla de `E`. Este teorema se demuestra por inducción en la estructura de los términos. Para su demostración necesitamos el teorema `not-eq-reducible-top-member`, presentado anteriormente, que establecía una propiedad análoga para `eq-reducible-top`. Tal propiedad está enunciada en términos de la relación de subsunción `subs`, por lo que necesitamos usar `subs-completeness` para expresar el resultado en términos de instancias de términos.

Nuevamente por la definición de `eq-legal`, junto con los teoremas de equivalencia para `position-p-rec` y `occurrence-rec`, podemos usar el lema anterior para el caso en el que `flg`, `posr` y `arg` tienen valor `t`, `nil` y `0`, respectivamente, obteniendo el teorema `not-eq-reducible-nothing-eq-legal`.

En el apéndice C, subsección C.5.2, damos alguna información adicional sobre la demostración de los teoremas presentados en esta sección.

7.3 Sistemas de reescritura: terminación y formas normales

En esta sección, presentamos la formalización en ACL2 del concepto de sistema de reescritura de términos. A continuación, definimos la terminación (o noetherianidad) de un sistema de reescritura a partir del concepto de orden de reducción. Finalmente, definimos y verificamos una función para el cálculo de formas normales de términos respecto de sistemas de reescritura noetherianos. Los eventos que conducen a los resultados presentados en esta sección se encuentran en en el libro `rewriting.lisp`.

7.3.1 Preliminares

Definición 7.14 *Una* **regla de reescritura** *en $T(\Sigma, X)$ es un par ordenado (l, r) tal que $l, r \in T(\Sigma, X)$, notada $l \to r$, verificando:*

- $\mathcal{V}(r) \subseteq \mathcal{V}(l)$.
- $l \notin X$.

El término l es el **lado izquierdo** *de la regla y r es el* **lado derecho**. *Un* **sistema de reescritura de términos** *en $T(\Sigma, X)$ (abreviadamente* **SRT***) es un conjunto de reglas de reescritura en $T(\Sigma, X)$.*

Como se observa, una regla de reescritura es simplemente una ecuación con algunas restricciones sintácticas. Por tanto, los SRTs son también sistemas de ecuaciones. Dado un sistema de reescritura $\mathcal{R}$, tiene sentido hablar de la teoría ecuacional de $\mathcal{R}$, notada $=_{\mathcal{R}}$ y de la relación $\rightarrow_{\mathcal{R}}$ como la reducción ecuacional asociada al conjunto de ecuaciones $\mathcal{R}$ (que en ese caso llamaremos también **reducción de reescritura**). En consecuencia, toda la terminología sobre reducciones abstractas se puede emplear en las reducciones de reescritura. En particular, diremos que un término está en **forma normal** (o que es **irreducible**) respecto de un SRT $\mathcal{R}$, si lo está respecto de la reducción $\rightarrow_{\mathcal{R}}$. Al decir que un SRT es **noetheriano** o que **termina** nos referimos nuevamente a la reducción $\rightarrow_{\mathcal{R}}$.

Las restricciones sintácticas impuestas a las reglas de reescritura permiten eliminar algunos casos obvios en los que la reducción $\rightarrow_{\mathcal{R}}$ no termina. En cualquier caso, la mayor parte de la teoría desarrollada sobre sistemas de reescritura se cumple también para ecuaciones arbitrarias.

Ejemplo 7.15 El siguiente conjunto de reglas de reescritura $\mathcal{R}_G$ es un sistema de reescritura de términos en $T(\Sigma_G, X)$, donde Σ_G es la signatura de la teoría de grupos libres, definida en el ejemplo 3.2:

$(x * y) * z$	$\rightarrow$	$x * (y * z)$	$x * e$	$\rightarrow$	x
$e * x$	$\rightarrow$	x	$x * i(x)$	$\rightarrow$	e
$i(x) * x$	$\rightarrow$	e	$i(i(x))$	$\rightarrow$	x
$i(e)$	$\rightarrow$	e	$i(x * y)$	$\rightarrow$	$i(y) * i(x)$
$i(x) * (x * y)$	$\rightarrow$	y	$x * (i(x) * y)$	$\rightarrow$	y

Veamos también ahora una sencilla propiedad que nos permitirá caracterizar la terminación de los sistemas de reescritura.

Definición 7.16 *Un* **orden de reducción** *es un orden parcial bien fundamentado, estable y compatible.*

El concepto de orden de reducción nos da una condición necesaria y suficiente para que un SRT sea noetheriano:

Teorema 7.17 *Un SRT $\mathcal{R}$ es noetheriano si y sólo si existe un orden de reducción que lo contiene.*

Demostración:
Si $\succ$ es un orden de reducción que contiene a $\mathcal{R}$, por el lema 7.12 se tiene que $\rightarrow_{\mathcal{R}} \subseteq \succ$ y por tanto $\mathcal{R}$ será noetheriano. Además si $\mathcal{R}$ es noetheriano es evidente que $\stackrel{+}{\rightarrow}_{\mathcal{R}}$ es un orden de reducción que contiene a $\mathcal{R}$. □

Existe abundante literatura (véase[1]) sobre los órdenes de reducción, aunque queda fuera del objetivo de esta memoria la formalización de los mismos.

7.3.2 Sistemas de reescritura

Como las reglas de reescritura son un caso particular de los sistemas de ecuaciones, la representación en ACL2 de una regla de reescritura es exactamente igual a la de las ecuaciones: mediante un par punteado.

Los sistemas de reescritura de términos son un caso particular de los sistemas de ecuaciones. Usaremos, pues, la función `system-s-p` para reconocer a los SRTs, pero además comprobaremos que se verifican las restricciones sintácticas que deben tener las reglas de reescritura: una regla no ha de tener una variable como lado izquierdo y las variables de sus lados derechos han de estar incluidos en los lados izquierdos. Esta condición la comprueba la función `rewrite-rules`:

```
(defun rewrite-rules (R)
  (if (atom R)
      (equal R nil)
    (and (not (variable-p (lhs (car R))))
         (subsetp (variables t (rhs (car R)))
                  (variables t (lhs (car R))))
         (rewrite-rules (cdr R)))))
```

Con esta función ya podemos definir la función `rewrite-system-s-p`, que reconoce aquellos objetos ACL2 que representan a sistemas de reescritura de términos en una signatura:

```
(defmacro rewrite-system-s-p (R)
  `(and (system-s-p ,R)
        (rewrite-rules ,R)))
```

Ejemplo 7.18 La siguiente función `(RG)` define la representación en ACL2 del sistema de reescritura $\mathcal{R}_G$ definido en el ejemplo 7.15, considerando que la signatura esta dada por la función `signat-g` que se definió en la subsección 3.1.2:

```
(defun RG ()
  '(
      ( (* (* x y) z) . (* x (* y z)) )
      ( (* (e) x) . x )
      ( (* (i x) x) . (e) )
      ( (i (e)) . (e) )
      ( (* (i x) (* x y)) . y )
      ( (* x (e)) . x )
      ( (* x (i x)) . (e) )
      ( (i (i x)) . x )
      ( (i (* x y)) . (* (i y) (i x)) )
      ( (* x (* (i x) y)) . y )
    ))
```

7.3.3 Órdenes de reducción

Vamos a formalizar en esta subsección el concepto de sistema de reescritura noetheriano. Según se ha definido en los preliminares, un sistema de reescritura se dice noetheriano si lo es la reducción de reescritura asociada. Recuérdese que en la subsección 6.2.3 formalizamos la noetherianidad de una reducción como su inclusión en un orden bien fundamentado. En concreto, establecer que una reducción abstracta es noetheriana, significaba que:

- existe una relación transitiva bien fundamentada en el dominio de definición de la reducción y
- para todo elemento del dominio de definición y todo operador que le sea aplicable, el resultado de aplicar dicho operador es menor en dicho orden bien fundamentado.

El concepto de orden de reducción nos permite, sin embargo, simplificar la formalización de noetherianidad para el caso particular de las reducciones de reescritura. Así, para un orden de reducción fijado, podemos definir una función en ACL2 que compruebe, dado un sistema de reescritura, si la reducción de reescritura asociada está incluida en el orden de reducción. Expliquemos esto con detalle.

Definición de un orden de reducción

Usaremos `encapsulate` para definir parcialmente un orden de reducción `red<`:

```
(encapsulate
 ((red< (t1 t2) booleanp)
  (fn-red< (term) e0-ordinalp))
 ...
 (defthm red<-well-founded-relation-on-term-s-p
   (and (implies (term-s-p t1)
                 (e0-ordinalp (fn-red< t1)))
        (implies (and (term-s-p t1) (term-s-p t2)
                      (red< t1 t2))
                 (e0-ord-< (fn-red< t1) (fn-red< t2))))
        :rule-classes :well-founded-relation)

 (defthm red<-stable
   (implies (and (term-s-p t1) (term-s-p t2)
                 (substitution-s-p sigma)
                 (red< t1 t2))
            (red< (instance t1 sigma)
                  (instance t2 sigma))))

 (defthm red<-compatible
   (implies (and (term-s-p t1) (term-s-p t2) (term-s-p term)
                 (position-p pos term)
                 (red< t1 t2))
            (red< (replace-term term pos t1)
                  (replace-term term pos t2))))

 (defthm red<-transitive
   (implies (and (term-s-p t1) (term-s-p t2) (term-s-p t3)
                 (red< t1 t2) (red< t2 t3))
            (red< t1 t3))))
```

Como se observa, las propiedades que se asumen sobre `red<` (buena fundamentación, estabilidad, compatibilidad y transitividad) se asumen únicamente para los objetos ACL2

que representan términos en una signatura[5]. En particular, el asumir la buena fundamentación sólo sobre los objetos que representan términos, tiene consecuencias sobre la definición ACL2 de una función que calcula formas normales respecto de un sistema de reescritura cuya noetherianidad esté justificada mediante inclusión en un orden de reducción. Otro punto importante a destacar es que la buena fundamentación del orden de reducción definido tiene que justificarse mediante la existencia de una función `fn-red<` monótona que asocie un ordinal a cada término de la signatura. En la subsección siguiente comentaremos estas dos cuestiones nuevamente.

Noetherianidad justificada por un orden de reducción

Siguiendo la caracterización dada por el teorema 7.17, podemos definir formalmente en ACL2 la noetherianidad de un sistema de reescritura mediante una función que comprueba si el sistema de reescritura está incluido en un orden de reducción fijado. En concreto, la función `noetherian-red<` comprueba, dado un un sistema de reescritura si cada una de sus reglas está incluida en el orden de reducción `red<` definido previamente:

```
(defun noetherian-red< (R)
  (if (endp R)
      t
    (and (red< (rhs (car R)) (lhs (car R)))
         (noetherian-red< (cdr R)))))
```

Esta función formaliza en ACL2 el concepto de sistema de reescritura noetheriano. La ventaja que tiene el usar ordenes de reducción es que nos permite definir una función que comprueba la noetherianidad mediante comprobación de una determinada propiedad sobre el conjunto finito de reglas de reescritura del sistema.

Propiedades principales

Veamos que efectivamente la función `noetherian-red<` formaliza el concepto de sistema de reescritura noetheriano. Para ello necesitamos probar que todo sistema de reescritura `R` tal que `(noetherian-red< R)` tiene asociada una reducción de reescritura noetheriana. Es decir, aplicando un operador ecuacional legal a un término se obtiene un término que es menor respecto del orden bien fundamentado `red<`. El siguiente teorema establece tal resultado:

```
(defthm R-noetherian-if-subsetp-of-a-reduction-ordering
  (implies (and (system-s-p R)
                (noetherian-red< R)
                (term-s-p term)
                (eq-legal term op R))
           (red< (eq-reduce-one-step term op) term)))
```

La demostración de este teorema en ACL2 es una consecuencia directa y sencilla de la estabilidad y compatibilidad tanto del orden `red<` como de la reducción de reescritura. Véase la sección `4.2` del libro `rewriting.lisp` para más detalles.

[5]Por motivos técnicos, la macro `term-s-p` tiene que ser sustituida en el teorema de buena fundamentación por una función con la misma definición. Véase `rewriting.lisp` para más detalles.

7.3.4 Cálculo de formas normales

En esta subsección vamos a definir una función que calcula formas normales de términos respecto de sistemas de reescritura noetherianos. Para ello definimos, en primer lugar, una función que aplica un paso de reducción a un término, respecto de un SRT. Si el SRT es noetheriano, entonces una aplicación iterativa de tal función, hasta obtener un término irreducible, define un algoritmo de cálculo de formas normales.

Aplicación de un paso de reescritura: la función `r-reduce`

En la figura 7.1 aparece la definición de la función `r-reduce` que implementa la aplicación de un paso de reescritura a un término respecto de un sistema de reescritura dado, en caso de que el término sea reducible.

```
(defun r-reduce-aux (flg term R)
  (if flg
      (if (variable-p term)
          (mv nil nil)
        (mv-let (equ match top-red)
                (eq-reducible-top term R)
                (if top-red
                    (mv (instance (rhs equ) match) t)
                  (mv-let (reduced-args reducible-args)
                          (r-reduce-aux nil (cdr term) R)
                          (if reducible-args
                              (mv (cons (car term) reduced-args) t)
                            (mv nil nil))))))
    (if (endp term)
        (mv nil nil)
      (mv-let (reduced-first reducible-first)
              (r-reduce-aux t (car term) R)
              (if reducible-first
                  (mv (cons reduced-first (cdr term)) t)
                (mv-let (reduced-rest reducible-rest)
                        (r-reduce-aux nil (cdr term) R)
                        (if reducible-rest
                            (mv (cons (car term) reduced-rest) t)
                          (mv nil nil))))))))

(defun r-reduce (term R) (r-reduce-aux t term R))
```

Figura 7.1: Un paso de reescritura respecto de un SRT

Como se observa, la función `r-reduce` se basa principalmente en una función auxiliar `r-reduce-aux`. Esta función recibe como entrada un indicador `flg`, un término o lista de términos `term` (dependiendo del valor de `flg`) y un sistema de reescritura `R`. Recorre la estructura del término (o lista de términos) buscando un subtérmino que sea instancia del

lado izquierdo de una regla de `R`. En el caso de que lo encuentre, devuelve un multivalor consistente en dos valores: primero, el término resultante de sustituir en el término que se recibe como entrada, el subtérmino encontrado que es instancia del lado izquierdo de una regla de `R`, por la misma instancia del lado derecho de la regla. Como segundo valor devuelve `t` en ese caso. Si no existe tal subtérmino, se devuelve el multivalor `(mv nil nil)`. Es decir, el segundo valor del multivalor devuelto por `r-reduce-aux` es `nil` si y sólo si el término es irreducible. La función `r-reduce` se define simplemente como `r-reduce-aux` actuando sobre términos (es decir, para `flg` igual a `t`).

Los siguientes ejemplos de ejecución de la función `r-reduce` muestran cómo aplicar reiteradamente pasos de reescritura para obtener la forma normal x del término $x*(i(y)*(i(e)*i(i(y))))$ respecto del sistema de reescritura $\mathcal{R}_G$, definido en el ejemplo 7.15. Recuérdese que dicho sistema viene definido en ACL2 por la función `RG` (ejemplo 7.18):

```
ACL2 !>(r-reduce '(* (* x (i y)) (i (* (i y) (e)))) (RG))
((* X (* (I Y) (I (* (I Y) (E)))))   T)
ACL2 !>(r-reduce '(* x (* (i y) (i (* (i y) (e))))) (RG))
((* X (* (I Y) (* (I (E)) (I (I Y)))))   T)
ACL2 !>(r-reduce '(* x (* (i y) (* (i (e)) (i (i y))))) (RG))
((* X (* (I Y) (* (E) (I (I Y)))))   T)
ACL2 !>(r-reduce '(* x (* (i y) (* (e) (i (i y))))) (RG))
((* X (* (I Y) (I (I Y))))   T)
ACL2 !>(r-reduce '(* x (* (i y) (i (i y)))) (RG))
((* X (E)) T)
ACL2 !>(r-reduce '(* x (e)) (RG))
(X T)
ACL2 !>(r-reduce 'x (RG))
(NIL NIL)
```

La definición de la función `r-reduce-aux` es muy similar a la de la función `eq--reducible-aux`, ya que ambas recorren un término buscando una posición redex. Sin embargo existen diferencias entre ambas funciones, principalmente de índole práctica:

- La función `eq-reducible-aux` es un test de reducibilidad "teórico", ya que devuelve operadores ecuacionales. La función `r-reduce` no devuelve la posición, la regla y la sustitución con la que se aplica el paso de reducción: simplemente realiza el paso de reescritura, cuando sea posible.

- Sería posible definir un paso de reescritura usando las funciones `eq-reducible` y `eq--reduce-one-step`: la primera de ellas obtendría, si existe, un operador ecuacional aplicable y la segunda lo aplicaría. Sin embargo esto supondría recorrer dos veces el término: una para buscar el operador aplicable y otra para aplicarlo. La función `r-reduce` realiza el paso de reescritura (o detecta la irreducibilidad) recorriendo el término una sóla vez.

- Por último, la función `r-reduce-aux` está diseñada específicamente para sistemas de reescritura de términos, mientras que `eq-reducible-aux` lo está para cualquier sistema de ecuaciones. La diferencia estriba en que cuando se alcanza un subtérmino variable éste se desecha como posición redex, ya que por definición ninguna regla de un SRT puede subsumir a una variable.

Definición y propiedades del cálculo de formas normales

Como se observa en el ejemplo anterior, la función `r-reduce` proporciona el componente fundamental en el cálculo de formas normales respecto de un sistema de reescritura: basta iterar la aplicación de pasos de reescritura hasta que se alcanza un término irreducible. Para que este algoritmo se pueda definir en ACL2, ha de poderse demostrar que el proceso termina. Es evidente que si el SRT respecto del cual se calcula la forma normal no es noetheriano, la correspondiente función de cálculo de formas normales no puede ser admitida en la lógica de ACL2. Este es el motivo por el cual definiremos en ACL2 una función de cálculo de formas normales respecto de un SRT noetheriano que supondremos fijo y externo a la función.

Para no perder generalidad en nuestro razonamiento sobre el cálculo de formas normales, definiremos parcialmente con `encapsulate` un SRT asumiendo únicamente que es un sistema de reescritura noetheriano. A continuación definiremos una función que calcula formas normales respecto de dicho SRT noetheriano genérico.

El siguiente encapsulado define de manera genérica un sistema de reescritura `(RN)` noetheriano. Nótese que usamos la función `noetherian-red<` para expresar la noetherianidad de `(RN)`. Es decir, estamos suponiendo que la terminación de `(RN)` está justificada por el orden de reducción `red<`, también definido anteriormente de manera general. Según la caracterización dada por el (meta) teorema 7.17, cualquier sistema de reescritura noetheriano se ajusta a esta formalización.

```
(encapsulate
 ((RN () noetherian-rewrite-system))
 ...
 (defthm RN-rewrite-system
   (rewrite-system-s-p (RN)))

 (defthm RN-noetherian-red<
   (noetherian-red< (RN))))
```

Podemos ahora definir la función `RN-normal-form` que calcula una forma normal de un término con respecto del sistema de reescritura `(RN)`:

```
(defun RN-normal-form (term)
   (declare (xargs :measure (if (term-s-p term) term 0)
                   :well-founded-relation red<))
   (if (term-s-p term)
       (mv-let (reduced reducible)
               (r-reduce term (RN))
               (if reducible
                   (RN-normal-form reduced)
                 term))
     term))
```

Como se observa, la función `RN-normal-form` aplica exhaustivamente pasos de reescritura respecto de `(RN)` hasta que se obtiene un término irreducible. Esta función no es ejecutable, ya que el sistema de reescritura `(RN)` sólo está parcialmente definido. Sin

embargo, esa generalidad hace que sus propiedades sean trasladables a otras funciones que sean instancias concretas de ésta y que implementen el cálculo de formas normales para sistemas de reescritura específicos (siempre que sea posible demostrar en ACL2 su noetherianidad).

Es de destacar también que para la admisión de la función hemos de especificar una medida y un orden bien fundamentado. Como es de esperar, la relación bien fundamentada que justifica la terminación de `RN-normal-form` es la misma que justifica la noetherianidad del sistema de reescritura `(RN)`, el orden de reducción `red<`. Recuérdese que la buena fundamentación de `red<` está sólo asegurada en el conjunto de objetos que representan términos en una signatura (es decir, en el conjunto definido por el predicado `term-s-p`). Ese es el motivo de que la medida para su admisión venga dada por la expresión `(if (term-s-p term) term 0)`. Por ese mismo motivo, la función `RN-normal-form` comprueba si el objeto que recibe como entrada verifica `term-s-p`: para aquellos objetos ACL2 que no verifiquen `term-s-p` se comporta como la función identidad.

Veamos a continuación los teoremas que verifican formalmente las principales propiedades de la función `RN-normal-form`:

```
(defthm RN-normal-form-irreducible
  (implies (term-s-p term)
           (not (eq-legal (RN-normal-form term) op (RN)))))

(defthm RN-normal-form-equivalent-term
  (implies (term-s-p term)
           (eq-equiv-s-p term
                         (RN-normal-form term)
                         (RN-proof-irreducible term)
                         (RN))))
```

El primero de los teoremas establece que `RN-normal-form` obtiene, para cada término, un elemento irreducible. Recuérdese que por irreducibilidad entendemos aquí la inexistencia de operadores aplicables. El segundo de los teoremas muestra que el término devuelto por `RN-normal-form` es equivalente al que recibe como entrada, respecto de la teoría ecuacional de `(RN)`. La prueba que justifica tal equivalencia viene construida por la función `RN-proof-irreducible` que definiremos más adelante.

Descripción de la demostración

Describimos a continuación la demostración ACL2 de los dos teoremas anteriores y de la prueba de terminación de la función `RN-normal-form`. La clave para el razonamiento que vamos a hacer radica en establecer previamente la relación existente entre la función `r-reduce` y las funciones `eq-reducible` y `eq-reduce-one-step`. El siguiente teorema establece el nexo de unión entre ellas:

```
(defthm eq-reducible-p-iff-r-reduce
  (implies (rewrite-rules R)
           (iff (second (r-reduce term R))
                (eq-reducible term R))))
```

```
(defthm r-reduce-equal-eq-reduce-one-step-when-non-nil
  (implies (and (rewrite-rules R)
                (second (r-reduce term R)))
           (equal (first (r-reduce term R))
                  (eq-reduce-one-step
                   term
                   (eq-reducible term R)))))
```

El primer teorema expresa que `r-reduce` detecta la irreducibilidad de un término si y sólo si lo hace `eq-reducible`. El segundo establece que para términos reducibles, el término que obtiene `r-reduce` es el mismo que se obtiene aplicando, con la función `eq-reduce-one-step`, el operador obtenido con la función `eq-reducible`. Ambos resultados son sólo ciertos cuando la reducción está definida por reglas de reescritura (ya que, como se ha comentado, `r-reduce` descarta las posiciones redex correspondientes a variables).

Estos dos teoremas nos permiten realizar nuevamente un cierto tipo de razonamiento compuesto. La función `r-reduce` proporciona una función ejecutable más eficiente para llevar a cabo un paso de reescritura, aunque sin embargo el razonamiento lo hemos realizado sobre las funciones `eq-reducible` y `eq-reduce-one-step`. Los teoremas sobre `eq-reducible` y `eq-reduce-one-step` se trasladan de manera sencilla a `r-reduce` sin más que usar estos dos teoremas puente.

Veamos en primer lugar la prueba de admisión de la función `RN-normal-form`. Se trata de probar que aplicando un paso de reescritura a un término reducible, se obtiene un término menor respecto del orden bien fundamentado `red<`. O más concretamente, para todo término `term` tal que se verifica `(second (r-reduce term (RN)))`, el término `(first (r-reduce term (RN)))` es menor que `term` respecto de `red<`. Usando los teoremas de equivalencia entre `r-reduce` y las funciones `eq-reducible` y `eq-reduce-one-step`, la prueba de terminación se reduce a probar justamente el siguiente teorema:

```
(defthm RN-noetherian
  (implies (and (term-s-p term)
                (eq-legal term op (RN)))
           (red< (eq-reduce-one-step term op) term)))
```

Pero este teorema se tiene como caso particular (tomando `R` igual a `(RN)`) del teorema `R-noetherian-if-subsetp-of-a-reduction-ordering`, teorema que establecía que la reducción asociada a un sistema de reescritura que verifique `noetherian-red<` es noetheriana.

Por lo que se refiere a la demostración del teorema `RN-normal-form-irreducible`, se tiene por una sencilla inducción en el número de reducciones que realiza la función `RN-normal-form`. Para cualquier término `term` devuelto por `RN-normal-form` se verifica, por definición, `(not (second (r-reduce term (RN))))` (ya que es la condición de parada). Por el teorema `eq-reducible-p-iff-r-reduce`, eso es equivalente a decir que `(not (eq-reducible term (RN)))`. Finalmente, por el teorema `not-eq-reducible-nothing--eq-legal` (sección 7.2) se tiene el teorema buscado.

Por último, para expresar el teorema `RN-normal-form-equivalent-term` debemos definir la función `RN-proof-irreducible` que construye la prueba que justifica la equivalencia de un término con su forma normal:

```
(defun RN-proof-irreducible (term)
  (declare (xargs :measure (if (term-s-p term) term 0)
                  :well-founded-relation red<))
  (if (term-s-p term)
      (let ((red (eq-reducible term (RN))))
        (if (not red)
            nil
          (let ((term2 (eq-reduce-one-step term red)))
            (cons (make-r-step
                   :direct t :elt1 term :elt2 term2
                   :operator red)
                  (RN-proof-irreducible term2)))))
    nil))
```

Esta función construye la concatenación de pasos de prueba correspondientes a los pasos de reescritura que llevan a un término hasta su forma normal. Como es de esperar, tanto el esquema recursivo como la medida y orden bien fundamentado necesarios para su admisión son análogos a los de la función `RN-normal-form`.

Una simple prueba por inducción en el número de pasos de prueba permite demostrar el teorema `RN-normal-form-equivalent-term`. Nuevamente es fundamental en esta demostración el uso de los dos teoremas puente entre `r-reduce` y las funciones `eq-reducible` y `eq-reduce-one-step`.

Crítica de la formalización expuesta

La función `RN-normal-form` proporciona una definición genérica del concepto de forma normal respecto de un sistema de reescritura noetheriano, concepto sobre el que podemos razonar para verificar sus propiedades principales, como se ha hecho anteriormente. Sin embargo, como ya se ha comentado, al estar `RN` parcialmente definido con `encapsulate`, la función `RN-normal-form` no es ejecutable. Cumple, pues, el papel de formalizar un concepto, pero no sirve para ejecutarlo sobre objetos concretos.

Para sistemas de reescritura concretos, para los cuales se ha probado su noetherianidad, es posible definir, usando el mismo esquema que `RN-normal-form`, una función ejecutable que calcule formas normales. Las propiedades demostradas anteriormente sobre `RN-normal-form` se podrían trasladar fácilmente a estas versiones concretas y ejecutables, usando instanciación funcional. Sin embargo, aún en ese caso existen ciertas dificultades.

En primer lugar, es necesario definir un orden de reducción concreto, respecto de una signatura concreta y probar en la lógica de ACL2 que efectivamente lo es. En particular, la buena fundamentación del orden de reducción ha de ser justificada mediante una función monótona de los términos en los ordinales. En la literatura relacionada, las pruebas de buena fundamentación de los órdenes de reducción están basadas en su mayoría en el teorema de Kruskal (con una demostración no constructiva, [1]), por lo que parece difícil obtener una prueba ACL2 de la buena fundamentación de algunos de los órdenes de reducción más conocidos (órdenes RPO y KBO, por ejemplo).

Aún en el caso de obtener una prueba ACL2 de la noetherianidad de un SRT concreto en una signatura concreta, un algoritmo definido en ACL2 para el cálculo de formas normales respecto de tal SRT, análogo al definido por `RN-normal-form`, tendría una fuente de ineficiencia en su ejecución: si el orden de reducción usado para su admisión está bien

fundamentado sólo en el conjunto de términos de la signatura, entonces es necesario incluir en el código de su definición la comprobación de que recibe un objeto que representa a un término en la signatura (tal y como hace `RN-normal-form`). Esta comprobación se efectúa innecesariamente en cada llamada recursiva.

Sería posible relajar esta restricción si se probara la terminación del SRT mediante un orden de reducción que esté bien fundamentado en todo los objetos ACL2, en lugar de sólo en los que representan términos de una signatura. Sin embargo, en ese caso aumentaría aún mas la dificultad de probar que el orden de reducción está bien fundamentado: en la mayoría de los órdenes de reducción conocidos, su prueba de buena fundamentación está estrechamente ligada a la signatura en la que están definidos.

Una alternativa para el cálculo de formas normales

Es posible evitar parcialmente los problemas apuntados anteriormente, que surgen al tener que asegurar la terminación del cálculo de formas normales respecto de un sistema de reescritura. Se trata de efectuar un número finito de pasos de reescritura. Si en ese número finito de pasos se alcanza un término irreducible, se ha calculado una forma normal. La función `normal-form-n-steps` implementa esta idea:

```
(defun normal-form-n-steps (n term R)
  (mv-let (reduced reducible)
          (r-reduce term R)
          (cond ((not reducible) (mv term t))
                ((zp n) (mv nil nil))
                (t (normal-form-n-steps (- n 1) reduced R)))))
```

Esta función recibe un término `term`, un sistema de reescritura `R` y un número natural `n`. Si en menos de `n` pasos de reescritura se alcanza una forma normal, se devuelve un multivalor consistente en esa forma normal y en el valor `t`. En caso contrario, se devuelve `(mv nil nil)`.

Esta función es ejecutable, su admisión es trivial y no necesita que el SRT que recibe como entrada sea noetheriano. Además tampoco es necesario comprobar que `term` sea un término en una signatura dada. En el caso de que `n` sea lo suficientemente grande, `(normal-form-n-steps n term R)` calcula una forma normal de `term` respecto de `R`, como establece el siguiente teorema, fácilmente demostrado por inducción en el número de pasos de reescritura:

```
(defthm normal-form-n-steps-RN-normal-form-relation
  (implies (and (term-s-p term)
                (second (normal-form-n-steps n term (RN))))
           (equal (first (normal-form-n-steps n term (RN)))
                  (RN-normal-form term))))
```

Veamos algunos ejemplos de la ejecución de esta función:

```
ACL2 !>(normal-form-n-steps
           50 '(i (* (* x (i y)) (* y z))) (RG))
((* (I Z) (I X)) T)
```

```
ACL2 !>(normal-form-n-steps
           50 '(* (* x (i y)) (i (* (i y) (e)))) (RG))
(X T)
ACL2 !>(normal-form-n-steps
           5 '(* (* x (i y)) (i (* (i y) (e)))) (RG))
(NIL NIL)
ACL2 !>(normal-form-n-steps
           50 '(* (i x) (* (* x (i y)) (i (* (i y) (e))))) (RG))
((E) T)
ACL2 !>(normal-form-n-steps
           5 '(* (i x) (* (* x (i y)) (i (* (i y) (e))))) (RG))
(NIL NIL)
```

Como se observa en los ejemplos anteriores, la desventaja de esta definición es que es necesario conocer a priori el número de pasos que se necesitan para calcular la forma normal buscada. En la práctica, sin embargo, un número suficientemente grande bastará.

7.4 Confluencia: el teorema de pares críticos de Knuth y Bendix

Estudiamos en esta sección propiedades relativas a la confluencia de la reducción asociada a un sistema de ecuaciones. En concreto, definimos el concepto de par crítico determinado por dos ecuaciones y demostramos formalmente en ACL2 el teorema de pares críticos de Knuth y Bendix, según el cual la reducción ecuacional asociada a un sistema de ecuaciones es localmente confluente si convergen todos los pares críticos formados entre ecuaciones del sistema. Los eventos que conducen a los resultados presentados en esta sección se encuentran en el libro `critical-pairs.lisp`.

7.4.1 Preliminares

Antes de definir el concepto de par crítico, veamos algunas cuestiones de índole técnica:

Definición 7.19 *Dos ecuaciones $s \approx t$ y $s' \approx t'$ son equivalentes si $s \cdot t \equiv s' \cdot t'$, donde $\cdot$ es un símbolo de función binario arbitrario*[6]*.*

Es inmediato comprobar que la equivalencia de ecuaciones es una relación de equivalencia, ya que $\equiv$ lo es. Si $s \approx t$ y $s' \approx t'$ son equivalentes, se verifica que $s \equiv s'$ y $t \equiv t'$, pero no al revés: el renombrado de variables tiene que ser el mismo para todas las variables de la ecuación. Es lo que expresa el siguiente resultado, sencillo corolario del teorema 4.6.

Proposición 7.20 *Dos ecuaciones $s \approx t$ y $s' \approx t'$ son equivalentes si y sólo si existe un renombrado de las variables de $\mathcal{V}(s) \cup \mathcal{V}(t)$ de manera que $\theta(s) = s'$ y $\theta(t) = t'$.*

Teorema 7.21 *Dos ecuaciones $s \approx t$ y $u \approx v$ tienen* **variables separadas** *si $(\mathcal{V}(s) \cup \mathcal{V}(t)) \cap (\mathcal{V}(u) \cup \mathcal{V}(v)) = \emptyset$.*

[6] Extendiendo la signatura con el nuevo símbolo, si fuera necesario.

Pasemos a definir el concepto de par crítico. El lema de Newman (teorema 6.24) nos permite simplificar el estudio de la confluencia de las reducciones y, en concreto, de las reducciones ecuacionales: la comprobación de la confluencia de una reducción ecuacional noetheriana, se limita a establecer la convergencia de todos los pares de términos s y t tales que existe un término u cumpliendo $u \to_E s$ y $u \to_E t$ (es decir, su confluencia local). Sin embargo, en general existen infinitos pares de términos que verifican tal propiedad. El lema de Knuth y Bendix nos permite efectuar esta comprobación sólo para una cantidad finita (cuando E sea finito) de pares de términos, llamados pares críticos.

Definición 7.22 *Sean:*

- $l_1 \approx r_1$ *y* $l_2 \approx r_2$ *dos ecuaciones en* $T(\Sigma, X)$,
- $l_1' \approx r_1'$ *y* $l_2' \approx r_2'$ *dos ecuaciones con variables separadas y equivalentes, respectivamente, a las ecuaciones* $l_1 \approx r_1$ *y* $l_2 \approx r_2$,
- $p \in \mathcal{P}(l_1)$ *(y por tanto* $p \in \mathcal{P}(l_1')$*) tal que* $l_1/p \notin X$ *y* σ *unificador de máxima generalidad de* l_1'/p *y* l_2',
- $u = \sigma(r_1')$ *y* $v = \sigma(l_1'[p \leftarrow r_2'])$.

Decimos entonces que el par de términos (u, v) *es* **un par crítico** *determinado por la* **superposición** *de* $l_2 \approx r_2$ **sobre** $l_1 \approx r_1$ **en** p.

Ejemplo 7.23 Un par crítico determinado por la superposición de la ecuación $i(x) * x \approx e$ sobre la ecuación $(x * y) * z \approx x * (y * z)$ en la posición 1 es $(e * z, i(v) * (v * z))$. En este caso, la primera ecuación se renombra con la sustitución $\{x \mapsto v\}$ y la segunda se deja igual. El unificador de máxima generalidad usado es la sustitución $\{x \mapsto i(v), y \mapsto v\}$.

Intuitivamente, un par crítico representa la forma más general de pico local en una reducción ecuacional. Es lo que viene a demostrar el siguiente lema.

Lema 7.24 *Sean:*

- $l_1 \approx r_1$ *y* $l_2 \approx r_2$ *dos ecuaciones en* $T(\Sigma, X)$,
- $p \in \mathcal{P}(l_1)$ *tal que* $l_1/p \notin X$,
- σ_1 *y* σ_2 *sustituciones tales que* $\sigma_1(l_1/p) = \sigma_2(l_2)$,
- $u_1 = \sigma_1(r_1)$ *y* $v_1 = \sigma_1(l_1)[p \leftarrow \sigma_2(r_2)]$.

Entonces existe una sustitución δ *y un par crítico* (u, v) *determinado por la superposición de* $l_2 \approx r_2$ *sobre* $l_1 \approx r_1$ *en* p*, tal que* $u_1 = \delta(u)$ *y* $v_1 = \delta(v)$.

Demostración:

Veamos en primer lugar que existe el par crítico determinado por la superposición de $l_2 \approx r_2$ sobre $l_1 \approx r_1$ en p. Sean $l_1' \approx r_1'$ y $l_2' \approx r_2'$ un par de ecuaciones con variables separadas y equivalentes, respectivamente, a las ecuaciones $l_1 \approx r_1$ y $l_2 \approx r_2$. Entonces, por el lema 7.20, existen dos sustituciones θ_1 y θ_2 tales que $\theta_1(l_1' \approx r_1') = l_1 \approx r_1$ y $\theta_2(l_2' \approx r_2') = l_2 \approx r_2$. Sean $V_1 = \mathcal{V}(l_1') \cup \mathcal{V}(r_1')$ y $V_2 = \mathcal{V}(l_2') \cup \mathcal{V}(r_2')$. Como las ecuaciones

renombradas tienen variables separadas, se verifica $V_1 \cap V_2 = \emptyset$. Sea μ la sustitución definida de la siguiente manera:

$$\mu = (\sigma_1\theta_1)_{|V_1} \cup (\sigma_2\theta_2)_{|V_2}$$

Esta unión de sustituciones está bien definida, ya que sus dominios son disjuntos. Se verifica que $\mu(l'_1/p) = \sigma_1(\theta_1(l'_1/p)) = \sigma_1(l_1/p) = \sigma_2(l_2) = \mu(l'_2)$, y por tanto μ es un unificador de l'_1/p y l'_2.

Esto significa que existe una sustitución σ, unificador de máxima generalidad de ambos términos y un par crítico $(u, v) = (\sigma(r'_1), \sigma(l'_1[p \leftarrow r'_2]))$.

Además, como μ es unificador de l'_1/p y l'_2 y σ unificador de máxima generalidad, existe δ tal que $\mu = \delta\sigma$. Se tiene entonces que

$$\delta(u) = \delta(\sigma(r'_1)) = \mu(r'_1) = \sigma_1(\theta_1(r'_1)) = \sigma_1(r_1) = u_1, \text{ y}$$

$$\delta(v) = \delta(\sigma(l'_1[p \leftarrow r'_2])) = \mu(l'_1)[p \leftarrow \mu(r'_2)] = \sigma_1(l_1)[p \leftarrow \sigma_2(r_2)] = v_1$$

□

En la situación descrita por este teorema, diremos que existe una **superposición crítica** de la ecuación $l_2 \approx r_2$ en la posición p del lado izquierdo de la ecuación $l_1 \approx r_1$.

Un par crítico viene determinado por dos ecuaciones y una posición no variable en el lado izquierdo de una de ellas. De la definición 7.22, concluiríamos que también lo determinan los renombrados concretos que se usen para separar las variables de las ecuaciones. Sin embargo:

Proposición 7.25 *Dado un par de ecuaciones y una posición en el lado izquierdo de una de ellas, el par crítico que determinan (si existe) es único módulo* $\equiv$.

Demostración:
Sean (u, v) y (u', v') dos pares críticos determinados por la superposición de $l_2 \approx r_2$ sobre $l_1 \approx r_1$ en $p \in \mathcal{P}(l_1)$. Entonces, por el lema 7.24 se tiene que si $\cdot$ es un operador binario (notación infija), entonces $u \cdot v \preccurlyeq u' \cdot v'$ y $u' \cdot v' \preccurlyeq u \cdot v$. Luego $u \cdot v \equiv u' \cdot v'$ y por tanto $u \equiv u'$ y $v \equiv v'$. □

Así pues podemos hablar, con cierto abuso del lenguaje, *del* par crítico determinado por un par de ecuaciones y una posición en el lado izquierdo de una de ellas. Además consideraremos iguales, salvo mención expresa, dos pares críticos equivalentes por $\equiv$.

Definición 7.26 *Sea E un sistema de ecuaciones en $T(\Sigma, X)$. El* **conjunto de pares críticos** *de E es el conjunto de todos los pares críticos determinados por la superposición de dos ecuaciones de E.*

Notaremos por $pc(E)$ al conjunto de todos los pares críticos del sistema de ecuaciones E. Nótese que $pc(E)$ es un sistema de ecuaciones, lo que nos permite formular el siguiente teorema:

Teorema 7.27 (**Lema de pares críticos de Knuth-Bendix**). *Sea E un sistema de ecuaciones en $T(\Sigma, X)$ y u, s, t términos tales que $u \to_E s$ y $u \to_E t$. Entonces $s \downarrow_E t$ ó $s \longleftrightarrow_{pc(E)} t$.*

Demostración:
Como $u \to_E s$, entonces existe $p_1 \in \mathcal{P}(u)$, $l_1 \approx r_1 \in E$ y σ_1 sustitución tal que $u/p_1 = \sigma_1(l_1)$ y $s = u[p_1 \leftarrow \sigma_1(r_1)]$. De la misma manera $u \to_E t$, implica que existe $p_2 \in \mathcal{P}(u)$, $l_2 \approx r_2 \in E$ y σ_2 sustitución tal que $u/p_2 = \sigma_2(l_2)$ y $t = u[p_2 \leftarrow \sigma_2(r_2)]$.

Distinguimos dos posibles casos según las posiciones relativas de p_1 y p_2.

Caso 1 $p_1 | p_2$.

Por el teorema 3.25 (apartado 4), se tiene $s/p_2 = u[p_1 \leftarrow \sigma_1(r_1)]/p_2 = u/p_2$, luego $s/p_2 = \sigma_2(l_2)$. Por el mismo motivo $t/p_1 = u[p_2 \leftarrow \sigma_2(r_2)]/p_1 = u/p_1$ y $t/p_1 = \sigma_1(l_1)$. Por tanto:

$$s \to_E s[p_2 \leftarrow \sigma_2(r_2)] = u[p_1 \leftarrow \sigma_1(r_1)][p_2 \leftarrow \sigma_2(r_2)]$$

Análogamente

$$t \to_E t[p_1 \leftarrow \sigma_1(r_1)] = u[p_2 \leftarrow \sigma_2(r_2)][p_1 \leftarrow \sigma_1(r_1)]$$

De la conmutatividad enunciada en el teorema 3.25 (4 c) se sigue que $s \downarrow_E t$.

Caso 2 $p_1 \leq p_2$.

Sea q tal que $p_2 = p_1 \cdot q$. Por el teorema 3.25 (apartado 3) y por ser $p_2 = p_1 \cdot q \in \mathcal{P}(u)$, se tiene que $q \in \mathcal{P}(u/p_1) = \mathcal{P}(\sigma_1(l_1))$ y $\sigma_1(l_1)/q = \sigma_2(l_2)$. Entonces $s = u[p_1 \leftarrow \sigma_1(r_1)]$ y

$$t = u[p_2 \leftarrow \sigma_2(r_2)] = u[p_1 \leftarrow \sigma_1(l_1)][p_2 \leftarrow \sigma_2(r_2)] =$$

$$= u[p_1 \leftarrow \sigma_1(l_1)][p_1 q \leftarrow \sigma_2(r_2)] = u[p_1 \leftarrow \sigma_1(l_1)[q \leftarrow \sigma_2(r_2)]]$$

(esta última igualdad nuevamente por el teorema 3.25, apartado 3). Si probamos que

$$\sigma_1(r_1) \downarrow_E \sigma_1(l_1)[q \leftarrow \sigma_2(r_2)] \text{ ó}$$

$$\sigma_1(r_1) \longleftrightarrow_{pc(E)} \sigma_1(l_1)[q \leftarrow \sigma_2(r_2)],$$

entonces por la compatibilidad de $\to_E$ y de $\longleftrightarrow_{pc(E)}$ se tendría que $s \downarrow_E t$ ó $s \longleftrightarrow_{pc(E)} t$.

Probémoslo. Como $q \in \mathcal{P}(\sigma_1(l_1))$, se tiene que o bien $q \in \mathcal{P}(l_1)$ y $l_1/q \notin X$, o bien $q = q_1 \cdot q_2$ con $q_1 \in \mathcal{P}(l_1), l_1/q_1 = x \in X, q_2 \in \mathcal{P}(\sigma_1(x))$ y $\sigma_1(x)/q_2 = \sigma_2(l_2)$. Por tanto existen dos posibles subcasos:

Subcaso 2a $q = q_1 \cdot q_2$ con $q_1 \in \mathcal{P}(l_1), l_1/q_1 = x \in X, q_2 \in \mathcal{P}(\sigma_1(x))$ y $\sigma_1(x)/q_2 = \sigma_2(l_2)$. Es decir p_2 queda más "profunda" que la estructura de l_1. En ese caso decimos que existe una **superposición variable**.

Sea σ_1' la sustitución definida de la siguiente manera:

$$\sigma_1'(x) = \sigma_1(x)[q_2 \leftarrow \sigma_2(r_2)]$$

$$\sigma_1'(y) = \sigma_1(y) \quad \text{si } y \neq x \ .$$

Es evidente que $\sigma_1(x) \to_E \sigma_1'(x)$. Por la compatibilidad de $\to_E$, se tiene que $\sigma_1(r_1) \overset{*}{\to}_E \sigma_1'(r_1)$ (reescribiendo con $l_2 \approx r_2$ en los subtérminos que ocurren en las posiciones q' de $\sigma_1(r_1)$ tales que $q' \in \mathcal{P}(r_1)$ y $r_1/q' = x$). Además, $\sigma_1(l_1)[q \leftarrow \sigma_2(r_2)] \overset{*}{\to}_E \sigma_1'(l_1)$ (reescribiendo con $l_2 \approx r_2$ en los subtérminos que ocurren en las posiciones q'' de $\sigma_1(l_1)$ tales que $q'' \in \mathcal{P}(l_1)$ y $l_1/q'' = x$, excepto en q_1). Como además $\sigma_1'(l_1) \to_E \sigma_1'(r_1)$, entonces

$$\sigma_1(r_1) \overset{*}{\to}_E \sigma_1'(r_1) \overset{*}{\leftarrow}_E \sigma_1(l_1)[q \leftarrow \sigma_2(r_2)].$$

Subcaso 2b $q \in \mathcal{P}(l_1)$ y $l_1/q \notin X$ (es decir, existe una superposición crítica de la ecuación $l_2 \approx r_2$ en la posición q de l_1).

Tenemos, por tanto, que $\sigma_1(l_1/q) = \sigma_1(l_1)/q = \sigma_2(l_2)$. Podemos aplicar el lema 7.24 y deducir que si (u, v) es el par crítico determinado por la superposición de $l_2 \approx r_2$ sobre $l_1 \approx r_1$ en q, entonces existe una sustitución δ tal que $\delta(v) = \sigma_1(l_1)[q \leftarrow \sigma_2(r_2)]$ y $\delta(u) = \sigma_1(r_1)$. Luego por la estabilidad de $\longleftrightarrow_{pc(E)}$ se tiene que

$$\sigma_1(r_1) \longleftrightarrow_{pc(E)} \sigma_1(l_1)[q \leftarrow \sigma_2(r_2)].$$

Caso 3 $p_2 \leq p_1$. El razonamiento es totalmente simétrico al caso 2, cambiando los papeles de s y t y teniendo en cuenta que las relaciones $\overset{*}{\leftrightarrow}_E$ y $\longleftrightarrow_{pc(E)}$ son simétricas.

□

El lema anterior nos lleva a una caracterización de la confluencia local de la reducción $\rightarrow_E$, que limita el número de "divergencias" locales a considerar:

Corolario 7.28 *(***Teorema de pares críticos de Knuth-Bendix***) Sea E un sistema de ecuaciones en $T(\Sigma, X)$. Entonces se verifica que $\rightarrow_E$ es localmente confluente si y sólo si para cualquier $(u, v) \in pc(E)$, se tiene $u \downarrow_E v$.*

Demostración:
Si (u, v) es un par crítico determinado por la superposición de $l_2 \approx r_2$ sobre $l_1 \approx r_1$, entonces $\sigma_1(l_1) \rightarrow_E u$ y $\sigma_1(l_1) \rightarrow_E v$ (con la notación de la definición 7.22). Por tanto, si $\rightarrow_E$ es localmente confluente, entonces $u \downarrow_E v$.

Por otro lado, de la compatibilidad y estabilidad de $\rightarrow_E$ se deduce que si para cualquier $(u, v) \in pc(E)$ se tiene que $u \downarrow_E v$, entonces si $t_1 \longleftrightarrow_{pc(E)} t_2$, $t_1 \downarrow_E t_2$. Por tanto del lema de pares críticos de Knuth-Bendix se deduce que si $s \rightarrow_E t_1$ y $s \rightarrow_E t_2$, entonces $t_1 \downarrow_E t_2$ (o lo que es lo mismo, $\rightarrow_E$ es localmente confluente). □

7.4.2 El teorema de los pares críticos de Knuth y Bendix

Presentamos aquí la formalización que hemos llevado a cabo del teorema 7.28 en la lógica de ACL2. Las definiciones, lemas y teoremas que se presentan a continuación, se encuentran en el libro `critical-pairs.lisp`. Sólo describimos aquí la implicación no trivial del teorema. Es decir, suponiendo que E es un sistema de ecuaciones tal que todos sus pares críticos convergen, hemos de probar que la reducción $\rightarrow_E$ es localmente confluente. La implicación contraria es trivial.

Pares críticos

La siguiente función `cp` calcula el par crítico (si existe) determinado por la superposición de una ecuación $l_2 \approx r_2$ sobre una posición p no variable de otra ecuación $l_1 \approx r_1$, suponiendo que ambas reglas tienen variables separadas. En el caso de que tal par crítico no exista, la función `cp` devuelve `nil`. Nótese el uso del algoritmo de unificación `mgu-mv`.

```
(defun cp (l1 r1 p l2 r2)
  (mv-let (theta unifiable)
```

```
          (mgu-mv (occurrence l1 p) l2)
          (if unifiable
              (cons (instance r1 theta)
                    (instance (replace-term l1 p r2) theta))
            nil)))
```

Para definir el par crítico entre dos ecuaciones cualesquiera, es necesario renombrar las ecuaciones para separar sus variables. La proposición 7.25 nos permite escoger cualquier tipo de renombrado para separar las variables de las ecuaciones involucradas. En nuestro caso, usaremos `number-rename-list` (subsección 4.1.5). La macro `number-rename--equation` nos servirá de abreviatura para el renombrado de ecuaciones:

```
(defmacro number-rename-equation (l r x y)
   `(number-rename-list (list ,l ,r) ,x ,y))
```

Así `(number-rename-equation l r x y)` es la ecuación obtenida renombrando las variables de `l` y `r` usando números, comenzando en `x` y aumentando la última asignación realizada en `y`, para cada nueva variable.

Una vez definido el renombrado de ecuaciones, podemos definir la función `cp-r` que formaliza el concepto de par crítico tal y como lo hace la definición 7.22. Esta función recibe dos ecuaciones, determinadas por los cuatro términos `l1`, `r1`, `l2` y `r2`, y una posición `p` no variable de `l1` y devuelve el par crítico que determina la segunda regla sobre la posición `p` de la primera, en el caso de que exista (`nil` en caso contrario):

```
(defun cp-r (l1 r1 p l2 r2)
  (let* ((eq1-r (number-rename-equation l1 r1 0 -1))
         (eq2-r (number-rename-equation l2 r2 1 1))
         (l1-r (nth 0 eq1-r)) (r1-r (nth 1 eq1-r))
         (l2-r (nth 0 eq2-r)) (r2-r (nth 1 eq2-r)))
    (cp l1-r r1-r p l2-r r2-r)))
```

Formalización del teorema

Para formalizar las hipótesis del teorema de pares críticos, suponemos la existencia de un sistema de ecuaciones, que llamaremos `(EKB)`, tal que todos sus pares críticos convergen. El siguiente encapsulado establece formalmente tales hipótesis:

```
(encapsulate
 ((EKB () system-with-joinable-critical-pairs)
  (transform-critical-pair (l1 r1 p l2 r2) valley-proof))
 ....
 (defthm EKB-system-s-p
   (system-s-p (EKB)))

 (defthm EKB-joinable-critical-pairs
   (implies (and (member (make-equation l1 r1) (EKB))
                 (member (make-equation l2 r2) (EKB))
                 (position-p p l1)
```

```
        (not (variable-p (occurrence l1 p))))
   (let ((cp-r (cp-r l1 r1 p l2 r2))
         (valley (transform-critical-pair l1 r1 p l2 r2)))
     (implies cp-r
              (and
                (eq-equiv-s-p (lhs cp-r) (rhs cp-r) valley (EKB))
                (steps-valley valley)))))))
```

La macro `make-equation` es simplemente un nombre alternativo para `cons`, haciendo énfasis así en que la usaremos para construir la ecuación que determinan sus dos términos:

```
(defmacro make-equation (lhs rhs) `(cons ,lhs ,rhs))
```

La convergencia de cada par crítico determinado por dos ecuaciones $l_1 \approx r_1$ y $l_2 \approx r_2$ en una posición p no variable de l_1, viene justificada por la existencia de una prueba valle para cada uno de tales pares críticos. En la lógica de ACL2, expresamos esta propiedad asumiendo que existe una función `transform-critical-pair` tal que, recibiendo como entrada un par de ecuaciones de `(EKB)` y una posición, devuelve una prueba valle ecuacional que demuestra la equivalencia de los dos componentes del par crítico que determinen, en su caso.

Una vez asumidas las hipótesis del teorema 7.28, hemos de demostrar que la reducción ecuacional correspondiente al sistema de ecuaciones `(EKB)` es localmente confluente. Siguiendo la formalización de la confluencia local que se presentaba en la subsección 6.2.3 para reducciones abstractas, debemos *definir* una función, que llamaremos `transform-eq-local-peak`, y *demostrar* que para todo pico local ecuacional `p` en `(EKB)`, `(transform-eq-local-peak p)` es una prueba valle equivalente. Formalmente:

```
(defthm knuth-bendix-theorem
  (implies
   (and (eq-equiv-s-p t1 t2 p (EKB))
        (local-peak-p p))
   (and (eq-equiv-s-p t1 t2 (transform-eq-local-peak p) (EKB))
        (steps-valley (transform-eq-local-peak p)))))
```

El resultado `knuth-bendix-theorem` establece formalmente en la lógica de ACL2 el teorema de pares críticos de Knuth y Bendix.

A continuación describimos en líneas generales la demostración del teorema `knuth-bendix-theorem` en ACL2, la más laboriosa de las presentadas en esta memoria. En esencia, seguimos la demostración a mano presentada en el teorema 7.28. En consonancia con esta demostración, hemos divido la descripción de la misma en cuatro bloques:

- Estudio de los picos locales correspondientes a reescritura en posiciones disjuntas (el caso 1 del teorema 7.28)
- Estudio de las superposiciones variables (caso 2a del teorema 7.28).
- Estudio de las superposiciones críticas (teorema 7.24 y caso 2b del teorema 7.28).
- Finalmente, recopilamos los tres casos anteriores para definir la función `transform-eq-local-peak` y demostrar el teorema `knuth-bendix-theorem`.

En lo que sigue, por *resolver* cada uno de los casos anteriores entenderemos el mostrar de la existencia de una prueba valle equivalente para cada caso, definiendo la función correspondiente que obtiene dicha prueba y demostrando sus propiedades.

7.4.3 Reescritura en posiciones disjuntas

Vamos a ver en esta subsección que si tenemos un pico local correspondiente a reescrituras en posiciones disjuntas, podemos construir una prueba valle equivalente. Se trata, pues, de demostrar formalmente el caso 1 del teorema 7.28.

Recuérdese que en el principio de la prueba a mano del teorema 7.28 se afirmaba:

> *Como* $u \rightarrow_E s$*, entonces existe* $p_1 \in \mathcal{P}(u)$*,* $l_1 \approx r_1 \in E$ *y* σ_1 *sustitución tal que* $u/p_1 = \sigma_1(l_1)$ *y* $s = u[p_1 \leftarrow \sigma_1(r_1)]$*. De la misma manera* $u \rightarrow_E t$*, implica que existe* $p_2 \in \mathcal{P}(u)$*,* $l_2 \approx r_2 \in E$ *y* σ_2 *sustitución tal que* $u/p_2 = \sigma_2(l_2)$ *y* $t = u[p_2 \leftarrow \sigma_2(r_2)]$*.*

Definimos un predicado `eq-local-peak-p` que explicita los elementos que intervienen en un pico local ecuacional y las relaciones existentes entre ellos:

```
(defun eq-local-peak-p (peak t1 t2 p1 p2 l1 r1 l2 r2 sigma1 sigma2 E)
  (and (system-s-p E)
       (term-s-p peak) (term-s-p t1) (term-s-p t2)
       (substitution-s-p sigma1) (substitution-s-p sigma2)
       (position-p p1 peak) (position-p p2 peak)
       (member (make-equation l1 r1) E)
       (member (make-equation l2 r2) E)
       (equal (occurrence peak p1) (instance l1 sigma1))
       (equal (occurrence peak p2) (instance l2 sigma2))
       (equal (replace-term peak p1 (instance r1 sigma1)) t1)
       (equal (replace-term peak p2 (instance r2 sigma2)) t2))))
```

Así, el párrafo anterior de la prueba a mano se corresponde, en nuestra formalización, con asumir que se verifica `(eq-local-peak-p peak t1 t2 p1 p2 l1 r1 l2 r2 sigma1 sigma2 E)`.

En el caso de reescritura en posiciones disjuntas, supondremos además que se verifica `(disjoint-positions p1 p2)`. Para resolver este caso definimos una función `transform--disjoint-eq-local-peak` que actuando sobre los componentes de un pico local ecuacional disjunto, obtiene una prueba valle equivalente a dicho pico local. El siguiente párrafo de la prueba a mano nos proporciona la idea intuitiva:

Por el teorema 3.25 (apartado 4), se tiene $s/p_2 = u[p_1 \leftarrow \sigma_1(r_1)]/p_2 = u/p_2$, *luego* $s/p_2 = \sigma_2(l_2)$. *Por el mismo motivo* $t/p_1 = u[p_2 \leftarrow \sigma_2(r_2)]/p_1 = u/p_1$ *y* $t/p_1 = \sigma_1(l_1)$. *Por tanto:*

$$s \rightarrow_E s[p_2 \leftarrow \sigma_2(r_2)] = u[p_1 \leftarrow \sigma_1(r_1)][p_2 \leftarrow \sigma_2(r_2)]$$

Análogamente

$$t \rightarrow_E t[p_1 \leftarrow \sigma_1(r_1)] = u[p_2 \leftarrow \sigma_2(r_2)][p_1 \leftarrow \sigma_1(r_1)]$$

De lo anterior y de la conmutatividad enunciada en el teorema 3.25 (4 c) se sigue que $s \downarrow_E t$.

Es decir, reescribiendo `t1` en la posición `p2` y `t2` en la posición `p1` se obtiene el mismo elemento y por tanto una prueba valle que conecta `t1` y `t2`. Esta construcción de la prueba valle se formaliza mediante la siguiente función `transform-disjoint-eq-local-peak`:

```
(defun transform-disjoint-eq-local-peak
  (peak t1 t2 p1 p2 l1 r1 l2 r2 sigma1 sigma2)
  (list (make-r-step
         :direct t
         :elt1 t1
         :elt2 (replace-term t1 p2 (instance r2 sigma2))
         :operator (make-eq-operator
                    :pos p2
                    :rule (cons l2 r2)
                    :match sigma2))
        (make-r-step
         :direct nil
         :elt1 (replace-term t2 p1 (instance r1 sigma1))
         :elt2 t2
         :operator (make-eq-operator
                    :pos p1
                    :rule (cons l1 r1)
                    :match sigma1))))
```

Los siguientes teoremas resuelven el caso de reescritura en posiciones disjuntas, estableciendo que efectivamente esta función obtiene una prueba valle de la equivalencia de `t1` y `t2`. Nótese que el resultado es cierto para cualquier sistema de ecuaciones `E` y no sólo para el sistema `(EKB)`:

```
(defthm transform-disjoint-eq-local-peak-is-a-proof
  (implies (and (eq-local-peak-p peak
                  t1 t2 p1 p2 l1 r1 l2 r2 sigma1 sigma2 E)
                (disjoint-positions p1 p2))
           (eq-equiv-s-p t1 t2
                         (transform-disjoint-eq-local-peak
                           peak t1 t2 p1 p2 l1 r1 l2 r2 sigma1 sigma2)
                         E)))
```

```
(defthm transform-disjoint-eq-local-peak-is-a-valley
  (steps-valley
   (transform-disjoint-eq-local-peak
    peak t1 t2 p1 p2 l1 r1 l2 r2 sigma1 sigma2)))
```

Como en la prueba a mano, la prueba del teorema `transform-disjoint-eq-local--peak-is-a-proof` necesita de los teoremas relativos a posiciones, ocurrencias y reemplazamiento presentados en la sección 3.2 (para el caso de posiciones disjuntas).

7.4.4 Superposición variable

Siguiendo con la notación de la prueba a mano, resolver el caso correspondiente a una superposición variable consiste en encontrar una prueba valle equivalente para el pico local descrito por la siguiente situación:

$\sigma_1(r_1) \leftarrow_E \sigma_1(l_1) \rightarrow_E \sigma_1(l_1)[q \leftarrow \sigma_2(r_2)]$
donde $q = q_1 \cdot q_2$, $q_1 \in \mathcal{P}(l_1), l_1/q_1 = x \in X, q_2 \in \mathcal{P}(\sigma_1(x))$ *y* $\sigma_1(x)/q_2 = \sigma_2(l_2)$.

El siguiente predicado `eq-top-variable-overlap-p` describe tal situación:

```
(defun eq-top-variable-overlap-p
  (peak u1 u2 q1 q2 sigma1 sigma2 l1 r1 l2 r2 x val valr E)
  (and (term-s-p u1) (term-s-p u2) (term-s-p peak) (system-s-p E)
       (substitution-s-p sigma1) (substitution-s-p sigma2)
       (member (make-equation l1 r1) E)
       (member (make-equation l2 r2) E)
       (equal peak (instance l1 sigma1))
       (position-p q1 l1)
       (equal (occurrence l1 q1) x)
       (variable-p x)
       (equal val (val x sigma1))
       (position-p q2 val)
       (equal (occurrence val q2) (instance l2 sigma2))
       (equal valr (replace-term val q2 (instance r2 sigma2)))
       (equal u1 (instance r1 sigma1))
       (equal u2 (replace-term peak q1 valr)))))
```

Como se observa, en esta definición `peak` representa término $\sigma_1(l_1)$. También `u1` y `u2`, son respectivamente los términos $\sigma_1(r_1)$ y $\sigma_1(l_1)[q_1 \leftarrow \sigma_1(x)[q_2 \leftarrow \sigma_2(r_2)]] = \sigma_1(l_1)[q \leftarrow \sigma_2(r_2)]$. Además `val` representa a $\sigma_1(x)$ y `valr` el resultado de reescribir dicho término en la posición q_2, usando la ecuación $l_2 \approx r_2$ (es decir, $\sigma_1(x)[q_2 \leftarrow \sigma_2(r_2)]$). En lo que sigue, usaremos estos nombres de variables y los términos que representan, indistintamente.

Resolver este caso consiste en *definir* una función `transform-eq-top-variable-overlap`, que actuando sobre los elementos de una superposición variable, obtiene una prueba valle que justifica la equivalencia de los términos `u1` y `u2`. Es decir, el caso quedará resuelto si se prueban los siguientes teoremas:

```
(defthm transform-eq-top-variable-overlap-is-a-proof
  (implies
   (eq-top-variable-overlap-p
     peak u1 u2 q1 q2 sigma1 sigma2 l1 r1 l2 r2 x val valr E)
   (eq-equiv-s-p u1 u2
    (transform-eq-top-variable-overlap
     peak u1 u2 q1 q2 sigma1 sigma2 l1 r1 l2 r2 x val valr)
    E)))

(defthm transform-eq-top-variable-overlap-is-a-valley
  (steps-valley
   (transform-eq-top-variable-overlap
    peak u1 u2 q1 q2 sigma1 sigma2 l1 r1 l2 r2 x val valr)))
```

Sorprendentemente (o quizá no tanto), la resolución de este caso es la más compleja y laboriosa de la prueba del teorema de pares críticos. La demostración está basada en la siguiente idea, presentada en la prueba a mano:

Sea σ_1' la sustitución definida de la siguiente manera:

$$\sigma_1'(x) = \sigma_1(x)[q_2 \leftarrow \sigma_2(r_2)]$$

$$\sigma_1'(y) = \sigma_1(y) \quad \textit{si } y \neq x \ .$$

Es evidente que $\sigma_1(x) \rightarrow_E \sigma_1'(x)$. Por la compatibilidad de $\rightarrow_E$, se tiene que $\sigma_1(r_1) \stackrel{*}{\rightarrow}_E \sigma_1'(r_1)$ (reescribiendo con $l_2 \approx r_2$ en los subtérminos que ocurren en las posiciones q' de $\sigma_1(r_1)$ tales que $q' \in \mathcal{P}(r_1)$ y $r_1/q' = x$). Además, $\sigma_1(l_1)[q \leftarrow \sigma_2(r_2)] \stackrel{*}{\rightarrow}_E \sigma_1'(l_1)$ (reescribiendo con $l_2 \approx r_2$ en los subtérminos que ocurren en las posiciones q'' de $\sigma_1(l_1)$ tales que $q'' \in \mathcal{P}(l_1)$ y $l_1/q'' = x$, excepto en q_1). Como además $\sigma_1'(l_1) \rightarrow_E \sigma_1'(r_1)$, entonces:

$$\sigma_1(r_1) \stackrel{*}{\rightarrow}_E \sigma_1'(r_1) \stackrel{*}{\leftarrow}_E \sigma_1(l_1)[q \leftarrow \sigma_2(r_2)].$$

En lo que sigue, damos las líneas generales de la definición de la función `transform--eq-top-variable-overlap` y de la demostración de los teoremas anteriores, que resuelven el caso de superposición variable. Para obtener más detalles de la misma, instamos al lector a consultar la sección `2.6` del libro `critical-pairs.lisp`.

Reescritura sucesiva en posiciones disjuntas de un término

Supongamos que tenemos un término `term` y una lista `l-pos` de posiciones disjuntas de `term` tal que en todas ocurre el mismo subtérmino. Supongamos además que dicho subtérmino tiene a su vez un subtérmino determinado por una posición `q2` que es instancia del lado izquierdo de una ecuación `l2=r2`, mediante la sustitución `sigma2`. La siguiente función `reduce-list-proof` obtiene la prueba ecuacional correspondiente a aplicar sucesivos reemplazamientos en tal lista de posiciones disjuntas:

```
(defun reduce-list-proof (term l-pos sigma2 l2 r2 q2)
```

```
  (if (endp l-pos)
      nil
    (cons (make-r-step
           :direct t
           :elt1 term
           :elt2 (replace-term term
                               (append (car l-pos) q2)
                               (instance r2 sigma2))
           :operator (make-eq-operator
                      :pos (append (car l-pos) q2)
                      :rule (make-equation l2 r2)
                      :match sigma2))
          (reduce-list-proof (replace-term term
                                           (append (car l-pos) q2)
                                           (instance r2 sigma2))
                             (cdr l-pos)
                             sigma2 l2 r2 q2))))
```

Bajo determinadas condiciones (por ejemplo, que `l-pos` sea un conjunto de posiciones de `term`, que sean todas disjuntas y que en tales posiciones ocurra el mismo subtérmino `u`), esta función obtiene una prueba ecuacional en `E`. Estas condiciones vienen codificadas por el siguiente predicado:

```
(defun condition-reduce-list-proof (term l-pos u sigma2 l2 r2 q2 E)
  (and (member (make-equation l2 r2) E)
       (position-p q2 u)
       (equal (occurrence u q2) (instance l2 sigma2))
       (term-s-p term) (system-s-p E)
       (substitution-s-p sigma2)
       (list-of-positions l-pos term)
       (mutually-disjoint-positions l-pos)
       (same-occurrence term l-pos u)))
```

Omitimos aquí la definición de las funciones `list-of-positions`, `mutually-disjoint-positions` y `same-occurrence` que definen, repectivamente, los conceptos de lista de posiciones de un término, de lista de posiciones disjuntas y de lista de posiciones de un término en las que ocurre el mismo subtérmino.

La prueba que obtiene `reduce-list-proof` conecta `term` con el término resultante de realizar los sucesivos reemplazamientos en los subtérminos que ocurren en las posiciones indicadas por `l-pos`. La función `reduce-list` nos sirve para implementar el concepto de aplicación sucesiva de reemplazamientos en una lista de posiciones de un término:

```
(defun reduce-list (term l-pos v)
  (if (endp l-pos)
      term
    (reduce-list (replace-term term (car l-pos) v)
                 (cdr l-pos) v)))
```

Estamos ya en condiciones de expresar la propiedad fundamental de `reduce-list--proof`, estableciendo que efectivamente obtiene una prueba ecuacional, cuando se verifican las condiciones expresadas por el predicado `condition-reduce-list-proof`:

```
(defthm reduce-list-proof-is-proof
  (implies
   (condition-reduce-list-proof term l-pos u sigma2 l2 r2 q2 E)
   (eq-equiv-s-p
    term
    (reduce-list term l-pos (replace-term u q2 (instance r2 sigma2)))
    (reduce-list-proof term l-pos sigma2 l2 r2 q2) E))
```

Además, la prueba que obtiene `reduce-list-proof` tiene todos sus pasos de prueba en sentido directo:

```
(defthm reduce-list-proof-steps-down
  (steps-down (reduce-list-proof term l-pos sigma2 l2 r2 q2)))
```

Posiciones de una variable en un término

La siguiente función `positions-variable-x` obtiene de manera recursiva, la lista de posiciones de un término (o lista de términos) `term` en las que ocurre la variable `x`:

```
(defun positions-variable-x (flg term x)
  (if flg
      (if (variable-p term)
          (if (equal x term) (list nil) nil)
        (positions-variable-x nil (cdr term) x))
    (if (endp term)
        nil
      (append (map-cons 1 (positions-variable-x t (car term) x))
              (map-plus-car 1 (positions-variable-x nil (cdr term) x))))))
```

Las función `map-cons` añade un elemento dado (usando `cons`) a todas las posiciones de una lista dada. La función `map-plus-car` suma un número dado a cada una de las posiciones de una lista dada. Hemos omitido aquí ambas definiciones.

El siguiente teorema verifica la función `positions-variable-x`, expresando que efectivamente es la lista de posiciones de un término (o lista de términos) en las que ocurre la variable `x`:

```
(defthm positions-variable-x-characterization
  (iff (member pos (positions-variable-x t term x))
       (and (position-p pos term)
            (variable-p (occurrence term pos))
            (equal (occurrence term pos) x))))
```

Como sugiere la prueba a mano y veremos más adelante, la lista de posiciones obtenida mediante la función `positions-variable-x` será utilizada para, mediante la función

`reduce-list-proof`, obtener una prueba ecuacional resultante de aplicar reemplazamiento en tales posiciones. Como se ha visto, para que `reduce-list-proof` produzca una prueba ecuacional, la lista de posiciones con la que se reduce ha de verificar el predicado `condition-reduce-list-proof`. En particular, ha de ser una lista de posiciones disjuntas, en las que ocurre el mismo subtérmino. Los siguientes lemas demuestran tales propiedades para `positions-variable-x`.

```
(defthm position-variable-x-list-of-positions
  (list-of-positions (positions-variable-x t term x) term)

(defthm mutually-disjoint-positions-positions-variable-x
  (mutually-disjoint-positions
   (positions-variable-x flg term x))))

(defthm position-variable-x-same-occurrence-x
  (same-occurrence term (positions-variable-x t term x) x)
```

La primera pieza de la prueba valle

Con lo visto hasta el momento, ya podemos definir la primera pieza con la que formar la prueba valle necesaria para resolver las superposiciones variables. La siguiente función la construye[7]:

```
(defun transform-eq-top-variable-overlap-first-piece
  (peak u1 u2 q1 q2 sigma1 sigma2 l1 r1 l2 r2 x val valr)
  (reduce-list-proof u1 (positions-variable-x t r1 x) sigma2 l2 r2 q2))
```

Según el teorema `reduce-list-proof-is-proof`, esta llamada a `reduce-list-proof` construye una prueba ecuacional en `E`, ya que se hace sobre una lista de posiciones calculada mediante `positions-variable-x`, que como vimos cumple las condiciones definidas por el predicado `condition-reduce-list-proof`[8]. Además, se trata de una prueba en la que todos los pasos se dan en sentido directo.

Recuérdese que aquí `u1` representa el término $\sigma_1(r_1)$. En la notación de la prueba a mano, la función anterior construye la reducción $\sigma_1(r_1) \xrightarrow{*}_E \sigma'_1(r_1)$, reescribiendo con $l_2 \approx r_2$ en los subtérminos que ocurren en las posiciones q' de $\sigma_1(r_1)$ tales que $q' \in \mathcal{P}(r_1)$ y $r_1/q' = x$ y donde σ'_1 es una sustitución igual que σ_1 excepto que $\sigma'_1(x) = \sigma_1(x)[q_2 \leftarrow \sigma_2(r_2)]$.

Para ver que efectivamente la prueba anterior conecta $\sigma_1(r_1)$ con $\sigma'_1(r_1)$, téngase en cuenta que por el teorema `reduce-list-proof-is-proof` aplicado a este caso, la función `reduce-list-proof` construye una prueba ecuacional que conecta el término `u1` con el término `(reduce-list u1 (positions-variable-x t r1 x) valr)`, donde recuérdese que `valr` representa el término $\sigma_1(x)[q_2 \leftarrow \sigma_2(r_2)]$.

Pero `(reduce-list u1 (positions-variable-x t r1 x) valr)` es igual al término `(instance r1 (cons (cons x valr) sigma1))`, lo cual se deduce como caso particular del siguiente teorema:

[7] En esta función y en otras que se verán a continuación, existen varios argumentos irrelevantes. Más adelante se comentará tal circunstancia.

[8] Aplicando además los resultados sobre posiciones e instancias de la sección 3.25, apartado 2.

```
(defthm applying-reductions-to-variable-positions
  (equal (reduce-list (instance term sigma)
                      (positions-variable-x t term x) valr)
         (instance term (cons (cons x valr) sigma)))))
```

Teniendo en cuenta todas las consideraciones anteriores, se deduce finalmente el siguiente teorema, propiedad principal de la función `transform-eq-top-variable-overlap-first-piece`, estableciendo que construye una prueba ecuacional entre $\sigma_1(r_1)$ y $\sigma'_1(r_1)$.

```
(defthm transform-eq-top-variable-overlap-first-piece-is-a-proof
  (implies
   (eq-top-variable-overlap-p
    peak u1 u2 q1 q2 sigma1 sigma2 l1 r1 l2 r2 x val valr E)
   (eq-equiv-s-p
    u1
    (instance r1 (cons (cons x valr) sigma1))
    (transform-eq-top-variable-overlap-first-piece
     peak u1 u2 q1 q2 sigma1 sigma2 l1 r1 l2 r2 x val valr)
    E))
```

Además se trata de una prueba con todos sus pasos en sentido directo:

```
(defthm transform-eq-top-variable-overlap-first-piece-steps-down
  (steps-down
   (transform-eq-top-variable-overlap-first-piece
    peak u1 t2 q1 q2 sigma1 sigma2 l1 r1 l2 r2 x val valr))))
```

La segunda pieza de la prueba valle

La segunda pieza que constituye la prueba valle que estamos buscando la podemos dividir a su vez en dos trozos. El primer trozo de esta segunda pieza queda definido por la siguiente función:

```
(defun transform-eq-top-variable-overlap-second-piece-1
  (peak t1 t2 pos1 q2 sigma1 sigma2 l1 r1 l2 r2 x val valr)
  (list (make-r-step
         :elt1 (instance r1 (cons (cons x valr) sigma1))
         :elt2 (instance l1 (cons (cons x valr) sigma1))
         :direct nil
         :operator (make-eq-operator
                    :pos nil
                    :rule (make-equation l1 r1)
                    :match (cons (cons x valr) sigma1)))))
```

En la notación de la prueba a mano, se trata de la prueba $\sigma'_1(r_1) \leftarrow_E \sigma'_1(l_1)$. Es evidente que esta función construye una prueba, con un único paso en sentido inverso, lo que establecen los siguientes teoremas:

```
(defthm transform-eq-top-variable-overlap-second-piece-1-is-a-proof
  (implies
   (eq-top-variable-overlap-p
    peak u1 u2 q1 q2 sigma1 sigma2 l1 r1 l2 r2 x val valr E)
   (eq-equiv-s-p
    (instance r1 (cons (cons x valr) sigma1))
    (instance l1 (cons (cons x valr) sigma1))
    (transform-eq-top-variable-overlap-second-piece-1
     peak u1 u2 q1 q2 sigma1 sigma2 l1 r1 l2 r2 x val valr)
    E))))

(defthm transform-eq-top-variable-overlap-second-piece-1-steps-up
   (steps-up
    (transform-eq-top-variable-overlap-second-piece-1
     peak u1 t2 q1 q2 sigma1 sigma2 l1 r1 l2 r2 x val valr)))
```

El segundo trozo de esta segunda pieza, lo define la siguiente función:

```
(defun transform-eq-top-variable-overlap-second-piece-2
  (peak u1 u2 q1 q2 sigma1 sigma2 l1 r1 l2 r2 x val valr)
  (inverse-proof
   (reduce-list-proof
    u2
    (delete-one q1 (positions-variable-x t l1 x))
    sigma2 l2 r2 q2)))
```

En la notación de la prueba a mano, esta función construye la prueba inversa de la reducción $\sigma_1(l_1)[q \leftarrow \sigma_2(r_2)] \stackrel{*}{\rightarrow}_E \sigma_1'(l_1)$, siendo $q = q_1 \cdot q_2$ y reescribiendo con $l_2 \approx r_2$ en los subtérminos que ocurren en las posiciones q'' de $\sigma_1(l_1)$ tales que $q'' \in \mathcal{P}(l_1)$ y $l_1/q'' = x$, excepto en q_1. Recuérdese que en este contexto estamos representando por `u2` al término $\sigma_1(l_1)[q_1 \leftarrow \sigma_1(x)[q_2 \leftarrow \sigma_2(r_2)]]$ o lo que es lo mismo, al término $\sigma_1(l_1)[q \leftarrow \sigma_2(r_2)]$.

Por un razonamiento parecido al que nos permitía concluir el teorema principal sobre la primera pieza de la prueba valle que estamos definiendo[9], podemos concluir que la llamada a `reduce-list-proof` que aparece en la función anterior, suponiendo una superposición variable, construye una prueba entre $\sigma_1(l_1)[q \leftarrow \sigma_2(r_2)]$ y $\sigma_1'(l_1)$, con todos los pasos en sentido directo. Esta prueba obtenida mediante `reduce-list-proof` se invierte usando `inverse-proof`, por lo que para concluir las propiedades deseadas necesitamos también dos propiedades fundamentales sobre inversión de pruebas:

- La inversa de una prueba es una prueba (teorema `eq-equiv-s-p-symmetric` de la subsección 7.1.4). Es decir, la relación $\stackrel{*}{\leftrightarrow}_E$ es simétrica.

- La inversa de una prueba con todos sus pasos directos es una prueba con todos sus pasos inversos, como establece el siguiente teorema:

[9] Ahora con la complicación técnica de que en una de las posiciones, la correspondiente a `q1`, no se ha de realizar reemplazamiento. Consúltese la sección `2.6` de `critical-pairs.lisp` para ver cómo se solventa esta dificultad.

```
(defthm steps-up-inverse-proof
  (equal (steps-up (inverse-proof p)) (steps-down p)))
```

Con esta argumentación, se tienen finalmente los siguientes teoremas sobre este segundo trozo, demostrando que se trata de una prueba ecuacional entre $\sigma_1'(l_1)$ y $\sigma_1(l_1)[q \leftarrow \sigma_2(r_2)]$, con todos sus pasos en sentido inverso:

```
(defthm transform-eq-top-variable-overlap-second-piece-2-is-a-proof
  (implies
   (eq-top-variable-overlap-p
    peak u1 u2 q1 q2 sigma1 sigma2 l1 r1 l2 r2 x val valr E)
   (eq-equiv-s-p
    (instance l1 (cons (cons x valr) sigma1))
    u2
    (transform-eq-top-variable-overlap-second-piece-2
     peak u1 u2 q1 q2 sigma1 sigma2 l1 r1 l2 r2 x val valr)
    E))

(defthm transform-eq-top-variable-overlap-second-piece-2-steps-up
  (steps-up
   (transform-eq-top-variable-overlap-second-piece-2
    peak u1 u2 q1 q2 sigma1 sigma2 l1 r1 l2 r2 x val valr)))
```

Concatenando piezas para obtener una prueba valle

Estamos ya en condiciones de construir la prueba valle que estamos buscando, sin más que pegar las piezas correspondientes:

```
(defun transform-eq-top-variable-overlap
  (peak u1 u2 q1 q2 sigma1 sigma2 l1 r1 l2 r2 x val valr)
  (append
   (transform-eq-top-variable-overlap-first-piece
    peak u1 u2 q1 q2 sigma1 sigma2 l1 r1 l2 r2 x val valr)
   (append (transform-eq-top-variable-overlap-second-piece-1
            peak u1 u2 q1 q2 sigma1 sigma2 l1 r1 l2 r2 x val valr)
           (transform-eq-top-variable-overlap-second-piece-2
            peak u1 u2 q1 q2 sigma1 sigma2 l1 r1 l2 r2 x val valr))))
```

Esta prueba se obtiene concatenado pruebas. Necesitamos por tanto una serie de resultados sobre la concatenación de pruebas:

- La concatenación de pruebas con extremos comunes es una prueba (teorema `eq--equiv-s-p-transitive` de la subsección 7.1.4). Es decir, la relación $\overset{*}{\leftrightarrow}_E$ es transitiva.

- La forma de las pruebas concatenadas se rige por los siguientes resultados:

```
(defthm steps-down-append
  (equal (steps-down (append p1 p2))
         (and (steps-down p1) (steps-down p2))))

(defthm steps-up-append
  (equal (steps-up (append p1 p2))
         (and (steps-up p1) (steps-up p2))))


(defthm steps-valley-append
  (implies (and (steps-down p1) (steps-up p2))
           (steps-valley (append p1 p2))))
```

Aplicando estos lemas a la concatenación de pruebas que efectúa la función anterior, se tiene que actuando sobre los elementos que forman una superposición variable, construye una prueba ecuacional que conecta los términos `u1` y `u2` y que tiene forma de valle. Por tanto, con esta definición de `transform-eq-top-variable-overlap` se tienen los dos teoremas `transform-eq-top-variable-overlap-is-a-proof` y `transform--eq-top-variable-overlap-is-a-valley`, presentados anteriormente, resolviendo así el caso correspondiente a las superposiciones variables.

Es interesante destacar que tanto el caso de la reescritura disjunta, como el caso de superposición variable, se resuelven para un sistema de ecuaciones arbitrario `E`, sin hacer mención al sistema `(EKB)`.

7.4.5 Superposición crítica

Se trata ahora de resolver el pico local descrito por la siguiente situación:

$$\sigma_1(r_1) \leftarrow_E \sigma_1(l_1) \rightarrow_E \sigma_1(l_1)[q \leftarrow \sigma_2(r_2)], \textit{ donde } q \in \mathcal{P}(l_1), l_1/q \notin X$$

En primer lugar, definimos la siguiente función `eq-top-local-peak-p`:

```
(defun eq-top-local-peak-p
  (peak u1 u2 q l1 r1 l2 r2 sigma1 sigma2 E)
  (and (term-s-p u1) (term-s-p u2) (term-s-p peak) (system-s-p E)
       (substitution-s-p sigma1) (substitution-s-p sigma2)
       (position-p q peak)
       (member (make-equation l1 r1) E)
       (member (make-equation l2 r2) E)
       (equal peak (instance l1 sigma1))
       (equal (occurrence peak q) (instance l2 sigma2))
       (equal (instance r1 sigma1) u1)
       (equal (replace-term peak q (instance r2 sigma2))
              u2))))
```

Este predicado reconoce situaciones del tipo $\sigma_1(r_1) \leftarrow_E \sigma_1(l_1) \rightarrow_E \sigma_1(l_1)[q \leftarrow \sigma_2(r_2)]$, donde $q \in \mathcal{P}(\sigma_1(l_1))$. Nuevamente `u1` y `u2`, representan respectivamente los términos $\sigma_1(r_1)$ y $\sigma_1(l_1)[q \leftarrow \sigma_2(r_2)]$. Usando este predicado, podemos definir la siguiente función, que describe la situación de superposición crítica:

```
(defun eq-top-critical-overlap-p
  (peak u1 u2 q l1 r1 l2 r2 sigma1 sigma2 E)
  (and (eq-top-local-peak-p
          peak u1 u2 q l1 r1 l2 r2 sigma1 sigma2 E)
       (position-p q l1)
       (not (variable-p (occurrence l1 q)))))
```

Hemos de *definir* una función que llamaremos `transform-eq-top-critical-overlap` y *demostrar* que actuando sobre los elementos que determinan una superposición crítica con respecto a `(EKB)`, obtiene una prueba valle que justifica la equivalencia de los términos `u1` y `u2`. Es decir, hemos de probar los siguientes teoremas:

```
(defthm transform-eq-top-critical-overlap-is-a-proof
  (implies
   (eq-top-critical-overlap-p
    peak u1 u2 q l1 r1 l2 r2 sigma1 sigma2 (EKB))
   (eq-equiv-s-p u1 u2
              (transform-eq-top-critical-overlap
               peak u1 u2 q l1 r1 l2 r2 sigma1 sigma2)
              (EKB))))

(defthm transform-eq-top-critical-overlap-is-a-valley
  (implies
   (eq-top-critical-overlap-p
    peak u1 u2 q l1 r1 l2 r2 sigma1 sigma2 (EKB))
   (steps-valley
    (transform-eq-top-critical-overlap
     peak u1 u2 pos l1 r1 l2 r2 sigma1 sigma2))))
```

Siguiendo la prueba a mano, hemos de probar en primer lugar una formalización del lema 7.24. Es lo que discutimos a continuación.

Una demostración formal del lema 7.24

Seguimos asumiendo la situación descrita por `(eq-top-critical-overlap-p peak u1 u2 q l1 r1 l2 r2 sigma1 sigma2 E)` (es decir, una superposición crítica). Probar el lema 7.24 consiste en:

- demostrar que las ecuaciones determinan un par crítico. Es decir, demostrar que `(cp-r l1 r1 q l2 r2)` es distinto de `nil` y
- encontrar una sustitución que aplicada al par crítico anterior es igual al par `(cons u1 u2)` determinado por la superposición crítica.

Seguiremos las ideas marcadas por la demostración a mano. La siguiente función construye la sustitución $\mu = (\sigma_1\theta_1)_{|V_1} \cup (\sigma_2\theta_2)_{|V_2}$ de la prueba a mano:

```
(defun critical-overlap-unifier
  (peak u1 u2 q l1 r1 l2 r2 sigma1 sigma2)
```

```
(append
 (restriction
  (composition sigma1 (matching-rename-equation l1 r1 0 -1))
  (var-renamed-equation l1 r1 0 -1))
 (restriction
  (composition sigma2 (matching-rename-equation l2 r2 1 1))
  (var-renamed-equation l2 r2 1 1))))
```

Las funciones `matching-rename-equation` y `var-renamed-equation`, que omitimos aquí, devuelven la sustitución que aplicada a una ecuación renombrada obtiene la ecuación original (sustituciones θ_1 y θ_2) y el conjunto de variables de una ecuación renombrada (conjuntos V_1 y V_2).

No es difícil ver que `critical-overlap-unifier` proporciona un unificador de l_1'/q y de l_2', donde l_1' y l_2', son los renombrados respectivos de l_1 y l_2. Por tanto, según el teorema `mgu-completeness`, el algoritmo de unificación `mgu-mv` actuando sobre ambos términos no falla, condición suficiente para que `cp-r` tampoco falle[10]:

```
(defthm critical-overlap-implies-critical-pair
  (implies (and (position-p q l1)
                (not (variable-p (occurrence l1 q)))
                (equal (occurrence (instance l1 sigma1) q)
                       (instance l2 sigma2)))
           (cp-r l1 r1 q l2 r2)))
```

Como l_1'/q y l_2' son unificables, un unificador de máxima generalidad de ambos subsume a cualquier otro unificador (teorema `mgu-most-general-unifier`, sección 4.4). Existirá por tanto una sustitución que compuesta con tal unificador de máxima generalidad sea igual al unificador construido por la función `critical-overlap-unifier`. Esta es la sustitución que da justamente la instanciación que buscamos y está definida por la siguiente función `critical-overlap-instace`:

```
(defun critical-overlap-instance
  (peak u1 u2 q l1 r1 l2 r2 sigma1 sigma2)
  (matching-subst-r
   (mgu (occurrence (lhs-r l1 r1 0 -1) q)
        (lhs-r l2 r2 1 1))
   (critical-overlap-unifier
    peak u1 u2 q l1 r1 l2 r2 sigma1 sigma2)))
```

Nótese el uso de la función `matching-subst-r` definida en la sección 3.4, que devolvía la sustitución testigo de la subsunción de dos sustituciones (subsección 3.4.3).

Con esta definición podemos finalmente establecer el lema 7.24, mediante el siguiente teorema:

[10] Las hipótesis de este teorema no usan explícitamente `eq-top-critical-overlap-p`, aunque sí una condición equivalente. La razón es que así se evita la aparición de variables libres que entorpecerían la reescritura en el demostrador automático.

```
(defthm critical-overlap-instance-of-cp
  (let* ((cp-r (cp-r l1 r1 q l2 r2))
         (delta (critical-overlap-instance
                 peak u1 u2 q l1 r1 l2 r2 sigma1 sigma2)))
    (implies
     (eq-top-critical-overlap-p
      peak u1 u2 q l1 r1 l2 r2 sigma1 sigma2 E)
     (and
      (equal (instance (lhs cp-r) delta) u1)
      (equal (instance (rhs cp-r) delta) u2)))))
```

Resolviendo las superposiciones críticas

Para definir ahora la función `transform-eq-top-critical-overlap` que obtiene una prueba valle para cada par de términos `u1` y `u2` determinados por una superposición crítica entre ecuaciones del sistema `(EKB)`, basta tener en cuenta las siguientes observaciones:

- Hemos asumido que la función `transform-critical-pair` obtiene una prueba valle para cada par crítico de `(EKB)`.
- Todo par de términos determinados por una superposición crítica entre ecuaciones del sistema `(EKB)` es instancia, mediante la sustitución definida por la función `critical-overlap-instance`, de un par crítico.

Por tanto, bastará instanciar convenientemente la prueba devuelta `transform-critical-pair`, usando para ello la función `eq-proof-instance`, definida en la subsección 7.1.3

```
(defun transform-eq-top-critical-overlap
  (peak u1 u2 q l1 r1 l2 r2 sigma1 sigma2)
  (eq-proof-instance
   (transform-critical-pair l1 r1 q l2 r2)
   (critical-overlap-instance
    peak u1 u2 q l1 r1 l2 r2 sigma1 sigma2)))
```

La prueba que se obtiene es instancia de una prueba valle (la que existe para cada par crítico). Para probar que esta instancia también es una prueba valle, se necesitan dos propiedades sobre la instanciación de pruebas:

- Instanciando una prueba se obtiene una prueba. Es el teorema `eq-equiv-s-p--stable` de la subsección 7.1.4. Es decir, usamos la propiedad de estabilidad de la relación $\overset{*}{\leftrightarrow}_E$.
- Instanciando una prueba en forma de valle se obtiene una prueba en forma de valle, como afirma el siguiente teorema:

```
(defthm eq-proof-instance-valley
  (implies (steps-valley proof)
           (steps-valley (eq-proof-instance proof sigma))))
```

Las consideraciones anteriores permiten demostrar fácilmente que con esta definición de `transform-eq-top-critical-overlap` se tiene los dos teoremas `transform--eq-top-critical-overlap-is-a-proof` y `transform-eq-top-critical-overlap-is--a-valley`, presentados anteriormente, resolviendo así el caso correspondiente a las superposiciones críticas.

7.4.6 La función de transformación de picos locales

Recuérdese que nuestro objetivo final, para demostrar formalmente el teorema de pares críticos de Knuth y Bendix, es definir una función `transform-eq-local-peak` y demostrar que para cada prueba ecuacional `p` en `(EKB)` que sea un pico local, `(transform-eq-local--peak p)` es una prueba valle equivalente. La resolución de las tres situaciones estudias previamente (reescritura disjunta, superposición variable y superposición crítica) nos va a permitir solventar cualquier tipo de pico local ecuacional que se produzca. En lo que sigue, analizamos todas la casuística existente y veremos que cada una de las posibilidades se reducen a una de las tres situaciones estudiadas. Por tanto, con una adecuada distinción de casos, podremos finalmente definir `transform-eq-local-peak` con las propiedades deseadas.

Picos locales con una reescritura en posición raíz

Estudiamos en primer lugar la situación en la que una de las dos reescrituras que ocurren en el pico local tiene como posición redex la posición raíz del término. Es decir, nos encontramos en la siguiente situación:

$$\sigma_1(r_1) \leftarrow_E \sigma_1(l_1) \rightarrow_E \sigma_1(l_1)[q \leftarrow \sigma_2(r_2)], \textit{ donde } q \in \mathcal{P}(\sigma_1(l_1))$$

Denominaremos a esta situación un **pico local raíz**. La función `eq-top-local--peak-p`, definida en la subsección anterior al estudiar las superposiciones críticas, define formalmente los picos locales raíz. Nuestro objetivo ahora es *definir* una función que llamaremos `transform-eq-top-local-peak` y *demostrar* que actuando sobre los elementos que determinan un pico local raíz con respecto a `(EKB)`, obtiene una prueba valle equivalente. Es decir, hemos de probar los siguientes teoremas:

```
(defthm transform-eq-top-local-peak-is-a-proof
  (implies
   (eq-top-local-peak-p
    peak u1 u2 q l1 r1 l2 r2 sigma1 sigma2 (EKB))
   (eq-equiv-s-p
    u1 u2
    (transform-eq-top-local-peak
     peak u1 u2 q l1 r1 l2 r2 sigma1 sigma2)
    (EKB)))))

(defthm transform-eq-top-local-peak-is-a-valley
  (implies
   (eq-top-local-peak-p
    peak u1 u2 q l1 r1 l2 r2 sigma1 sigma2 (EKB))
```

```
(steps-valley
 (transform-eq-top-local-peak
 peak u1 u2 q l1 r1 l2 r2 sigma1 sigma2)))))
```

Para resolver los picos locales raíz, nuevamente seguimos la demostración hecha a mano. En la prueba a mano, el estudio de un pico local raíz queda reflejado en los siguientes párrafos:

> *Como* $q \in \mathcal{P}(\sigma_1(l_1))$*, se tiene que o bien* $q \in \mathcal{P}(l_1)$ *y* $l_1/q \notin X$*, o bien* $q = q_1 \cdot q_2$ *con* $q_1 \in \mathcal{P}(l_1), l_1/q_1 = x \in X, q_2 \in \mathcal{P}(\sigma_1(x))$ *y* $\sigma_1(x)/q_2 = \sigma_2(l_2)$*. Por tanto existen dos posibles subcasos:*
>
> *Subcaso 2a* $q = q_1 \cdot q_2$ *con* $q_1 \in \mathcal{P}(l_1), l_1/q_1 = x \in X, q_2 \in \mathcal{P}(\sigma_1(x))$ *y* $\sigma_1(x)/q_2 = \sigma_2(l_2) \ldots\ldots$
>
> *Subcaso 2b* $q \in \mathcal{P}(l_1)$ *y* $l_1/q \notin X \ldots\ldots$

Es decir, actuaremos dependiendo de la posición $q \in \sigma_1(l_1)$. Existen tres posibilidades:

- $q \in l_1$ y $l_1/q \notin X$.
- $q \in l_1$ y $l_1/q \in X$.
- $q \notin l_1$.

La primera de las situaciones ya la hemos resuelto: se trata justamente de una superposición crítica y se corresponde con el subcaso 2b de la demostración a mano. Las dos restantes situaciones se corresponden con el subcaso 2a. Ambas situaciones describen una superposición variable. Siguiendo la idea de la demostración a mano, basta dividir la posición q en dos trozos q_1 y q_2, con $q_1 \in \mathcal{P}(l1)$, $l_1/q = x \in X$ y $q_2 \in \mathcal{P}(\sigma_1(x))$. La división de q en estas dos posiciones las dan respectivamente las dos funciones siguientes:

```
(defun position-portion-inside (pos l)
  (if (or (endp pos) (variable-p l))
      nil
    (cons (car pos)
          (position-portion-inside
           (cdr pos)
           (nth (- (car pos) 1) (cdr l))))))

(defun position-portion-outside (pos l)
  (if (or (endp pos) (variable-p l))
      pos
    (position-portion-outside
     (cdr pos)
     (nth (- (car pos) 1) (cdr l)))))
```

Una vez definidas estas dos funciones, podemos establecer el siguiente resultado, que muestra efectivamente que estamos hablando de una superposición variable (recuérdese que las superposiciones variables las describimos formalmente por el predicado `eq-top--variable-overlap-p`):

```
(defthm eq-top-local-peak-p-variable-overlap-reduction-theorem
  (let* ((q1 (position-portion-inside q l1))
         (q2 (position-portion-outside q l1))
         (x (occurrence l1 q1))
         (val (val x sigma1))
         (valr (replace-term val q2 (instance r2 sigma2))))
    (implies (and (eq-top-local-peak-p
                      peak u1 u2 q l1 r1 l2 r2 sigma1 sigma2 E)
                  (or (not (position-p q l1))
                      (and (position-p q l1)
                           (variable-p (occurrence l1 q)))))
             (eq-top-variable-overlap-p
               peak u1 u2 q1 q2 sigma1 sigma2 l1 r1 l2 r2 x val valr E)))
```

Ese teorema nos permite reducir cierto tipo de pico local raíz a la situación de superposición variable. Más concretamente, nos permite obtener los elementos que describen la superposición variable, a partir de los elementos que describen el pico local raíz. Como tenemos resueltas las superposiciones variables (función `transform-eq-top-variable-overlap`, subsección 7.4.4), este teorema nos permite resolver estos picos locales raíz.

Todas estas consideraciones nos llevan a la siguiente definición de la función `transform-eq-top-local-peak`. Basta con distinguir si el pico local raíz da lugar a una superposición crítica (en cuyo caso se llama a la función `transform-eq-top-critical-overlap`) o si da lugar a una superposición variable (en cuyo caso se llama a la función `transform-eq-top-variable-overlap`):

```
(defun transform-eq-top-local-peak
  (peak u1 u2 q l1 r1 l2 r2 sigma1 sigma2)
  (if (and (position-p q l1)
           (not (variable-p (occurrence l1 q))))
      (transform-eq-top-critical-overlap
       peak u1 u2 q l1 r1 l2 r2 sigma1 sigma2)
    (let* ((q1 (position-portion-inside q l1))
           (q2 (position-portion-outside q l1))
           (x (occurrence l1 q1))
           (val (val x sigma1))
           (valr (replace-term val q2 (instance r2 sigma2))))
      (transform-eq-top-variable-overlap
       peak u1 u2 q1 q2 sigma1 sigma2 l1 r1 l2 r2 x val valr))))
```

Teniendo en cuenta las propiedades de las funciones que resuelven las superposiciones críticas y las superposiciones variables, la demostración de los teoremas `transform-eq-top-local-peak-is-a-proof` y `transform-eq-top-local-peak-is-a-valley` presentados anteriormente, se tiene de manera inmediata, con lo que queda resuelto el caso correspondiente a un pico local raíz.

Pico local en posiciones prefijas

Supongamos que tenemos un pico ecuacional local en la que la primera de las reescrituras se produce en una posición prefija de la posición correspondiente a la segunda reescritura.

Es decir, nos encontramos en la siguiente situación:

$$u[p_1 \leftarrow \sigma_1(r_1)] \leftarrow_E u \rightarrow_E u[p_2 \leftarrow \sigma_2(r_2)], \textit{ donde } p_1 \leq p_2$$

Es lo que llamamos un **pico local prefijo**. Para describir formalmente los picos locales prefijos respecto a `(EKB)`, basta con suponer `(eq-local-peak-p peak t1 t2 p1 p2 l1 r1 l2 r2 sigma1 sigma2 (EKB))` junto con `(prefix p1 p2)`. El predicado `eq--local-peak-p` fue definido en la subsección 7.4.3 al hablar de reescritura en posiciones disjuntas y `prefix` (cuya definición omitimos) es un predicado que comprueba si su primer argumento es prefijo del segundo.

Nuevamente, para resolver este caso, debemos *definir* una función que llamaremos `transform-eq-prefix-peak` y *demostrar* que actuando sobre los elementos que determinan un pico local prefijo respecto a `(EKB)`, obtiene una prueba valle que justifica la equivalencia de los términos `t1` y `t2` en la teoría ecuacional de `(EKB)`. Es decir, hemos de probar los siguientes teoremas:

```
(defthm transform-eq-prefix-peak-is-a-proof
  (implies
   (and
    (eq-local-peak-p peak t1 t2 p1 p2 l1 r1 l2 r2 sigma1 sigma2 (EKB))
    (prefix p1 p2))
   (eq-equiv-s-p
    t1 t2
    (transform-eq-prefix-peak
     peak t1 t2 p1 p2 l1 r1 l2 r2 sigma1 sigma2)
    (EKB))))

(defthm transform-eq-prefix-peak-is-a-valley
  (implies
   (and
    (eq-local-peak-p peak t1 t2 p1 p2 l1 r1 l2 r2 sigma1 sigma2 (EKB))
    (prefix p1 p2))
   (steps-valley
    (transform-eq-prefix-peak
     peak t1 t2 p1 p2 l1 r1 l2 r2 sigma1 sigma2))))
```

Para definir la función `transform-eq-prefix-peak`, volvamos a analizar la prueba a mano. En concreto, el siguiente párrafo:

> *Sea q tal que $p_2 = p_1 \cdot q$. Por el teorema 3.25 (apartado 3) y por ser $p_2 = p_1 \cdot q \in \mathcal{P}(u)$, se tiene que $q \in \mathcal{P}(u/p_1) = \mathcal{P}(\sigma_1(l_1))$ y $\sigma_1(l_1)/q = \sigma_2(l_2)$. Entonces $s = u[p_1 \leftarrow \sigma_1(r_1)]$ y*
>
> $$t = u[p_2 \leftarrow \sigma_2(r_2)] = u[p_1 \leftarrow \sigma_1(l_1)][p_2 \leftarrow \sigma_2(r_2)] =$$
>
> $$= u[p_1 \leftarrow \sigma_1(l_1)][p_1 q \leftarrow \sigma_2(r_2)] = u[p_1 \leftarrow \sigma_1(l_1)[q \leftarrow \sigma_2(r_2)]]$$

(esta última igualdad nuevamente por el teorema 3.25, apartado 3). Si probamos que

$$\sigma_1(r_1) \downarrow_E \sigma_1(l_1)[q \leftarrow \sigma_2(r_2)] \text{ ó}$$

$$\sigma_1(r_1) \longleftrightarrow_{pc(E)} \sigma_1(l_1)[q \leftarrow \sigma_2(r_2)],$$

entonces por la compatibilidad de $\rightarrow_E$ *y de* $\longleftrightarrow_{pc(E)}$ *se tendría que* $s \downarrow_E t$ *ó* $s \longleftrightarrow_{pc(E)} t$.

Es decir, este caso se puede resolver si somos capaces de resolver un pico local raíz: justamente lo que se ha estudiado en el apartado anterior. El siguiente teorema formaliza esta idea, reduciendo una situación de pico local prefijo a una situación de pico local raíz:

```
(defthm eq-local-peak-p-eq-top-local-peak-p-reduction-theorem
  (implies
   (and (eq-local-peak-p peak t1 t2 p1 p2 l1 r1 l2 r2 sigma1 sigma2 E)
        (prefix p1 p2))
   (eq-top-local-peak-p
     (occurrence peak p1) (occurrence t1 p1) (occurrence t2 p1)
     (difference-pos p1 p2) l1 r1 l2 r2 sigma1 sigma2 E))
```

Más concretamente, este teorema nos da explícitamente los elementos del pico local raíz al que se reduce un pico local prefijo: si tenemos un pico local prefijo con los términos `peak`, `t1`, `t2` y en las posiciones `q1` y `q2`, eso da lugar al pico local raíz con los términos `(occurrence peak p1)`, `(occurrence t1 p1)` y `(occurrence t2 p1)` y en la posición `(difference-pos p1 p2)`.

Podemos ahora fácilmente resolver un pico local prefijo: obtenemos el pico local raíz al que se reduce, calculamos la prueba valle equivalente (con la función `transform-eq--top-local-peak`) e incluimos dicha prueba (usando `eq-proof-context`, definida en la sección 7.1) en el contexto determinado por `peak` y `p1`. Esto es lo que hace la función `transform-eq-prefix-peak`:

```
(defun transform-eq-prefix-peak
  (peak t1 t2 p1 p2 l1 r1 l2 r2 sigma1 sigma2)
  (eq-proof-context
   (transform-eq-top-local-peak
    (occurrence peak p1) (occurrence t1 p1) (occurrence t2 p1)
    (difference-pos p1 p2) l1 r1 l2 r2 sigma1 sigma2)
   peak p1))
```

Puesto que la prueba que construye esta función se obtiene en último término como inclusión de otra prueba en un contexto, para demostrar sus propiedades necesitamos las siguientes propiedades de la función `eq-proof-context`:

- La inclusión de una prueba en un contexto es una prueba. Es el teorema `eq-equiv--s-p-compatible` de la subsección 7.1.4. Es decir, usamos la propiedad de compatibilidad de la relación $\overset{*}{\leftrightarrow}_E$.
- La inclusión de una prueba en forma de valle en un contexto es un prueba en forma de valle, como afirma el siguiente teorema:

```
(defthm eq-proof-context-valley
  (implies (steps-valley proof)
           (steps-valley (eq-proof-context proof term pos)))
```

Todas estas argumentaciones nos permiten concluir los teoremas `transform-eq-prefix-peak-is-a-proof` y `transform-eq-prefix-peak-is-a-valley` presentados anteriormente, resolviendo así las situación correspondiente a picos locales prefijos.

Una caso simétrico del anterior

Para completar el análisis de la casuística existente en la situaciones de picos locales ecuacionales, debemos ahora resolver la situación en la que la segunda de las reescrituras se produce en una posición prefija de la posición correspondiente a la primera reescritura. Es decir, nos encontramos en la siguiente situación:

$$u[p_1 \leftarrow \sigma_1(r_1)] \leftarrow_E u \rightarrow_E u[p_2 \leftarrow \sigma_2(r_2)], \textit{ donde } p_2 \leq p_1$$

Se trata, obviamente de una situación simétrica a la del pico local prefijo estudiada en el apartado anterior. Para describir formalmente estos picos locales prefijos, basta con suponer `(eq-local-peak-p peak t1 t2 p1 p2 l1 r1 l2 r2 sigma1 sigma2 E)` junto con `(prefix p2 p1)`.

Para resolver este caso, debemos *definir* una función que llamaremos `transform--eq-prefix-peak-symmetric--case` y *demostrar* que actuando sobre los elementos que determinan un pico local prefijo respecto a `(EKB)`, obtiene una prueba valle que justifica la equivalencia de los términos `t1` y `t2` en la teoría ecuacional de `(EKB)`. Es decir, hemos de probar los siguientes teoremas:

```
(defthm transform-eq-prefix-peak-symmetric-case-is-a-proof
  (implies
   (and
    (eq-local-peak-p peak t1 t2 p1 p2 l1 r1 l2 r2 sigma1 sigma2 (EKB))
    (prefix p2 p1))
   (eq-equiv-s-p
    t1 t2
    (transform-eq-prefix-peak-symmetric-case
     peak t1 t2 p1 p2 l1 r1 l2 r2 sigma1 sigma2)
    (EKB))))

(defthm transform-eq-prefix-peak-symmetric-case-is-a-valley
  (implies
   (and
    (eq-local-peak-p peak t1 t2 p1 p2 l1 r1 l2 r2 sigma1 sigma2 (EKB))
    (prefix p2 p1))
   (steps-valley
    (transform-eq-prefix-peak-symmetric-case
     peak t1 t2 p1 p2 l1 r1 l2 r2 sigma1 sigma2))))
```

En la prueba a mano, esta situación se corresponde con el caso 3. Recuérdese el párrafo correspondiente:

Caso 3 $p_2 \leq p_1$. *El razonamiento es totalmente simétrico al caso 2, cambiando los papeles de* s *y* t *y teniendo en cuenta que las relaciones* $\overset{*}{\leftrightarrow}_E$ *y* $\longleftrightarrow_{pc(E)}$ *son simétricas.*

Siguiendo esta idea, para resolver este caso simplemente llamamos a la función `transform-eq-prefix-peak`, intercambiando los papeles de los términos `t1` y `t2`. Es decir, la aplicamos a los elementos del pico local $u[p_2 \leftarrow \sigma_2(r_2)] \leftarrow_E u \rightarrow_E u[p_1 \leftarrow \sigma_1(r_1)]$, obteniendo una prueba valle que deberá ser invertida para que sea una prueba equivalente al pico local original. Es lo que hace la siguiente definición:

```
(defun transform-eq-prefix-peak-symmetric-case
  (peak t1 t2 p1 p2 l1 r1 l2 r2 sigma1 sigma2)
  (inverse-proof
   (transform-eq-prefix-peak
    peak t2 t1 p2 p1 l2 r2 l1 r1 sigma2 sigma1)))
```

Para concluir las propiedades deseadas sobre esta función, usamos nuevamente las siguientes propiedades sobre inversión de pruebas:

- La inversa de una prueba es una prueba (teorema `eq-equiv-s-p-symmetric` de la subsección 7.1.4). Es decir, la relación $\overset{*}{\leftrightarrow}_E$ es simétrica.
- La inversa de una prueba valle es una prueba valle, como establece el siguiente teorema:

```
(defthm steps-valley-inverse-proof
  (implies (steps-valley p)
           (steps-valley (inverse-proof p))))
```

Con estas propiedades, junto con las propiedades ya vistas en el apartado anterior sobre la función `transform-eq-prefix-peak`, obtenemos los teoremas presentados anteriormente sobre `transform-eq-prefix-peak-symmetric-case`, resolviendo así la situación simétrica de un pico local.

Finalmente, la función `transform-eq-local-peak`

Llegado este punto, hemos resuelto los picos locales correspondientes a posiciones p_1 y p_2 tales que $p_1|p_2$, o $p_1 \leq p_2$, o bien $p_1 \leq p_2$. El siguiente teorema muestra que esos tres casos cubren todas las posibilidades:

```
(defthm prefix-disjoint
  (implies (and (not (prefix pos1 pos2))
                (not (prefix pos2 pos1)))
           (disjoint-positions pos1 pos2))))
```

Por tanto, ya podemos definir una función, que actuando sobre los elementos que forman un pico local, construye una prueba valle equivalente. Recuérdese que en nuestra formalización, la situación de pico local ecuacional se describe asumiendo que `(eq-local-peak-p peak t1 t2 p1 p2 l1 r1 l2 r2 sigma1 sigma2 (EKB))` es cierto.

Por una simple distinción de casos, podemos definir una función `transform-eq-local--peak-aux`, que actuando sobre los elementos de tal pico local, obtiene una prueba ecuacional en forma de valle que conecta `t1` y `t2`:

```
(defun transform-eq-local-peak-aux
  (peak t1 t2 p1 p2 l1 r1 l2 r2 sigma1 sigma2)
  (cond ((prefix p1 p2)
         (transform-eq-prefix-peak
          peak t1 t2 p1 p2 l1 r1 l2 r2 sigma1 sigma2))
        ((prefix p2 p1)
         (transform-eq-prefix-peak-symmetric-case
          peak t1 t2 p1 p2 l1 r1 l2 r2 sigma1 sigma2))
        (t (transform-disjoint-eq-local-peak
            peak t1 t2 p1 p2 l1 r1 l2 r2 sigma1 sigma2))))
```

Sólo queda ya un detalle final: téngase en cuenta que la función que pretendemos definir, `transform-eq-local-peak`, tiene un único argumento de entrada, ya que debe actuar sobre pruebas ecuacionales y no sobre sus elementos individuales. El siguiente teorema tiende un puente entre una prueba ecuacional con forma de pico local y sus componentes:

```
(defthm local-peak-p-eq-local-peak-p-reduction-theorem
  (let* ((step1 (first p))
         (step2 (second p))
         (peak (elt2 step1))
         (operator1 (operator step1))
         (operator2 (operator step2))
         (rule1 (op-rule operator1))
         (rule2 (op-rule operator2))
         (p1 (op-pos operator1))
         (p2 (op-pos operator2))
         (l1 (lhs rule1))
         (l2 (lhs rule2))
         (r1 (rhs rule1))
         (r2 (rhs rule2))
         (sigma1 (op-match operator1))
         (sigma2 (op-match operator2)))
    (implies (and (system-s-p E)
                  (eq-equiv-s-p t1 t2 p E)
                  (local-peak-p p))
             (eq-local-peak-p
               peak t1 t2 p1 p2 l1 r1 l2 r2 sigma1 sigma2 E)))
```

Ya podemos definir cómo actúa la función `transform-eq-local-peak` sobre un pico local ecuacional `p`. Basta llamar a la función `transform-eq-local-peak-aux` sobre los elementos individuales de `p`:

```
(defun transform-eq-local-peak (p)
```

```
  (let* ((step1 (first p))
         (step2 (second p))
         (peak (elt2 (first p)))
         (operator1 (operator step1))
         (operator2 (operator step2))
         (rule1 (op-rule operator1))
         (rule2 (op-rule operator2))
         (pos1 (op-pos operator1))
         (pos2 (op-pos operator2))
         (l1 (lhs rule1))
         (l2 (lhs rule2))
         (r1 (rhs rule1))
         (r2 (rhs rule2))
         (sigma1 (op-match operator1))
         (sigma2 (op-match operator2))
         (t1 (elt1 step1))
         (t2 (elt2 step2)))
    (transform-eq-local-peak-aux
     peak t1 t2 pos1 pos2 l1 r1 l2 r2 sigma1 sigma2)))
```

Usando este teorema y las propiedades ya vistas sobre las funciones que resuelven los picos locales disjuntos, prefijos y simétrico, se tienen de manera inmediata los siguientes teoremas, que muestran que cuando `p` es una prueba ecuacional en `(EKB)` con forma de pico local, `(transform-eq-local-peak p)` es una prueba valle equivalente:

```
(defthm transform-eq-local-peak-is-a-proof
  (implies
   (and
    (eq-equiv-s-p t1 t2 p (EKB))
    (local-peak-p p))
   (eq-equiv-s-p t1 t2 (transform-eq-local-peak p) (EKB))))

(defthm transform-eq-local-peak-is-a-valley
  (implies
   (and
    (eq-equiv-s-p t1 t2 p (EKB))
    (local-peak-p p))
   (steps-valley (transform-eq-local-peak p))))
```

Estos dos teoremas implican trivialmente el teorema `knuth-bendix-theorem`, demostrando así el teorema de pares críticos de Knuth y Bendix.

Las funciones del álgebra de pruebas

Antes de finalizar la descripción de la demostración formal del teorema de pares críticos, merece la pena destacar un aspecto interesante de la misma. Nótese que en la prueba a mano se hace uso en varias ocasiones (tanto implícita como explícitamente) de las propiedades que hacen que $\overset{*}{\leftrightarrow}_E$ sea una congruencia: simetría, transitividad, estabilidad,

compatibilidad, etc. Como ya vimos en la subsección 7.1.4, cada una de estas propiedades se formalizan en nuestra teoría mediante una operación del álgebra de pruebas y su propiedad correspondiente. Así, el uso de estas propiedades en la prueba formal se corresponde con el uso de la correspondiente operación de construcción de pruebas. Por ejemplo:

- En las superposiciones variables, el uso de la transitividad se traduce en la concatenación, usando `append`, de los tres trozos de prueba que finalmente forman la prueba valle.
- En las superposiciones críticas, la propiedad de estabilidad está usándose a través de la función `eq-proof-instance`, que obtiene la prueba valle a partir de la prueba entre pares críticos.
- En los picos locales prefijos, se usa compatibilidad a través de la función `eq-proof--context`, usada para obtener la prueba valle a partir de la obtenida para picos locales raíz.
- En el caso simétrico a los picos locales prefijos, la simetría se traduce en el uso de `inverse-proof` para obtener la inversa de la prueba obtenida para picos locales prefijos.

Como en la subsección 7.1.4, insistimos nuevamente en la ventaja que para la demostración automática (en este caso del teorema de pares críticos) supone el tratar las pruebas ecuacionales como objetos susceptibles de ser transformados y el que las propiedades básicas de las teorías ecuacionales se expresen a partir de estas operaciones.

Con esto finalizamos la descripción de la demostración formal en ACL2 del teorema de pares críticos. En el apéndice C, subsección C.5.3, damos alguna información adicional sobre la demostración automática y la interacción con el demostrador.

7.5 Cálculo de pares críticos

Como hemos visto en la sección anterior, el teorema de pares críticos permite estudiar la confluencia local de la reducción ecuacional asociada a un sistema de ecuaciones, comprobando la convergencia de un número finito de pares de términos: los pares críticos del sistema. En esta sección definimos y verificamos una función que calcula todos los pares críticos de un sistema de ecuaciones dado. Los eventos presentados en esta sección se encuentran también en el libro `critical-pairs.lisp`.

7.5.1 Definición y verificación

En la figura 7.2 mostramos la definición de la función `cps-trs`, que calcula todos los pares críticos determinados por la superposición entre ecuaciones de un sistema que recibe como entrada.

Explicamos a continuación brevemente cada una de las funciones auxiliares que se necesitan para la definición de `cps-trs`:

- `(cps-rule-vs-rule l1 r1 l2 r2)` calcula todos los pares críticos determinados por la superposición de la ecuación `l2=r2` en una posición no variable del lado izquierdo de la ecuación `l1=r1` (renombrando previamente las ecuaciones para separar sus variables).

```
(defun cps-rule-vs-rule-aux (l1 r1 flg posr arg term l2 r2)
 (if flg
     (if (variable-p term)
         nil
       (mv-let
        (mgu unifiable)
        (mgu-mv term l2)
        (let ((cp-top
               (if unifiable
                  (list (cons (instance r1 mgu)
                              (instance
                               (replace-term l1 (reverse posr) r2)
                               mgu)))
                  nil)))
          (append cp-top
                  (cps-rule-vs-rule-aux
                   l1 r1 nil posr 0 (cdr term) l2 r2)))))
   (if (endp term) nil
     (append (cps-rule-vs-rule-aux
              l1 r1 t (cons (1+ arg) posr) 0 (car term) l2 r2)
             (cps-rule-vs-rule-aux
              l1 r1 nil posr (1+ arg) (cdr term) l2 r2)))))

(defun cps-rule-vs-rule (l1 r1 l2 r2)
  (let* ((rule1-r (number-rename-equation l1 r1 0 -1))
         (rule2-r (number-rename-equation l2 r2 1 1))
         (l1-r (nth 0 rule1-r)) (r1-r (nth 1 rule1-r))
         (l2-r (nth 0 rule2-r)) (r2-r (nth 1 rule2-r)))
    (cps-rule-vs-rule-aux l1-r r1-r t nil 0 l1-r l2-r r2-r)))

(defun cps-rule-vs-trs (l1 r1 R)
  (if (endp R)
      nil
    (append (cps-rule-vs-rule l1 r1 (lhs (car R)) (rhs (car R)))
            (cps-rule-vs-trs l1 r1 (cdr R)))))

(defun cps-trs-vs-trs (R1 R2)
  (if (endp R1)
      nil
    (append (cps-rule-vs-trs (lhs (car R1)) (rhs (car R1)) R2)
            (cps-trs-vs-trs (cdr R1) R2))))

(defun cps-trs (R) (cps-trs-vs-trs R R))
```

Figura 7.2: Cálculo de pares críticos

- `(cps-rule-vs-trs l1 r1 R)` calcula todos los pares críticos determinados por la superposición de una ecuación de `R` en una posición no variable del lado izquierdo de la ecuación `l1=r1`.
- `(cps-trs-vs-trs R1 R2)` calcula todos los pares críticos determinados por la superposición de una ecuación de `R2` en una posición no variable del lado izquierdo de una ecuación de `R1`.
- Finalmente `(cps-trs R)` calcula todos los pares críticos obtenidos por superposición de una ecuación de `R` en una posición no variable del lado izquierdo de una regla de `R`. Es decir, todos los pares críticos determinados del sistema de ecuaciones `R`.

Como se observa en las definiciones, el proceso principal de `(cps-rule-vs-rule l1 r1 l2 r2)` lo realiza la función `cps-rule-vs-rule-aux`, que implementa el proceso recursivo que permite calcular todos los pares críticos determinados por la superposición de una ecuación `l2=r2` en el lado izquierdo de otra ecuación `l1=r1`, mediante un recorrido por la estructura de `l1` buscando posiciones en las que ocurra un subtérmino no variable que unifique con `l2`. En concreto, esta función tiene como argumentos (además de los términos `l1`, `r1`, `l2` y `r2` que determinan las dos ecuaciones) a `flg`, `term`, `arg` y `posr`. Estos permiten saber en cada momento qué subtérmino de `l1` se está analizando. En concreto:

- Si `flg`$\neq$`nil`, entonces `term` es el subtérmino que ocurre en la posición `pos` obtenida como lista inversa de `posr`. En este caso, `arg` es irrelevante.
- Si `flg=nil`, `term` es la lista de términos obtenida quitando los primeros `arg` argumentos de la lista de argumentos del subtérmino que ocurre en la posición `pos`, lista inversa de `posr`.

Es decir: en cada momento, `posr` (su inversa en realidad) es una posición que apunta al término que se está analizando. Si `flg` es `nil`, `arg` además indica el número de argumento de dicho subtérmino que en ese momento se analiza.

La función `cps-rule-vs-rule` se define simplemente llamando inicialmente a la función `cps-rule-vs-rule-aux` para `flg=t`, `posr=nil`, `arg=0` y los términos que forman las ecuaciones de entrada (una vez se han renombrado las mismas para separar sus variables).

Lo que sigue son varios ejemplos de cálculo, usando `cps-trs` de los pares críticos de un sistema de ecuaciones:

```
ACL2 !>(cps-trs '( ((* (* x y) z) . (* x (* y z)))
                   ((* (i x) x) . (e))
                   ((* (e) x) . x)) )

(((* 1 (* 2 3)) . (* 1 (* 2 3)))
 ((* (* 1 2) (* 3 -2)) . (* (* 1 (* 2 3)) -2))
 ((* (I 1) (* 1 -2)) . (* (E) -2))
 ((* (E) (* 1 -2)) . (* 1 -2))
 ((E) . (E))
 (1 . 1))
```

```
ACL2 !>(cps-trs '( ((* (i x) (* x y)) . y)))

((2 . 2)
 ((* 1 2) . (* (I (I 1)) 2)))

ACL2 !>(cps-trs '( ((* (* a b) (* b c)) . b)))

((2 . 2)
 ((* 2 3) . (* 2 (* (* 2 3) -2)))
 ((* 1 2) . (* (* 0 (* 1 2)) 2)))

ACL2 !>(cps-trs '(((r a (r b c a) d) . c)))
((3 . 3)
 ((R 2 3 1) . (R 4 3 -3))
```

El primero de los ejemplos obtiene los pares críticos del sistema G del ejemplo 3.18. Los pares críticos calculados son[11]:

$$\{(x*(y*z), x*(y*z)), ((x*y)*(z*u), (x*(y*z))*u),$$
$$(i(x)*(x*y), e*y), (e*(x*y), x*y), (e,e), (x,x)\}$$

Para verificar la función `cps-trs`, hemos de demostrar que efectivamente calcula todos los pares críticos obtenidos por superposición entre ecuaciones del sistema que recibe como entrada. El siguiente teorema establece formalmente este hecho:

```
(defthm cps-trs-main-property
  (implies (and (member (make-equation l1 r1) E)
                (member (make-equation l2 r2) E)
                (position-p pos l1)
                (not (variable-p (occurrence l1 pos)))
                (cp-r l1 r1 pos l2 r2))
           (member (cp-r l1 r1 pos l2 r2)
                   (cps-trs E))))
```

Es decir, si existen dos ecuaciones en `E` y una posición no variable en el lado izquierdo de una de ellas tal que ambas ecuaciones determinan un par crítico en dicha posición, ese par crítico se encuentra en la lista calculada por `(cps-trs E)`.

7.5.2 Descripción de la demostración

Describimos a continuación la demostración del teorema `cps-trs-main-property`. El siguiente lema es el resultado principal que se necesita para su demostración y expresa la propiedad fundamental de la función `cps-rule-vs-rule-aux`:

```
(defthm cps-rule-vs-rule-aux-main-property
  (let ((t1 (occurrence-rec flg term q)))
```

[11] Volviendo a renombrar los números con variables x, y, z,... y usando notación infija.

```
    (implies (and (true-listp posr)
                  (position-p-rec flg q term)
                  (not (variable-p t1))
                  (unifiable t1 l2)
                  (integerp arg))
             (let* ((pos-l1 (append (revlist posr)
                                    (if flg q (cons (+ arg (car q))
                                                    (cdr q)))))
                    (cp (cons (instance r1 (mgu t1 l2))
                              (instance
                               (replace-term l1 pos-l1 r2) (mgu t1 l2)))))
               (member cp
                       (cps-rule-vs-rule-aux
                        l1 r1 flg posr arg term l2 r2)))))
```

Este teorema especifica la propiedad invariante que se verifica durante el proceso que lleva a cabo `cps-rule-vs-rule-aux`. En concreto, expresa que si `t1` es un subtérmino no variable del término o lista de términos `term` (donde `term` debe ser interpretado como se ha descrito anteriormente) que unifica con `l2`, entonces `(cps-rule-vs-rule-aux l1 r1 flg posr arg term l2 r2)` contiene al par crítico obtenido por superposición de `l2` con el subtérmino `t1` de `l1`. El teorema anterior se demuestra por inducción en la estructura de `term`.

Podemos ahora demostrar el teorema que establece la propiedad fundamental de la función `cps-rule-vs-rule`, sin más que instanciar con los valores concretos para los que la función `cps-rule-vs-rule` llama a la función `cps-rule-vs-rule-aux`.

```
(defthm cps-rule-vs-rule-main-lemma-renamed-version
 (let* ((rule1-r (number-rename-equation l1 r1 0 -1))
        (rule2-r (number-rename-equation l2 r2 1 1))
        (l1-r (nth 0 rule1-r)) (r1-r (nth 1 rule1-r))
        (l2-r (nth 0 rule2-r)) (r2-r (nth 1 rule2-r))
        (l1-r/q (occurrence l1-r q))
        (unifiable (unifiable l1-r/q l2-r))
        (theta (mgu l1-r/q l2-r)))
   (implies (and (position-p q l1-r)
                 (not (variable-p l1-r/q))
                 unifiable)
            (member (cons (instance r1-r theta)
                          (instance (replace-term l1-r q r2-r) theta))
                    (cps-rule-vs-rule l1 r1 l2 r2)))))
```

Usando la definición de `cp-r`, podemos expresar el teorema anterior en términos de pares críticos:

```
(defthm cps-rule-vs-rule-main-property
  (implies (and (position-p q l1)
                (not (variable-p (occurrence l1 q)))
                (cp-r l1 r1 q l2 r2))
```

```
          (member  (cp-r l1 r1 q l2 r2)
                   (cps-rule-vs-rule l1 r1 l2 r2))))
```

Una vez demostrada la propiedad anterior, propiedad fundamental de la función `cps--rule-vs-rule`, es inmediato obtener (por inducción en la longitud de los sistemas de ecuaciones) los siguientes teoremas, que establecen la propiedad principal de las funciones `cps-rule-vs-trs` y `cps-trs-vs-trs`:

```
(defthm cps-rule-vs-trs-main-property
  (implies (and (position-p pos l1)
                (not (variable-p (occurrence l1 pos)))
                (member (make-equation l2 r2) E)
                (cp-r l1 r1 pos l2 r2))
           (member (cp-r l1 r1 pos l2 r2)
                   (cps-rule-vs-trs l1 r1 E))))

(defthm cps-trs-vs-trs-main-property
  (implies (and (member (make-equation l1 r1) E1)
                (member (make-equation l2 r2) E2)
                (position-p pos l1)
                (not (variable-p (occurrence l1 pos)))
                (cp-r l1 r1 pos l2 r2))
           (member (cp-r l1 r1 pos l2 r2)
                   (cps-trs-vs-trs E1 E2))))
```

Como caso particular del teorema anterior (haciendo los dos sistemas `E1` y `E2` iguales a `E`), obtenemos finalmente el teorema `cps-trs-main-property`.

7.6 Decidibilidad de teorías ecuacionales

En esta sección establecemos que la teoría ecuacional de un sistema de reescritura de términos, noetheriano y cuyos pares críticos convergen, es decidible. Este resultado es un sencillo corolario del teorema pares críticos y de la decidibilidad de la relación de equivalencia descrita por una reducción convergente.

Definición 7.29 *Decimos que un sistema de reescritura de términos R es* **completo** *si es noetheriano y $u \downarrow_R = v \downarrow_R$ para todo $(u, v) \in pc(R)$.*

Nuestro objetivo es demostrar formalmente el siguiente teorema:

Teorema 7.30 *Sea R un SRT completo. Entonces $\overset{*}{\leftrightarrow}_R$ es decidible.*

Demostración:
Es evidente que $u \downarrow_R v$ para todo $(u, v) \in pc(R)$ (ya que u y v se reducen a la misma forma normal) y por tanto, por el teorema de Knuth y Bendix (7.28) la reducción $\rightarrow_R$ es localmente confluente. Por tanto $\rightarrow_R$ es una reducción convergente. Además, existe un test de reducibilidad para $\rightarrow_R$ (por ejemplo, el definido en la sección 7.2). Podemos aplicar, por tanto, el teorema 6.29 para concluir que $\overset{*}{\leftrightarrow}_R$ es decidible. □

La demostración formal de este teorema constituye un buen ejemplo de integración de toda la teoría desarrollada en esta memoria, mediante el uso de instanciación funcional. En particular, veremos cómo sacamos provecho de haber formalizado las teorías ecuacionales siguiendo el marco general descrito para reducciones abstractas en el tema 6. Los eventos que conducen a los resultados presentados en esta sección se incluyen en el libro `kb-decidability.lisp`.

7.6.1 Formalización del resultado

El siguiente encapsulado define parcialmente un sistema de reescritura de términos `(RKB)` con las propiedades asumidas en las hipótesis del teorema 7.30:

```
(encapsulate
 ((RKB () terminating-TRS-with-common-n-f-critical-pairs))
 ...
 (defthm RKB-rewrite-system
   (rewrite-system-s-p (RKB)))

 (defthm RKB-noetherian-red<
   (noetherian-red< (RKB)))

 (defun RKB-normal-form (term)
   (declare (xargs :measure (if (term-s-p term) term 0)
                   :well-founded-relation red<))
   (if (term-s-p term)
       (mv-let (reduced reducible)
               (r-reduce term (RKB))
               (if reducible
                   (RKB-normal-form reduced)
                 term))
     term))

 (defun RKB-common-normal-form (l)
   (if (endp l)
       t
     (and (equal (RKB-normal-form (lhs (car l)))
                 (RKB-normal-form (rhs (car l))))
          (RKB-common-normal-form (cdr l)))))

 (defthm RKB-common-n-f-critical-pairs
   (RKB-common-normal-form (cps-trs (RKB)))))
```

Varias aspectos de esta formalización de las hipótesis del teorema son reseñables:

- La noetherianidad de `(RKB)` se asume suponiendo su inclusión en un orden de reducción general `red<`, parcialmente definido en la subsección 7.3.3. Recuérdese que la función `noetherian-red<` compruebe que cada uno de las ecuaciones del sistema que recibe como entrada está contenida en `red<`.

- Por tanto (RKB) tiene las misma propiedad de terminación que el sistema noetheriano (RN) también definido en la subsección 7.3.3. En particular, la función RKB-normal--form se define de manera completamente análoga a RN-normal-form y por tanto tiene análogas propiedades.

- La función RKB-common-normal-form comprueba que todos los pares de términos de una lista tienen formas normales comunes respecto al sistema (RKB). Por tanto, teniendo en cuenta las propiedades de la función cps-trs, el asumir que todos los pares críticos de (RKB) tienen forma normal común se traduce aquí en asumir (RKB--common-normal-form (cps-trs (RKB))).

Para probar el teorema 7.30, hemos de probar que existe un algoritmo de decisión para la teoría ecuacional de (RKB). Este algoritmo va a consistir simplemente en comparar la igualdad entre las formas normales de dos términos:

```
(defun RKB-equivalent (t1 t2)
  (equal (RKB-normal-form t1) (RKB-normal-form t2)))
```

Para probar que RKB-equivalent define un algoritmo que decide la teoría ecuacional de (RKB), hemos de demostrar su corrección y completitud. La completitud afirma que cuando el algoritmo actúa sobre dos términos equivalentes en (RKB), éste responde afirmativamente:

```
(defthm RKB-equivalent-complete
  (implies (eq-equiv-s-p t1 t2 p (RKB))
           (RKB-equivalent t1 t2)))
```

La corrección del algoritmo establece que cuando el algoritmo responde afirmativamente sobre dos términos de entrada, entonces existe una prueba ecuacional que demuestra su equivalencia:

```
(defthm RKB-equivalent-sound
  (implies (and (term-s-p t1) (term-s-p t2)
                (RKB-equivalent t1 t2))
           (eq-equiv-s-p
             t1 t2 (RKB-make-proof-common-n-f t1 t2) (RKB))))
```

Esta prueba ecuacional viene definida por la siguiente función RKB-make-proof-common-n-f. Necesita como función auxiliar RKB-proof-irreducible, que construye una prueba ecuacional que conecta un término con su forma normal respecto de (RKB), cuya definición omitimos, ya que es la versión completamente análoga (reemplazando (RN) por (RKB)) de la función RN-proof-irreducible definida en la sección 7.3.4. La función RKB-make-proof-common-n-f simplemente concatena las subpruebas que llevan a cada elemento a su forma normal:

```
(defun RKB-make-proof-common-n-f (t1 t2)
   (append (RKB-proof-irreducible t1)
           (inverse-proof (RKB-proof-irreducible t2))))
```

7.6.2 Descripción de la demostración

La demostración formal sigue, esencialmente, la de la prueba a mano del teorema 7.30 presentada anteriormente. Se trata de:

- Probar que se cumplen las condiciones que permiten aplicar el teorema de pares críticos para el sistema `(RKB)` y concluir su confluencia local.
- Se tendría por tanto que la reducción ecuacional que define `(RKB)` es convergente, lo que nos permitiría aplicar los resultados de la sección 6.5 para concluir la decidibilidad de su clausura de equivalencia, que como vimos en la sección 7.1, coincide con su teoría ecuacional.

Veamos en primer lugar que el sistema `(RKB)` cumple las condiciones del teorema de pares críticos. Para ello será necesario definir una función `RKB-transform-critical--pair` tal que que `(RKB-transform-critical-pair l1 r1 pos l2 r2)` sea una prueba valle equivalente del par crítico determinado por superposición de la ecuación `l2=r2` en la posición `pos` del lado izquierdo de la ecuación `l1=r1`, para cualquier par de reglas `l1=r1`, y `l2=r2` de `(RKB)`.

La función `RKB-transform-critical-pair` se define de manera sencilla, usando la función `RKB-make-proof-common-n-f`:

```
(defun RKB-transform-critical-pair (l1 r1 pos l2 r2)
  (let ((cp-r (cp-r l1 r1 pos l2 r2)))
    (RKB-make-proof-common-n-f (lhs cp-r) (rhs cp-r)))))
```

Para probar sus propiedades, basta probar previamente los dos teoremas fundamentales sobre las funciones `RKB-make-proof-common-n-f` y `RKB-common-normal-form`, que a continuación presentamos:

```
(defthm RKB-make-proof-common-n-f-valley-proof
  (implies (and (term-s-p t1) (term-s-p t2)
                (equal (RKB-normal-form t1)
                       (RKB-normal-form t2)))
           (and (eq-equiv-s-p t1 t2 (RKB-make-proof-common-n-f t1 t2)
                              (RKB))
                (steps-valley (RKB-make-proof-common-n-f t1 t2)))))

(defthm RKB-common-normal-form-main-property
  (implies (and (member p l)
                (RKB-common-normal-form l))
           (equal (RKB-normal-form (lhs p))
                  (RKB-normal-form (rhs p)))))
```

Es decir, si dos términos `t1` y `t2` tienen la misma forma normal, entonces `(RKB-make--proof-common-n-f t1 t2)` es una prueba valle de la equivalencia de `t1` y `t2`. Además, si se verifica `(RKB-common-normal-form l)` para cierta lista `l`, entonces los elementos de `l` son pares de términos con forma normal común, respecto de `(RKB)`.

Como estamos asumiendo `(RKB-common-normal-form (cps-trs (RKB)))`, entonces todos los elementos de `(cps-trs (RKB))` tienen forma normal común. Usando `cps-trs--main-property` (sección anterior), se tiene que cualquier par crítico de `(RKB)` tiene forma normal común y por tanto la función `RKB-transform-critical-pair` construye una prueba valle para cada par crítico de `(RKB)`. Es decir, se cumplen las condiciones del teorema de pares críticos:

```
(defthm RKB-joinable-critical-pairs
  (implies (and (member (make-equation l1 r1) (RKB))
                (member (make-equation l2 r2) (RKB))
                (position-p pos l1)
                (not (variable-p (occurrence l1 pos))))
           (let* ((cp-r (cp-r l1 r1 pos l2 r2)))
             (implies cp-r
                      (eq-equiv-s-p
                       (lhs cp-r)
                       (rhs cp-r)
                       (RKB-transform-critical-pair l1 r1 pos l2 r2)
                       (RKB))))))))

(defthm RKB-transform-critical-pair-is-a-valley
  (steps-valley
   (RKB-transform-critical-pair l1 r1 pos l2 r2)))
```

Podemos, pues, aplicar el teorema de pares críticos probado anteriormente, para deducir la confluencia local de `(RKB)`. Para ello definimos una función `RKB-transform--eq-local-peak` de manera totalmente análoga a la función `transform-eq-local-peak` definida en la demostración del teorema de pares críticos, en la que ahora `(RKB)` juega el papel de `(EKB)`. Por instanciación funcional, se tiene que dicha función proporciona una prueba valle equivalente por cada pico local en `(RKB)`:

```
(defthm RKB-transform-eq-local-peak-is-a-proof
  (implies
   (and
    (eq-equiv-s-p t1 t2 p (RKB))
    (local-peak-p p))
   (eq-equiv-s-p t1 t2 (RKB-transform-eq-local-peak p) (RKB))))

(defthm RKB-transform-eq-local-peak-is-a-valley
  (implies
   (and
    (eq-equiv-s-p t1 t2 p (RKB))
    (local-peak-p p))
   (steps-valley (RKB-transform-eq-local-peak p))))
```

La definición de la función `RKB-transform-eq-local-peak` es muy extensa. La omitimos aquí porque, como ya hemos comentado, es totalmente análoga a la función `transform-eq-local-peak` definida en la sección 7.4. Nótese que algunas de las funciones auxi-

liares que entonces se usaron no eran específicas del sistema `(EKB)` y aquí no es necesario redefinirlas. Consúltese el libro `kb-decidability.lisp` para más detalles.

De los dos teoremas anteriores, se deduce que la reducción asociada a `(RKB)` es localmente confluente. Según el teorema `R-noetherian-if-subsetp-of-a-reduction--ordering`, esta reducción también es noetheriana, ya que estamos asumiendo `(noetherian-red< (RKB))`. Es decir, estamos ante una reducción convergente. Además, disponemos de un test de reducibilidad, definido por la función `eq-reducible` de la sección 7.2, como demuestran los teoremas `eq-reducible-implies-eq-legal` y `not-eq-reducible--nothing-eq-legal`. Por tanto, se cumplen las hipótesis del teorema sobre decidibilidad de reducciones abstractas convergentes presentado en la sección 6.5.

La función `RKB-equivalent` es análoga a la función `r-equivalent` que definía el algoritmo de decisión para reducciones abstractas, en la sección 7.2. Simplemente la reducción ecuacional asociada a `(RKB)` sustituye ahora a la reducción abstracta. Así pues, los teoremas de corrección y completitud de `RKB-equivalent` se obtienen por una simple instanciación funcional de los teoremas análogos para `r-equivalent`.

Teoría necesaria para la demostración

Resulta interesante recopilar todos los resultados que se han necesitado para la demostración de este teorema de decidibilidad. Esta teoría incluye:

- El teorema de pares críticos (sección 7.4). A su vez, en la demostración de este teorema se necesitaba:
  - Las propiedades básicas de términos y sustituciones, en particular las relativas a la estructura de árbol de los mismos (sección 3.2).
  - Para las propiedades de las superposiciones críticas, un algoritmo de unificación verificado (sección 4.4) y el concepto de subsunción entre sustituciones y sus propiedades (sección 3.4).
  - Para el renombrado de ecuaciones, la definición y propiedades de un algoritmo de renombrado y sus propiedades (sección 4.1).
  - Para la manipulación de pruebas ecuacionales, todos los conceptos y propiedades relativos a las mismas y al álgebra de pruebas (sección 7.1).
- La definición y verificación de la función `cps-trs` (sección 7.5), que calcula todos los pares críticos de un sistema de ecuaciones.
- La demostración de que toda reducción (abstracta) convergente tiene una clausura de equivalencia decidible (sección 6.5). Recuérdese que en la demostración de este resultado, se necesitaba:
  - La demostración de que una reducción normalizadora con la propiedad de Church-Rosser tiene clausura de equivalencia decidible (sección 6.3).
  - El lema de Newman (sección 6.4). Para este resultado, necesitábamos las propiedades de las relaciones entre multiconjuntos bien fundamentadas (sección 5.2).

Para integrar todos estas teorías es necesario usar la regla de instanciación funcional con dos finalidades distintas: aplicar el teorema de pares críticos a `(RKB)` y trasladar resultados sobre reducciones abstractas a la reducción de reescritura. En el apéndice C, subsección C.5.4, comentamos algunos detalles técnicos sobre esta cuestión.

Como se observa, gran parte de la teoría desarrollada en esta memoria confluye en el resultado presentado en esta sección. Sin embargo, nótese que esta teoría auxiliar queda "oculta" por los mecanismos de estructuración que posee ACL2 (sección 2.2). Por ejemplo, los lemas auxiliares que se necesitaron para cada resultado parcial no están presentes ya en la demostración de los resultados de esta sección. Otro aspecto interesante es que en el enunciado de este teorema no se hace referencia alguna a conceptos abstractos de reducciones tales como test de reducibilidad o de aplicabilidad, u operaciones en el álgebra de prueba. Estos conceptos han servido para obtener los resultados intermedios necesarios pero no aparecen en la formulación final del resultado. Es más, el algoritmo `RKB-equivalent` no maneja en ningún momento los conceptos de operador ecuacional ni de prueba ecuacional.

Sumario

En este capítulo hemos descrito:

- La formalización del concepto de teoría ecuacional, como relación de equivalencia asociada por la reducción de reescritura.
- Definición y verificación de un test de reducibilidad para la reducción de reescritura.
- Definición del concepto de sistema de reescritura de términos y de orden de reducción.
- Definición y verificación de una función de cálculo de formas normales respecto de un sistema de reescritura.
- Demostración formal del teorema de pares críticos de Knuth y Bendix.
- Definición y verificación de una función que calcula los pares críticos de un sistema de ecuaciones.
- Recopilación de los resultados anteriores, para demostrar formalmente que la teoría ecuacional descrita por un sistema de reescritura completo es decidible.

Capítulo 8

Conclusiones

En los capítulos precedentes, se ha presentado una teoría computacional, desarrollada en ACL2, acerca de los términos de primer orden, las reducciones abstractas, la lógica ecuacional y los sistemas de reescritura de términos. A nuestro juicio, los principales objetivos conseguidos en esta memoria son los siguientes:

- Se ha mostrado cómo una lógica aparentemente poco expresiva (de primer orden, sin cuantificación, etc.), se puede usar para aplicar métodos formales al estudio de los sistemas de reescritura de términos. Destacan, a nuestro juicio, cinco resultados:
  - Demostración de la estructura de retículo bien fundamentado del conjunto de los términos de primer orden, incluyendo la verificación del algoritmo de unificación de Martelli y Montanari.
  - Buena fundamentación de la extensión a multiconjuntos de una relación bien fundamentada.
  - Lema de Newman.
  - Teorema de pares críticos de Knuth y Bendix.
  - Decidibilidad de teorías ecuacionales descritas por sistemas de reescritura completos.
- Se ha elaborado una biblioteca de resultados básicos sobre reescritura, formada por la colección de libros desarrollados en ACL2, que puede ser reutilizada en trabajos posteriores.
- Se ha constatado que las tareas de deducción formal y de cálculo se pueden llevar a cabo en un mismo entorno: como consecuencia de la teoría desarrollada, se han definido una serie de algoritmos básicos en el campo de la reescritura, que están verificados formalmente y que se pueden ejecutar en cualquier sistema Common Lisp con relativa eficiencia (véase el apéndice B).

En cuanto al desarrollo de la teoría, éstas son las principales características de la formalización presentada:

- Las reducciones abstractas se han formalizado en un marco completamente general. La regla derivada de instanciación funcional permite trasladar los resultados abstractos a la teoría de los sistemas de reescritura. Con este mismo grado de generalidad se ha desarrollado la teoría sobre las relaciones entre multiconjuntos.

- Los conceptos de "prueba abstracta" y de "prueba ecuacional" son claves en el desarrollo de la teoría. Como en [2], las pruebas se tratan como objetos con entidad propia, que pueden ser transformados para obtener nuevas pruebas. Este punto de vista tiene gran influencia tanto en la formalización como en las demostraciones.

- La técnica del razonamiento compuesto se usa repetidas veces. Inicialmente, se verifican versiones simples y posiblemente ineficientes, de los algoritmos, ya que el razonamiento sobre ellos es más sencillo. Cuando se define una versión más refinada del algoritmo, se prueban resultados de equivalencia que permiten trasladar los resultados obtenidos para la versión simplificada.

Algunos datos de interés

En la tabla de la figura 8.1, se presentan algunos datos cuantitativos sobre la teoría desarrollada. La primera columna contiene el nombre de cada uno de los libros ACL2. Las tres columnas siguientes contienen el número de líneas (excluyendo comentarios y líneas en blanco), de definiciones y de teoremas en cada uno de los libros. Se han incluido estos datos porque pensamos que pueden dar una idea del "tamaño" de la teoría desarrollada en cada uno de los libros. En la última columna se incluye el número de teoremas que que necesitan alguna sugerencia para completar con éxito su demostración y que suponen aproximadamente la cuarta parte del total. La mayoría de las indicaciones son para usar instancias de teoremas, o bien para habilitar o deshabilitar reglas. El resto de teoremas se prueban automáticamente por el sistema, sin ningún tipo de ayuda.

Por otra parte, el esfuerzo humano invertido es difícil de cuantificar. En nuestro caso, el trabajo realizado se ha combinado, en sus fases iniciales, con el aprendizaje del sistema. Es evidente que si se abordara ahora un trabajo de similares características al presentado en esta memoria, el número de horas dedicadas sería mucho menor. También, el hecho de disponer cada vez de una biblioteca más amplia de reglas y de utilidades, hace que el tiempo de dedicación para una demostración sea menor al final del desarrollo.

Este trabajo se comenzó usando Nqthm. Debido a que la mayoría de usuarios de Nqthm se habían trasladado a ACL2 (Nqthm es poco usado en la actualidad) y puesto que uno de los objetivos en este trabajo es la ejecución relativamente eficiente de las funciones verificadas, decidimos trasladar el trabajo realizado hasta el momento a ACL2. El paso de Nqthm a ACL2 no fue, sin embargo, excesivamente costoso, debido a la similitud de ambos sistemas.

Algunas consideraciones

Hacemos algunas reflexiones sobre la experiencia que ha supuesto el desarrollo formal presentado en esta memoria, tanto en los aspectos relacionados específicamente con el sistema ACL2, como en la tarea de formalización en general:

- Como Nqthm, el sistema ACL2 es excelente en la automatización de la inducción y en el uso de simplificación. Esto hace que el usuario se vea "liberado" de la demostración de teoremas relativamente triviales.

- En cuanto a las novedades que presenta ACL2 frente a Nqthm, éstas son las principales ventajas encontradas:

Libro	Líneas	Definiciones	Teoremas	Consejos
`basic`	378	22	79	2
`terms`	770	53	76	12
`matching`	325	7	48	8
`subsumption`	295	13	29	18
`subsumption-subst`	327	16	38	13
Subtotal	2095	111	270	53
`renamings`	578	9	64	25
`subsumption-well-founded`	216	3	30	7
`anti-unification`	434	10	37	6
`unification-pattern`	808	7	105	33
`unification`	277	12	24	8
`mg-instance`	159	3	17	11
`lattice-of-terms`	148	17	20	5
Subtotal	2620	61	297	95
`multiset`	576	24	69	16
`defmul`	652	24	14	13
`ackermann`	127	10	10	4
`mccarthy-91`	110	10	13	3
Subtotal	1465	68	106	36
`abstract-proofs`	152	18	17	0
`confluence`	242	13	32	7
`newman`	451	15	56	10
`convergent`	269	22	20	8
Subtotal	1114	68	125	25
`equational-theories`	219	12	24	7
`rewriting`	528	18	54	13
`critical-pairs`	1710	54	152	41
`kb-decidability`	370	17	24	7
Subtotal	2827	101	254	68
Total	10121	409	1052	277

Figura 8.1: Datos cuantitativos

  - Reescritura congruente: ya se ha visto a lo largo de la memoria que la combinación de `defequiv` con `defcong` ha resultado muy útil en la automatización de resultados que en Nqthm hubieran requerido mayor interacción.
  - Estructuración de teorías: creemos que es fundamental, tanto para la eficiencia de las demostraciones como para la claridad de las mismas, estructurar las teorías en libros independientes, que sólo "exportan" sus resultados no locales. Este mismo principio se aplica al desarrollo interno de un libro, usando encapsulados.
  - La ejecución de funciones en ACL2, una vez verificadas las protecciones y compiladas (apéndice B), es mucho más eficiente que en Nqthm.

- El desarrollo de pruebas formales tiene la evidente ventaja del rigor que aporta a la demostración de los resultados obtenidos. Pero, además, creemos que la formalización permite una mayor comprensión, por parte del usuario, de los resultados que se demuestran. La formalización no suele ser una tarea rutinaria. Como afirma Shankar [68] en sus conclusiones, el proceso de formalizar y verificar automáticamente un argumento informal en matemáticas, es una actividad creativa. Por ejemplo, escoger una buena representación de los objetos que se estudian es fundamental para el desarrollo posterior. Elegir una secuencia adecuada de lemas también puede ser crucial para el éxito de la prueba.

- Una cuestión que creemos debe ser abordada en el futuro es la legibilidad de las demostraciones. Aunque el usuario ACL2 obtiene una mayor comprensión de la prueba que realiza, la estrategia por la cual se alcanza el resultado final, puede no quedar clara al leer el libro ACL2 correspondiente, incluso para un usuario experto en ACL2.

- En toda formalización, existe el problema de saber si el modelo construido refleja realmente aquello sobre lo que se quiere razonar. Para ello, creemos que es importante disponer de un entorno, como el que proporciona ACL2, en el que además de razonar sobre los algoritmos, sea posible ejecutarlos. Esto aumenta la confianza en la formalización.

- El principio de definición en ACL2 obliga a asegurar la terminación de las funciones definidas en ACL2, *para cualquier* argumento de entrada. Esto puede suponer a veces que en la definición de un algoritmo sea necesario incluir condiciones que afectan a la eficiencia de su ejecución. Esto se observa, por ejemplo, en la definición de la función que calcula formas normales respecto de un sistema de reescritura noetheriano (subsección 7.3.4). Puesto que su dominio pretendido es el conjunto de los términos de primer orden en una signatura y sólo para estos objetos tenemos asegurada la terminación, debemos incluir en la definición de esta función recursiva una condición que compruebe si el objeto que se recibe como entrada es un término. Esta comprobación afecta a la eficiencia de ejecución del algoritmo y en realidad no es necesaria para su ejecución (véase la discusión de la página 269). Sería deseable que el sistema obviase estas comprobaciones en el momento de ejecutar la función.

- Hemos de subrayar la conveniencia del empleo de la abstracción. Al igual que las matemáticas, la deducción automática se beneficia de tratar sólo con los aspectos

esenciales del problema. En nuestro caso, hemos visto las ventajas de razonar previamente con reducciones abstractas, antes de pasar al estudio de los sistemas de reescritura. Relacionado con esta cuestión, destacar la importancia del razonamiento compuesto, que permite tratar inicialmente con prototipos que luego se refinan.

- Es conveniente la creación de bibliotecas básicas, que permitan empezar una formalización reutilizando teorías formales, desarrolladas por otros usuarios.

- Demostrar teoremas no triviales en ACL2 no es una tarea sencilla. Nuestro objetivo original era verificar un algoritmo de completación de sistemas de reescritura. Sin embargo, sólo la teoría necesaria para que esta tarea fuera abordable constituye, a nuestro juicio, un trabajo de gran envergadura. Como se afirma en el capítulo 1 de [37], la dificultad radica más en la construcción de una prueba formal, que en el manejo del demostrador. Además de conocer el demostrador, el usuario debe estar familiarizado con la teoría que formaliza (quizá sea esa la razón por la que la mayoría de las pruebas formales son acerca de sistemas formales).

Trabajo futuro

La investigación presentada en esta memoria se puede continuar en varias direcciones:

- El grado de abstracción con el que se han formalizado las reducciones parece prometedor. Existen otras propiedades importantes de las reducciones abstractas que no hemos abordado. Por ejemplo, otros criterios de confluencia o el estudio de reducciones en conjuntos cociente [30].

- La formalización de la terminación de los sistemas de reescritura presentada en esta memoria, aunque correcta desde el punto de vista teórico, es deficiente desde el punto de vista práctico (página 269). El problema principal radica en cómo encontrar una prueba en ACL2 de la buena fundamentación de los principales órdenes de reducción existentes (LPO, RPO, etc.). Una posible línea de investigación sería estudiar si esto es posible.

- Como se muestra en el apéndice B, subsección B.3, el carácter aplicativo de ACL2 hace que la representación de los términos mediante listas sea ineficiente, ya que las estructuras no se comparten. Sería deseable almacenar un término en forma de grafo, en lugar de representarlo en forma de árbol. Los *objetos de hebra simple*[1], permiten definir operaciones destructivas en ACL2, sin perder la semántica aplicativa (véase `stobj`). Una interesante dirección de investigación consistiría en estudiar si estas estructuras permiten una representación más eficiente de los términos.

- El trabajo de McCune y Shumsky descrito en [51] hace pensar en otra posible aplicación. Se trata de usar las funciones ACL2 que hemos verificado, a fin de comprobar la corrección de la salida de otros demostradores. Por ejemplo, de la misma manera que en [51] se describe la interacción de ACL2 y Otter, se podrían combinar las funciones verificadas en esta memoria con la salida del sistema RRL, para construir y verificar automáticamente algoritmos de decisión de ciertas teorías ecuacionales.

[1] Del inglés *single-threaded objects*.

- Por último, la continuación natural después de haber demostrado el teorema de pares críticos de Knuth y Bendix es verificar un algoritmo de completación. Estamos convencidos de que la teoría desarrollada permite abordar este problema.

Apéndice A

Preliminares matemáticos

A.1 Relaciones

Definición A.1 *Una* **relación** n**-aria** R *definida en un conjunto* A *es un subconjunto de* A^n*. Si* $(a_1, \ldots, a_n) \in R$*, lo notamos por* $R(a_1, \ldots, a_n)$*.*

En lo que sigue, si hablamos de una relación R en A, entenderemos implícitamente que se trata de una relación binaria. En ese caso, podremos usar la notación aRb en lugar de $R(a, b)$. La relación **inversa** de R es la relación $R^{-1} = \{(x, y) : (y, x) \in R\}$. Puesto que una reducción es una relación binaria (definición 6.1), algunos de los conceptos definidos en esta sección se solapan con los definidos en la sección 6.1.

Definición A.2 *Sea* A *un conjunto y* R *una relación en* A*. Diremos que:*

- R *es* **reflexiva** *si* xRx *para todo* $x \in A$*.*
- R *es* **irreflexiva** *si no existe* $x \in A$ *tal que* xRx*.*
- R *es* **simétrica** *si* xRy *implica que* yRx *para todo* $x, y \in A$*.*
- R *es* **transitiva** *si* xRy *y* yRz *implica que* xRz *para todo* $x, y, z \in A$*.*

Definición A.3 *Sean* R *y* S *dos relaciones binarias en* A*. La composición de* R *y* S *es la relación* $R \circ S = \{(x, z) :$ *existe* $y \in A$ *tal que* xRy *y* $yRz\}$*.*

Definición A.4 *Sea* R *una relación en un conjunto* A*. Decimos que* R *es una relación de* **equivalencia** *si es reflexiva, simétrica y transitiva. Dado* $x \in A$*, la* **clase de equivalencia** *de* x *respecto de* R *es el conjunto* $[x]_R = \{y \in A : xRy\}$*. El* **conjunto cociente** *de* A *módulo* R *es* $A/R = \{[x]_R : x \in A\}$*.*

Definición A.5 *Sea* R *una relación en un conjunto* A*. Decimos que* R *es un* **preorden** *si es reflexiva y transitiva. Decimos que* R *es un* **orden parcial** *(estricto), o simplemente una relación de orden, si es irreflexiva y transitiva.*

Es bastante frecuente usar la notación $>$ ó $<$ para nombrar relaciones binarias, especialmente si la relación es de orden. Para preórdenes, la notación habitual es $\succeq$ ó $\preceq$. Si $<$ ó $\preceq$ es una relación, entonces $>$ ó $\succeq$, respectivamente, denota su relación inversa y

viceversa. La relación inversa de un orden parcial es un orden parcial. Lo mismo ocurre con los preórdenes.

Dado un orden parcial $<$ en A, entonces la relación $< \cup =$, notada usualmente por $\leq$, es un preorden en A. Dado un preorden $\preceq$ en A, entonces la relación $\preceq \setminus \succeq$ es un orden parcial en A, notado usualmente por $\prec$.

Definición A.6 *Sea A un conjunto, $\preceq$ un preorden en A y $B \subseteq A$. Decimos que $x \in A$ es una* **cota superior** *de B si para todo $y \in B$ se verifica $y \preceq x$. Si x es una cota superior de B, decimos que es un* **supremo** *de B si $x \preceq z$ para cualquier z que sea cota superior de B.*

De manera dual, decimos que $x \in A$ es una **cota inferior** *de B si para todo $y \in B$ se verifica $x \preceq y$. Si x es una cota inferior de B, decimos que es un* **ínfimo** *de B si $z \preceq x$ para cualquier z que sea cota inferior de B.*

Definición A.7 *Sea A un conjunto y $\preceq$ un preorden en A. Decimos que A tiene estructura de* **retículo** *(o, simplemente, que es un retículo) respecto de $\preceq$, si para todo $x, y \in A$, el conjunto $\{x, y\}$ tiene ínfimo y supremo. Diremos además que es* **completo** *si todo $B \subseteq A$, tiene ínfimo y supremo.*

A.2 Inducción bien fundamentada

La definición siguiente es la misma que la dada en 5.7:

Definición A.8 *Una relación R en un conjunto A se dice* **noetheriana** *(o que* **termina***) si no existe una sucesión infinita $\{x_n\}_{n\geq 0}$ de elementos en A tal que $x_i R x_{i+1}$ para todo $i \geq 0$. En ese caso, decimos que $\{x_n\}_{n\geq 0}$ es una R-***derivación infinita*.

Definición A.9 *Sea $B \subseteq A$ y R una relación sobre A. Decimos que $b \in B$ es* **minimal** *en B respecto de R, si cRb implica que $c \notin B$. Una relación sobre un conjunto A está* **bien fundamentada** *si todo subconjunto no vacío de A tiene un elemento minimal respecto de dicha relación.*

Definición A.10 *Una propiedad P sobre un conjunto A admite una prueba por* **inducción noetheriana** *con respecto a una relación R sobre A si siempre que*

$$(\forall a \in A)[(\forall b \in A)[aRb \Rightarrow P(b)] \Rightarrow P(a)] \tag{A.1}$$

se tiene que $(\forall a \in A)[P(a)]$. Una relación R sobre un conjunto A decimos que **admite inducción noetheriana** *si toda propiedad P sobre A admite una prueba por inducción noetheriana con respecto a R.*

La hipótesis de la fórmula A.1 es lo que usualmente llamamos hipótesis de inducción. El principio de inducción noetheriana, en relación con otros principios de inducción clásicos, tiene la ventaja de que la relación respecto a la que se realiza no necesariamente ha de ser total.

Los conceptos de noetherianidad, buena fundamentación e inducción noetheriana están íntimamente ligados, como enunciamos en el siguiente teorema:

Teorema A.11 *Sea R una relación sobre un conjunto A. Entonces, son equivalentes:*

1) R es noetheriana.

2) R^{-1} está bien fundamentada.

3) R admite inducción noetheriana.

Demostración:

$\boxed{1)\Rightarrow 2)}$ Supongamos que R^{-1} no está bien fundamentada y construyamos una R-derivación infinita $\{b_n\}_{n\geq 0}$ de elementos en A. Sea $B \subseteq A$ no vacío tal que no tiene elementos minimales y sea $b_0 \in B$. Entonces existe $b_1 \in B$ tal que $b_1 R^{-1} b_0$, ya que en otro caso b_0 sería minimal. De la misma manera, existe $b_2 \in B$ tal que $b_2 R^{-1} b_1$. De esta manera es posible obtener una R-derivación infinita $b_0 R b_1 R b_2 \cdots$.

$\boxed{2)\Rightarrow 3)}$ Supongamos que R^{-1} está bien fundamentada y que R no admite inducción noetheriana. Por tanto, existe una propiedad P verificando la fórmula A.1 y tal que no se verifica $(\forall a \in A)[P(a)]$. En consecuencia, $B = \{a \in A : \neg P(a)\}$ es un conjunto no vacío y por estar R^{-1} bien fundamentada tiene un elemento minimal $a \in B$. Por definición de elemento minimal, cualquier $b \in A$ tal que aRb necesariamente cumple que $b \notin B$, o equivalentemente, se verifica $P(b)$. Luego de la fórmula A.1 se deduce $P(a)$, lo cual está en contradicción con que $a \in B$.

$\boxed{3)\Rightarrow 1)}$ Sea $P(x)$ la propiedad "no existe una R-derivación infinita cuyo primer elemento es x". Es evidente que R es noetheriana si y sólo si $(\forall x \in A)[P(x)]$. Probemos esto por inducción noetheriana. Supongamos que $\forall y \in A$ tal que xRy se verifica $P(y)$. Entonces evidentemente se verifica $P(x)$, ya que si no, existiría una R-derivación infinita comenzando en un cierto y tal que xRy. Puesto que R admite inducción noetheriana, se verifica $(\forall x \in A)[P(x)]$.

□

El teorema anterior justifica que, alternativamente, llamemos **inducción bien fundamentada** a la inducción noetheriana.

A.3 Producto lexicográfico

Definición A.12 *Sean R_A una relación sobre un conjunto A y R_B una relación sobre un conjunto B. Entonces definimos la relación* **producto (binario) lexicográfico** *R sobre $A \times B$ de la siguiente manera:*

$$(a,b)R(c,d) \Leftrightarrow (aR_Ac) \vee (a = c \wedge bR_Bd).$$

Teorema A.13 *La relación producto lexicográfico de dos relaciones noetherianas es noetheriana.*

Demostración:
Sean R_A y R_B dos relaciones noetherianas sobre A y B, respectivamente, y R su relación producto lexicográfico. Sea $T(x,y)$ la propiedad (definida en $A \times B$) "no existen R-derivaciones infinitas cuyo primer elemento es (x,y)". Es evidente que R es noetheriana

si y sólo si $(\forall x \in A)[(\forall y \in B)T(x,y)]$. Si $S(x)$ es la propiedad definida en A por $(\forall y \in B)T(x,y)$, bastará probar $(\forall x \in A)S(x)$. Esto lo podemos hacer por inducción noetheriana sobre la relación R_A (que es noetheriana). Sea $x \in A$ y supongamos que para cualquier $x' \in A$ tal que xR_Ax', se verifica $S(x')$. Para probar $S(x)$, (es decir $(\forall y \in B)T(x,y)$), usaremos inducción noetheriana sobre R_B (que es noetheriana). Por tanto, sea $y \in B$ tal que para todo $y' \in B$ tal que yR_By', se verifica $T(x,y')$. Si $T(x,y)$ no fuera cierto, entonces existiría una R-derivación infinita

$$(x,y)R(x',y')R(x'',y'')R\cdots.$$

Como $(x,y)R(x',y')$, existen dos posibles casos:

Caso 1 xR_Ax'. Entonces por hipótesis de inducción se verifica $S(x')$, lo cual está en contradicción con la existencia de la R-derivación infinita $(x',y')R(x'',y'')R\cdots$.

Caso 2 $x = x'$ e yR_By'. Entonces, por hipótesis de inducción, se verifica $T(x,y')$ y (puesto que $x = x'$) se tiene $T(x',y')$. Lo cual vuelve a estar de nuevo en contradicción con la existencia de la R-derivación infinita $(x,y')R(x'',y'')R\cdots$.

En los dos casos se obtiene contradicción, luego podemos deducir $T(x,y)$. Aplicando inducción noetheriana sobre R_B se deduce $S(x)$. Nuevamente aplicando inducción noetheriana sobre R_A, obtenemos $(\forall x \in A)S(x)$. □

Definición A.14 *Sean R_i, $1 \leq i \leq n$, n relaciones sobre n conjuntos A_i, $1 \leq i \leq n$ respectivamente. Decimos que la relación R sobre $A_1 \times \cdots \times A_n$ es el* **producto (n-ario) lexicográfico** *de las relaciones R_i si:*

$$(a_1,\ldots,a_n)R(b_1,\ldots,b_n) \Leftrightarrow (\exists k \leq n)[[(\forall i < k)a_i = b_i] \wedge a_kR_kb_k]$$

Teorema A.15 *El producto lexicográfico de relaciones noetherianas es noetheriano.*

Demostración:
Sean n relaciones noetherianas R_i definidas sobre n conjuntos A_i, $1 \leq i \leq n$ y R su producto lexicográfico. Es fácil ver que R es el producto binario lexicográfico de R_1 (definido en A_1) por el producto lexicográfico de $R_2,\ldots,R_n$ sobre $A_2 \times \cdots \times A_n$. Con este resultado, el teorema se muestra fácilmente por inducción sobre n usando el teorema A.13. □

A.4 Cadenas y árboles

Definición A.16 *Dado un conjunto A, una* **cadena** *en A es un elemento u de A^n, donde $n \in \mathbb{N}$. Este número natural se denomina la* **longitud** *de u y se denota por $|u|$. La única cadena de longitud 0, se denomina* **cadena vacía** *y es notada por ϵ.*

El conjunto $\bigcup_{i\geq 0} A^i$ de todas las cadenas en A se denota por A^*.

Definición A.17 *Dadas dos cadenas $u = (u_1,\ldots,u_n)$ y $v = (v_1,\ldots,v_m)$, la* **concatenación** *de ambas, notada $u \cdot v$ es la cadena $(u_1,\ldots,u_n,v_1,\ldots,v_m)$. Si no hay lugar a confusión, lo notaremos también como uv.*

Definición A.18 *Dado $u, v \in A^*$, v es prefijo de u si existe $w \in A^*$ tal que $u = v \cdot w$. Si v es prefijo de u lo notamos por $v \leq u$. Análogamente, v es sufijo de u si existe $w \in A^*$ tal que $u = w \cdot v$. v es* **subcadena** *de u si existen $w_1, w_2 \in A^*$ tal que $u = w_1 \cdot v \cdot w_2$. Si v es prefijo (sufijo, subcadena) de u, se dice* **propio** *si $v \neq u$. Si u no es prefijo de u, ni u es prefijo de v, entonces decimos que u y v son* **disjuntas** *y lo notamos por $u|v$.*

Definición A.19 *Un* **dominio de árbol** *D es un subconjunto no vacío de $\mathbb{N}_+^*$ (cadenas de enteros positivos), cumpliendo:*

- *Para todo $u \in D$, todo prefijo de u está en D.*
- *Para todo $u \in D$ e $i \in \mathbb{N}_+$, si $ui \in D$, entonces $uj \in D$ para $1 \leq j \leq i$.*

Definición A.20 *Dado un conjunto Σ (llamado conjunto de* **etiquetas***), un* **árbol Σ-etiquetado** *(o simplemente un árbol) es una aplicación $t : D \to \Sigma$, siendo D un dominio de árbol. El dominio de un árbol t lo notamos por $dom(t)$.*

Definición A.21 *Sea t un Σ-árbol. Un* **nodo** *o* **dirección** *de t es un elemento de $dom(t)$. El* **grado** *de un nodo u es el cardinal $g(u) = |\{i : ui \in dom(t)\}|$. El árbol t está* **finitamente ramificado** *si para todo $u \in dom(t)$, $g(u)$ es finito. Una* **hoja** *de t es un nodo u tal que $g(u) = 0$. El nodo representado por la palabra vacía ϵ se denomina* **raíz** *del árbol. El árbol t es* **finito** *si $dom(t)$ lo es. Dado $u \in dom(t)$, los nodos $ui \in dom(t)$ se denominan* **hijos** *de u. Dados dos nodos $u, v \in dom(t)$, decimos que u es* **antecesor** *de v, (o que v es* **sucesor** *de u) si $u \leq v$.*

Definición A.22 *Sea t un Σ-árbol. Un* **camino finito** *con* **origen** *u y* **destino** *v es una sucesión de nodos de t, $u_0, \ldots, u_n$ con $u_0 = u$ y $u_n = v$ y tal que para todo j, $1 \leq j \leq n$, $u_j = u_{j-1} i_j$. La* **longitud** *del camino $u_0, \ldots, u_n$ es n. Una* **rama** *es un camino con origen en la raíz y destino en una hoja. Un* **camino infinito** *con origen u es una secuencia infinita de nodos de t, $u_0, u_1, u_2, \ldots$ tal que $u_0 = u$ y para todo $j \geq 1$, $u_j = u_{j-1} i_j$. Si t es un árbol finito, la* **altura** *de un nodo $u \in dom(t)$ es la mayor longitud de los caminos con origen en u. La* **profundidad** *de t es la altura de su raíz.*

A.5 El lema de König

Definición A.23 *Sea R una relación en un conjunto A. Diremos que y es un* **sucesor inmediato** *de x si xRy. Dados $n > 0$ y $x, y \in A$, escribiremos $xR^n y$ si existen x_i, $0 \leq i \leq n$ tales que $x_i R x_{i+1}$ para todo $0 \leq i \leq n-1$, $x = x_0$ y $x_n = y$. Diremos también que y es* **sucesor** *de x si $xR^n y$ para cierto $n > 0$.*

Definición A.24 *Una relación R sobre un conjunto A se dice* **localmente finita** *si para cualquier $x \in A$, el conjunto de sus sucesores inmediatos es finito. Se dice* **globalmente finita** *si el conjunto de los sucesores de cualquier $x \in A$ es finito.*

Definición A.25 *Sea R una relación sobre un conjunto A. Dado un elemento $x \in A$, definimos el* **conjunto de las longitudes de las derivaciones** *comenzando en x como $L(x) = \{i \in \mathbb{N} : (\exists y)(xR^i y)\}$. Decimos que R es* **acotada** *si para todo $x \in A$, $L(x)$ es un conjunto acotado.*

Es evidente que si existe una R-derivación infinita comenzando en un elemento x, entonces el conjunto de las longitudes de las derivaciones comenzando en x es no acotado. Es decir, se verifica el siguiente lema:

Lema A.26 *Sea R una relación acotada sobre un conjunto A. Entonces R es noetheriana.*

La implicación contraria no es cierta en general, como muestra el siguiente ejemplo:

Ejemplo A.27 Sea $>_\top$ la relación definida sobre $\mathbb{N} \cup \{\top\}$ ($\top \notin \mathbb{N}$), ampliando el orden $>$ usual de $\mathbb{N}$ con $\top > n$ para cualquier $n \in \mathbb{N}$. Es evidente que esta relación es noetheriana, ya que $>$ lo es. Sin embargo existen $>_\top$-derivaciones de longitud arbitraria cuyo primer elemento es $\top$. Por tanto, $L(\top)$ no es acotado.

Sin embargo, cuando la relación es localmente finita, sí se tiene la implicación contraria:

Teorema A.28 *Sea R una relación localmente finita. Entonces R es acotada si y sólo si es noetheriana.*

Demostración:
Por el lema A.26 basta probar que toda relación localmente finita y noetheriana es acotada. Sea $P(x)$ la propiedad "$L(x)$ es un conjunto acotado". Probemos por inducción noetheriana que $P(x)$ se verifica para cualquier $x \in A$ (tiene sentido, pues R es noetheriana). Supongamos que para todo $y \in A$ tal que xRy se verifica $P(y)$. Sea S el conjunto de los sucesores inmediatos de x, que es finito por ser R localmente finita. Si $S = \{y_1, \ldots, y_n\}$, entonces por hipótesis de inducción los conjuntos $L(y_i)$, $1 \leq i \leq n$, están acotados. Sea M_i una cota de $L(y_i)$, $1 \leq i \leq n$. Es evidente que

$$L(x) = \{1 + k : k \in L(y_i), 1 \leq i \leq n\} \cup \{0\}.$$

Por tanto, $L(x)$ está acotado por $1 + M$ donde $M = max\{M_1, \ldots, M_n\}$. □

Lema A.29 *Sea R una relación localmente finita y noetheriana. Entonces R es globalmente finita.*

Demostración:
Es una sencilla aplicación de inducción noetheriana a la propiedad $P(x)$ definida por "el conjunto de sucesores de x es finito". □

Definición A.30 *Una relación R en A se dice* **acíclica** *si no existe $a \in A$ y $n > 0$ tal que $aR^n a$.*

Lema A.31 *Sea R una relación globalmente finita y acíclica. Entonces R es noetheriana.*

Demostración:
Supongamos, por reducción al absurdo, que R es globalmente finita, acíclica y no noetheriana. Por tanto, existe una R-derivación infinita $x_1 R x_2 R x_3 R \cdots$. Luego $\{x_i : i \geq 2\}$ es un subconjunto del conjunto de sucesores de x_1. Como R es globalmente finita, el conjunto $\{x_i : i \geq 2\}$ debe ser finito. Por tanto, existen $i, j \in \mathbb{N}$ tales que $2 \leq i < j$ y $x_i = x_j$, lo cual implica que $x_i R^{j-i} x_i$, que está en contradicción con ser R acíclica. □

Los lemas A.29 y A.31 implican el siguiente resultado que enunciamos como teorema.

Teorema A.32 *Sea R una relación acíclica y localmente finita. Entonces R es globalmente finita si y sólo si es noetheriana.*

Este teorema tiene una reformulación en términos de árboles, que se conoce como el lema de König. Para enunciarlo, vamos a traducir los conceptos definidos para relaciones en los conceptos equivalentes para árboles. Es por ello que usamos este lema, cuya sencilla demostración omitimos:

Lema A.33 *Sea Σ un conjunto y t un Σ-árbol cualquiera. Sea R la relación definida sobre $dom(t)$ de la siguiente manera:*

$$(\forall u, v \in dom(t)) \quad [uRv \Leftrightarrow (\exists i \in \mathbb{N}_+)[v = ui]]$$

(es decir, v es hijo de u). Entonces:

1) R es una relación acíclica.

2) R es localmente finita si y sólo si t está finitamente ramificado.

3) R es globalmente finita si y sólo si t es finito.

4) R es noetheriana si y sólo si t no tiene caminos infinitos.

Este lema nos permite reescribir el teorema A.32 para árboles, de la siguiente manera, resultado conocido como lema de König:

Corolario A.34 (Lema de König). *Sea t un Σ-árbol finitamente ramificado. Entonces t es infinito si y sólo si tiene un camino infinito.*

Apéndice B

Verificación de protecciones y ejecución

Comentamos en este apéndice la verificación de protecciones de algunas de las funciones presentadas a lo largo de los capítulos precedentes. Como ya se comentó en la sección 2.1 (página 24), el mecanismo de las protecciones permite especificar y demostrar que las llamadas a las funciones definidas se producen siempre sobre determinados tipos de argumentos. Esto influye favorablemente sobre la eficiencia de la ejecución de las funciones cuyas protecciones han sido verificadas. En la última sección de este capítulo daremos algunos datos cuantitativos al respecto.

B.1 Protecciones y ejecución eficiente

Desde el punto de vista lógico, las funciones en ACL2 son totales. Esto significa que los axiomas que especifican el comportamiento de una función permiten deducir un valor para cualesquiera argumentos de entrada. Sin embargo, es posible especificar qué tipo de argumentos se espera recibir cuando una función ha de ser ejecutada. Es lo que llamamos el *dominio pretendido* de una función.

El usuario ACL2 puede especificar el dominio pretendido de una función, en el momento de su definición, declarando su *protección*; se trata de una fórmula sobre los argumentos de la función, que especifica las propiedades que deben verificar los objetos del dominio pretendido. Por ejemplo, la protección de la función `car` expresa que su argumento debe ser un par punteado o `nil`. Todas las funciones predefinidas en ACL2 tienen una protección. En particular, las funciones ACL2 que también son Common Lisp tienen definida una protección que refleja el dominio pretendido que se define en el estándar Common Lisp [69]. La especificación de protecciones de funciones definidas por el usuario es opcional.

La *verificación de protecciones* es el proceso de demostrar que una función respeta las protecciones de todas las funciones que usa en su definición, siempre que sus argumentos de entrada respeten la protección de la función. Esto tiene varios objetivos:

- En primer lugar, se trata de un mecanismo de especificación de los argumentos de entrada.
- Si una expresión usa funciones con protecciones verificadas, podemos asegurar que se

devuelve siempre el mismo valor en cualquier implementación de Common Lisp[1], ya que se tiene asegurado que las llamadas a las funciones predefinidas Common Lisp se producen sobre argumentos de su dominio pretendido y por tanto respetando el estándar.

- Una función con protecciones verificadas se puede ejecutar, en general, más eficientemente. Esto es debido a que las comprobaciones de tipo se pueden evitar en tiempo de ejecución. Más adelante abundaremos en este aspecto.

La especificación de la protección de una función se realiza mediante la declaración `:guard` al definirla mediante `defun`. Por ejemplo, en la definición de la función `apply-subst`, especificamos que su protección es `(and (alistp sigma) (term-p-aux flg term))` de la siguiente manera:

```
(defun apply-subst (flg sigma term)
  (declare (xargs :guard (and (alistp sigma)
                              (term-p-aux flg term))))
 ...)
```

La verificación de protecciones de una función provoca un intento de prueba, por parte del demostrador, de las conjeturas que aseguran que en cualquier llamada de la función sobre argumentos que verifican su protección, sólo se producen llamadas de funciones sobre argumentos que verifican sus protecciones. Si se especifica protección de una función, por defecto el demostrador intenta la verificación de la misma. En algunos casos (por ejemplo, en el caso de algunas funciones recursivas), la verificación de protecciones sólo se puede terminar con éxito si previamente se han probado algunos lemas. En este caso, es posible retrasar esta verificación. Véase `guard` y `verify-guard`.

Cuando una función se define en ACL2, se definen al mismo tiempo dos funciones en el sistema Common Lisp sobre el que está construido ACL2. Estas dos definiciones son las que se van a usar cuando la función se ejecute. Por un lado, se define una función Common Lisp con el mismo cuerpo que la función ACL2 (la versión *pura* de la función). Ademas, se define otra función que incluye comprobaciones, durante el tiempo de ejecución, de que las llamadas se producen sobre los dominios pretendidos. Cuando éste sea el caso, se usa la versión pura de la función. En caso contrario, se usan los axiomas lógicos que especifican el comportamiento de las funciones fuera de sus dominios pretendidos.

Es por este motivo que la verificación de protecciones puede mejorar la eficiencia de ejecución de una función ACL2. Cuando la llamada a una función con sus protecciones verificadas se produce sobre argumentos que verifican la protección, la versión pura de la función se utiliza y se deja al sistema Common Lisp sobre el que está construido ACL2 que lleve a cabo la evaluación, sin más comprobaciones posteriores. Si además la función está compilada (vease `:comp`), se consigue un considerablemente aumento en la eficiencia de ejecución.

Otro apunte acerca de la eficiencia de ejecución en ACL2: existen funciones ACL2 que tienen el mismo significado lógico pero diferente protección. El uso de unas u otras funciones es irrelevante desde el punto de vista lógico, pero puede afectar a la eficiencia de ejecución de las funciones que los usan. Veamos algunos ejemplos:

[1] Con los ficheros adecuados cargados previamente.

- Además del predicado `equal`, existen otros predicados que comprueban igualdad. Por ejemplo, `eql` tiene el mismo significado lógico que `equal`, pero la protección para `eql` establece que al menos uno de sus argumentos debería ser número, símbolo o un carácter. La ejecución de `eql` es más eficiente que la de `equal`, siempre y cuando se pueda asegurar (mediante verificación de protecciones) que será llamado sobre argumentos que verifican su protección.

- Esta diferencia entre los distintos tipos de igualdad se transmite también a las funciones que manipulan listas. Por ejemplo `member` usa `eql`, frente a `member-equal` que usa `equal`. Lo mismo ocurre con `assoc` frente a `assoc-equal`.

- La comprobación de parada en funciones definidas recursivamente sobre listas es más eficiente usando `endp` que usando `atom`. La función `endp` tiene una protección más restrictiva, ya que espera que sus argumentos sea listas (es decir, objetos ACL2 que verifican `true-listp`).

Una última cuestión acerca de la eficiencia, aunque no directamente relacionada con la verificación de protecciones. Si una función devuelve varios valores, en general es más eficiente devolverlos usando multivalores que usando listas, ya que tal y como está definida la función `mv`, se evita el coste de construcción de la estructura de lista.

B.2 Verificación de protecciones

Presentamos en esta sección las protecciones que hemos especificado para las principales funciones de la teoría desarrollada en este capítulo. Además describimos brevemente el proceso de verificación de las mismas.

Términos, sustituciones y sistemas propios

La idea principal a la hora de especificar la protección de una función que manipula términos es que, por las razones de eficiencia comentadas en la sección anterior, se prefiere usar `eql` frente a `equal` y `endp` frente a `atom`, siempre que esto sea posible. Es por esto que las protecciones de las funciones definidas especifican objetos ACL2 cuya estructura permite usar estas funciones.

Por ejemplo, la macro `term-p` define la clase de objetos ACL2 que esperan como argumentos de entrada las funciones que manipulan términos. Intuitivamente, `term-p` define aquellos objetos ACL2 que representan términos propios (véase la subsección 3.1.2). Esta macro necesita como función auxiliar la función `term-p-aux` que define recursivamente los términos propios y las listas de términos propios. Los objetos que verifican `term-p` representan términos mediante listas propias (lo que nos permite usar `endp` en lugar de `atom` para recorrer su estructura) y sus símbolos de función y variables verifican el predicado `eqlablep` (lo que permite el uso de `eql` para comprobar la igualdad de símbolos o variables). El lector puede observar el uso de `eql` y de `endp` en las definiciones de los capítulos 3, 4 y 7. Esta es la definición de `term-p-aux` y de `term-p`:

```
(defun term-p-aux (flg x)
  (declare (xargs :guard t))
  (if flg
```

```
      (if (atom x)
          (eqlablep x)
        (and (eqlablep (car x))
             (term-p-aux nil (cdr x))))
    (if (atom x)
        (equal x nil)
      (and (term-p-aux t (car x))
           (term-p-aux nil (cdr x))))))

(defmacro term-p (x)
  (declare (xargs :guard t))
  `(term-p-aux t ,x))
```

De la misma manera podemos definir las sustituciones propias y los sistemas de ecuaciones propios:

```
(defun substitution-p (l)
  (declare (xargs :guard t))
  (if (atom l)
      (equal l nil)
    (and (consp (car l))
         (eqlablep (caar l))
         (term-p (cdar l))
         (substitution-p (cdr l)))))


(defun system-p (S)
  (declare (xargs :guard t))
  (if (atom S)
      (equal S nil)
    (and (consp (car S))
         (term-p (caar S)) (term-p (cdar S))
         (system-p (cdr S)))))
```

Usando estos predicados, especificamos las protecciones de las funciones que tratan con términos, sustituciones y sistemas de ecuaciones. Nótese que las protecciones de estos predicados son `t` en todos los casos: es decir, su dominio pretendido es el conjunto de todos los objetos ACL2.

Algunas funciones y sus protecciones

Obviamente, las funciones cuya finalidad es la de definir propiedades que nos permiten enunciar teoremas, no necesitan ser ejecutadas con eficiencia. Es por esto que no es necesario especificar y verificar sus protecciones. Por ejemplo, las funciones definidas en los capítulos 5 y 6 no tienen protecciones definidas.

En la teoría presentada en esta memoria, se han verificado las protecciones de 67 funciones. Lo que sigue es una lista de las principales funciones definidas junto con sus protecciones.

FUNCIONES	PROTECCIONES
(apply-subst flg sigma term)	(and (alistp sigma) (term-p-aux flg term))
(composition sigma1 sigma2)	(and (alistp sigma1) (substitution-p sigma2))
(position-p pos term)	(term-p term)
(occurrence term pos)	(term-p term)
(replace-term term1 pos term2)	(term-p term)
(match-mv S)	(system-p S)
(subs-mv t1 t2)	(and (term-p t1) (term-p t2))
(subs-list-mv l1 l2)	(and (term-p-aux nil l1) (term-p-aux nil l2))
(number-rename term x y)	(and (term-p term) (acl2-numberp x) (acl2-numberp y))
(number-rename-list l x y)	(and (term-p-aux nil l) (acl2-numberp x) (acl2-numberp y))
(mgs-mv S)	(system-p S)
(mgu-mv t1 t2)	(and (term-p t1) (term-p t2))
(anti-unify t1 t2)	(and (term-p t1) (term-p t2))
(mg-instance t1 t2)	(and (term-p t1) (term-p t2))
(r-reduce term R)	(and (term-p term) (system-p R))
(normal-form-n-steps n term R)	(and (term-p term) (system-p R) (integerp n) (>= n 0))
(cps-trs R)	(system-p R)

Teoremas de clausura y verificación de protecciones

Como ya se ha comentado, el proceso de verificación de la protección de una función exige un trabajo de demostración en el sistema. Se trata de probar que cualquier llamada a la función sobre argumentos que verifiquen su protección, no provoca a su vez llamadas de funciones sobre argumentos que estén fuera de lo especificado por sus respectivas protecciones.

Esto supone, en la mayoría de los casos, demostrar previamente una serie de lemas que aseguren que el resultado devuelto por una función verifica las protecciones de otras funciones. Véamoslo con un ejemplo.

La protección de la función `(mg-instance-mv t1 t2)` (página 165) está especificada por la fórmula `(and (term-p t1) (term-p t2))`. Esta función está definida de manera que renombra los términos que recibe como entrada y cálcula un unificador de máxima generalidad de estos términos renombrados. Como consecuencia de esto, durante el proceso de verificación de protecciones de la función `mg-instance-mv`, será necesario demostrar que la llamada a la función `mgu-mv` provocada por la ejecución de `(mg-instance-mv t1 t2)`, se hace respetando sus protecciones, siempre que `t1` y `t2` verifiquen `term-p`.

Esto significa demostrar que la función `number-rename` devuelve un término propio siempre que recibe un término propio. Es decir, la verificación de protecciones de `mg-instance-mv` hace necesario demostrar previamente el siguiente lema:

```
(defthm number-rename-term-p
  (implies (and (acl2-numberp x) (term-p term))
           (term-p (number-rename term x y))))
```

Este lema es ilustrativo del tipo de lemas que son necesarios para la verificación de protecciones de las funciones ejecutables de esta teoría. El lector puede consultar los libros ACL2 que acompañan a esta memoria, donde encontrará convenientemente indicados aquellos lemas cuya finalidad es la verificación de protecciones.

Como se observa en el ejemplo anterior, estos teoremas son muy similares a los teoremas de clausura que se han visto en los capítulos 3 y 4. Por ejemplo, recuérdese el teorema de clausura de la función `number-rename`, ya presentado en la subsección 4.1.3:

```
(defthm number-rename-term-s-p
  (implies (and (acl2-numberp x) (term-s-p term))
           (term-s-p (number-rename term x y))))
```

De hecho, podemos obtener el teorema `number-rename-term-p` a partir del teorema `number-rename-term-s-p`, usando instanciación funcional. La clave está en observar que la función `term-p-aux` puede ser vista como un caso particular de `term-s-p-aux`, aquélla definida para una signatura concreta (véase la definición de `term-s-p-aux` en la subsección 3.1.2). Esta signatura concreta es la que considera cualquier objeto ACL2 que verifique el predicado `eqlablep`, como un símbolo de función, de aridad variable. Es decir, si en la definición de `term-s-p-aux` sustituimos la función `signat` (la función que define una signatura general), por la expresión lambda `(lambda (x n) (eqlablep x))`, obtenemos la función `term-p-aux`.

Con esta idea, cada vez que se ha demostrado un teorema de clausura para una función, se demuestra mediante instanciación funcional un teorema análogo que será de utilidad

en el proceso de verificación de protecciones. Esta instancia funcional deberá sustituir la función general `signat` por la lambda expresión anterior. Por ejemplo, el teorema `number-rename-term-p` se demuestra de manera inmediata mediante el siguiente consejo:

```
;  :hints (("Goal" :use (:functional-instance
;                        number-rename-term-s-p
;                        (signat (lambda (x n) (eqlablep x)))
;                        (term-s-p-aux term-p-aux)))))
```

La mayoría de los lemas que se necesitan durante la verificación de protecciones se han obtenido a partir de los teoremas de clausura mediante instanciación funcional, a partir de instancias análogas a la de este ejemplo.

B.3 Algunos ejemplos de ejecución

Como ilustración de la ejecución de funciones con protecciones verificadas, presentamos en esta sección algunos ejemplos, junto con alguna información cuantitativa. Los detalles se pueden consultar en el fichero `benchmark.lisp`[2].

Unificación

Comprobamos el algoritmo de unificación de sistemas, ejecutando la función `mgs-mv` (sección 4.4.5) sobre sistemas de la forma:

$$S_n = \{x_n \approx f(x_{n-1}, x_{n-1}), x_{n-1} \approx f(x_{n-2}, x_{n-2}), \ldots, x_1 \approx f(x_0, x_0)\}$$

Una solución idempotente y de máxima generalidad de S_n es:

$$\sigma_n = \{x_1 \mapsto f(x_0, x_0), x_2 \mapsto f(f(x_0, x_0), f(x_0, x_0)), \ldots\},$$

que asigna a cada x_i un arbol binario y completo de altura i. Por tanto, el número de símbolos del rango de σ_n es exponencial respecto a n.

Para efectuar las comprobaciones, definimos la funciones `exp-unif-problem` y `exp--unif`. La primera de ellas construye los problemas S_n y la segunda comprueba su solubilidad mediante `mgs-mv`:

```
(defun exp-unif-problem (n)
  (if (zp n)
      nil
    (cons (cons n (list 'f (1- n) (1- n)))
          (exp-unif-problem (1- n)))))

(defun exp-unif (n)
  (mv-let (sigma bool)
          (mgs-mv (exp-unif-problem n))
          bool))
```

[2]Todos los datos de tiempos de ejecución han sido obtenidos en una máquina con dos procesadores Intel Pentium III a 800 Mhz y 256 Mb de RAM. Tambien se han compilando previamente las funciones (véase `comp`).

La tabla de la figura B.1 muestra los datos de tiempo de ejecución (en segundos) de la expresión `(exp-unif` n`)`, para varios valores de n.

`Tamaño`	Tiempo
$n = 13$	0.1
$n = 14$	0.2
$n = 15$	0.42
$n = 16$	1.9
$n = 17$	3.37
$n = 18$	7.7
$n = 19$	17.57
$n = 20$	45.87

Figura B.1: Unificación de S_n

Para el caso $n = 21$ ya no se obtiene respuesta por falta de recursos. La ejecución de `mgs-mv` sobre S_n necesita una cantidad de espacio exponencial en n, ya que el carácter aplicativo de ACL2 hace que las repeticiones de una variable en un término se representen por distintas copias del mismo objeto. Si la representación de los términos estuviera implementada en forma de grafos, compartiendo posiciones de memoria, la complejidad sería lineal. Comentamos más sobre esta cuestión en el capítulo 8.

Formas normales

Veamos ahora algunos ejemplos de ejecución de la función `normal-form-n-steps` (sección 7.3.4) para el cálculo de formas normales. Para ello calculamos la forma normal, respecto del sistema $\mathcal{R}_G$ definido en el ejemplo 7.15, de los términos t_n definidos como sigue:

$$t_n = i(x_1 * (x_2 * (x_3 * \ldots (x_{n-1} * x_n)))) * (\ldots (((x_1 * x_2) * x_3) \ldots) * x_n)$$

Las siguientes funciones implementan las herramientas necesarias para construir en ACL2 los términos t_n. Nótese el uso de la función de renombrado:

```
(defun nf-term-i (n)
  (cond ((zp n) nil)
        ((= n 1) 1)
        ((= n 2) '(* 2 1))
        (t (list '* n (nf-term-i (1- n))))))

(defun nf-term-d (n)
  (cond ((zp n) nil)
        ((= n 1) 1)
        ((= n 2) '(* 1 2))
        (t (list '* (nf-term-d (1- n)) n))))

(defun nf-term (n)
  (list '* (number-rename (list 'i (nf-term-i n)) 1 1)
        (nf-term-d n)))
```

De esta manera, `(nf-término` n`)` construye t_n. Nótese que la fórma normal de t_n respecto de $\mathcal{R}_G$ es la identidad e. Usamos la función `normal-form-n-steps` para el cálculo de la forma normal, aplicando un número suficientemente grande de pasos de reducción. En concreto, si `(RG)` es la representación en ACL2 del sistema $\mathcal{R}_G$ (ejemplo 7.18), ejecutamos lo siguiente:

```
(normal-form-n-steps 10000 (nf-term n)  (RG))
```

La tabla de la figura B.2 muestra los datos de tiempo de ejecución (en segundos) de esta expresión, para varios valores de n.

Tamaño	Tiempo
$n = 50$	0.23
$n = 100$	0.79
$n = 200$	3.31
$n = 300$	7.41
$n = 500$	20.92
$n = 1000$	85.17

Figura B.2: Forma normal de t_n

Cálculo de pares críticos

La siguiente función comprueba si todos los pares de términos de un sistema de ecuaciones `S` tienen forma normal común respecto de un SRT `R`, siempre que estas formas normales se tengan en un número de pasos de reducción menor que un cierto número dado `n`:

```
(defun common-n-f-list-n-steps (n S R)
  (if (endp S)
      t
    (mv-let (nf-1 bool1)
            (normal-form-n-steps n (caar S) R)
            (mv-let (nf-2 bool2)
                    (normal-form-n-steps n (cdar S) R)
                    (if (and bool1 bool2)
                        (if (equal nf-1 nf-2)
                            (common-n-f-list-n-steps n (cdr S) R)
                          (list nf-1 nf-2))
                      nil)))))
```

Nótese que la función devuelve `nil` cuando en algún intento de cálculo de forma normal el número de pasos `n` no es suficiente. Devuelve `t` si todos los pares de `S` tiene forma normal común, obtenidas en menos de `n` pasos. En caso contrario, devuelve el par de formas normales correspondientes al primer par de términos de `S` que no tenga forma normal común (respecto de `R`).

Permitiendo un número de pasos de reducción suficientemente grande, podemos usar la función anterior en conjunción con `cps-trs` (subsección 7.5.1), para comprobar la confluencia local de un sistema de reescritura dado. Es lo que hace la siguiente función:

```
(defun local-confluence-n-steps (R n)
  (common-n-f-list-n-steps n (cps-trs R) R))
```

Para obtener varios ejemplos de ejecución de esta función, hemos calculado su valor actuando sobre cada uno de los sistemas de reescritura que en las referencias [44] y [49] se presentan como sistemas de reescritura completos. En la tabla de las figura B.3 resumimos los resultados.

Ejemplo en [44]	Pares críticos	Confluencia Local
1	65	Sí
4	13	Sí
5	67	Sí
6	21	Sí
7	3	Sí
9 (1)	2	Sí
9 (2)	22	Sí
10	50	Sí
12	73	Sí
13	125	No
14	177	Sí
15	175	Sí
16	87	Sí
17	339	No
Ejemplo en [49]	Pares críticos	Confluencia local
1	78	Sí
2	80	Sí
3	91	Sí
4	101	Sí
5	99	Sí
6	66	Sí
7	199	No
8	169	No
9	145	Sí
12	12	Sí
13	47	No
15	4	Sí
16 (1)	67	Sí
16 (2)	37	Sí
17 (1)	16	Sí
17 (2)	8	Sí
18 (1)	6	Sí
18 (2)	32	Sí

Figura B.3: Confluencia local en los ejemplos de [44] y de [49]

Los tiempos de ejecución en todos los casos son inapreciables, por lo que no los hemos incluido en la tabla. Como dato cuantitativo, hemos incluido el número total de pares críticos calculados en cada caso. Además hemos añadido una columna para indicar si

nuestra función reconoce el SRT correspondiente como localmente confluente. Sorprendentemente, algunos de los sistemas que se presentan como completos en las dos referencias anteriores no lo son. En el capítulo 8 comentaremos más sobre esta cuestión.

Apéndice C

Algunos detalles sobre las demostraciones automáticas

En este apéndice explicamos algunos detalles (principalmente de carácter técnico) relativos a la demostración automática de los resultados presentados a los largo de la memoria.

C.1 Términos y sustituciones

C.1.1 Los términos como árboles

La mayoría de las propiedades sobre la estructura de árbol de los términos (subsección 3.2.2) se demuestran en ACL2 sin ayuda por parte del usuario, por inducción en la longitud de las posiciones de los términos. Por ejemplo, en la prueba del teorema `occurrence-instance`, el demostrador escoge un esquema de inducción análogo al que se usa en la prueba a mano dada en 3.25. Así, el paso de inducción de tal esquema es:

```
(IMPLIES (AND (NOT (ENDP POS))
              (NOT (VARIABLE-P TERM))
              (INTEGERP (CAR POS)) (< 0 (CAR POS))
              (<= (CAR POS) (LEN (CDR TERM)))
              (:P (CDR POS) SIGMA (NTH (+ -1 (CAR POS)) (CDR TERM))))
         (:P POS SIGMA TERM))
```

Este paso de inducción se corresponde formalmente con la frase *"podemos suponer, como hipótesis de inducción para* t_i *y* $q \in \mathcal{P}(t_i)$*, que* $\sigma(t_i/q) = \sigma(t_i)/q$*"* de la demostración a mano.

En el caso de la demostración de las propiedades relativas a las posiciones disjuntas, el sistema no escoge el esquema de inducción adecuado. Esto hace que sea necesario proporcionar el siguiente esquema de inducción, inspirado en la prueba a mano:

```
(defun induct-position-p-disjoint (pos1 pos2 term)
  (cond
    ((variable-p term) t)
    ((and (consp pos1) (consp pos2))
     (if (equal (car pos1) (car pos2))
```

```
        (induct-position-p-disjoint
           (cdr pos1) (cdr pos2) (nth (- (car pos1) 1) (cdr term)))
       t))
   (t t))))
```

C.1.2 Equiparación y subsunción

Describimos en esta subsección algunos aspectos a destacar sobre la demostración automática de los resultados presentados en la sección 3.3, donde se verifica un algoritmo de equiparación de términos.

La función de selección

Nótese que la función `transform-subs-sel`, tal y como está implementada, selecciona una ecuación del primero de los sistemas y coloca el resultado de transformar la ecuación *al principio* del segundo sistema (siempre que no detecte fallo). De esta manera se produce una cierta "asimetría" entre el par de sistemas original y el transformado, que complica el razonamiento automático.

Cuando se está razonado sobre el concepto de equiparador (implementado por la función `matcher`), es posible usar una combinación de congruencia y regla de reescritura para solventar esta dificultad:

```
(defcong equal-set iff (matcher x y) 2)

(defthm equal-set-selection-and-eliminate
  (implies (consp (car S-match))
           (equal-set (car S-match)
                      (cons (a-pair (car S-match))
                            (eliminate (a-pair (car S-match)
                                       (car S-match)))))))
```

Este regla, junto con la congruencia, permite al demostrador reescribir el primero del par de sistemas sobre el que se aplica la transformación, de manera que la ecuación seleccionada queda al principio[1]. Debido a la congruencia definida, esto se puede hacer cuando tal sistema figura como segundo argumento de la función `matcher` (en otras palabras, cuando se está razonando sobre los equiparadores del sistema). Nótese que la regla sólo está definida para sistemas expresados de la forma `(car S-match)`, lo cual evita problema de no terminación de la regla de reescritura.

Esta misma táctica basada en congruencias no se puede usar para razonar sobre el número de símbolos del sistema, ya que `equal-set` no es una congruencia respecto de `length-system` (el número de símbolos puede ser menor después de aplicar `eliminate`, que borra *todas* las ocurrencias de un elemento). Las dos reglas de reescritura siguientes permiten aplicar una táctica análoga, aún sin usar congruencias.

```
(defthm length-system-selection-and-delete-one
  (implies (consp (car S-match))
           (equal (length-system (car S-match))
```

[1] Desde el punto de vista del razonamiento, se entiende.

```
                  (length-system
                   (cons (a-pair (car S-match))
                         (delete-one (a-pair (car S-match))
                                     (car S-match)))))))

(defthm length-system-eliminate-delete-one-x
  (<= (length-system (eliminate x S)) (length-system (delete-one x S)))
  :rule-classes :linear)
```

La demostración de los invariantes de las transformaciones

La distinción de casos que tiene la definición de `transform-subs-sel` hace que, en la prueba que intenta el demostrador, las propiedades invariantes en las transformaciones se demuestren mediante una distinción de casos totalmente análoga a la de la prueba a mano: cada uno de los invariantes se prueba para cada una de las posibles reglas de transformación. En algunos casos, se necesitan lemas previos que ayudan a resolverlos.

Una vez probadas las propiedades fundamentales de las reglas de transformación, es necesario deshabilitar la definición de `transform-subs` (función no recursiva) para que las propiedades demostradas se puedan usar como reglas de reescritura.

Los invariantes de `transform-subs-sel` se trasladan fácilmente a la función `subs--system-sel` (que aplica iterativamente las reglas de transformación). Para ello, el sistema genera un intento de prueba por inducción en la reducción $\Rightarrow_s$[2], como sugiere el siguiente esquema de inducción, generado automáticamente por el sistema y justificado por la prueba de terminación de la función `subs-system-sel`:

```
(AND (IMPLIES (AND (NOT (NORMAL-FORM-SYST S-MATCH))
                   (:P (TRANSFORM-SUBS-SEL S-MATCH) SIGMA))
              (:P S-MATCH SIGMA))
     (IMPLIES (NORMAL-FORM-SYST S-MATCH)
              (:P S-MATCH SIGMA))).
```

Finalmente, las propiedades de `subs-system-sel` se trasladan fácilmente a la función `match-sel`, mediante los correspondientes consejos de instanciación.

C.2 El retículo de los términos de primer orden

C.2.1 Renombrados

En esta subsección comentamos algunas cuestiones sobre la demostración automática de los resultados presentados en la subsección 4.1.3.

Generalización en la verificación del predicado `renamed`

Es interesante destacar cómo a veces un intento de prueba en el demostrador puede ser considerablemente simplificado si se escoge una generalización adecuada del resultado que se quiere probar. Lo ilustramos explicando la demostración del lema `renamed-implies--setp-codomain` (página 118). Recuérdese que a partir del siguiente lema.

[2]También puede verse como una inducción en la longitud de la $\Rightarrow_s$-derivación.

```
(defthm renamed-implies-injective-val
  (implies (and (equal (apply-subst flg delta
                                     (apply-subst flg sigma term))
                       term)
                (member x (variables flg term))
                (member y (variables flg term))
                (not (equal x y)))
           (not (equal (val x sigma) (val y sigma)))))
```

concluíamos que no existen elementos repetidos en el codominio de la sustitución resultante de restringir `sigma` a las variables de `term`. Es decir:

```
(defthm renamed-implies-setp-codomain
  (implies (equal (apply-subst flg delta
                               (apply-subst flg sigma term))
                  term)
           (setp (co-domain (normal-form-subst flg sigma term)))))
```

El demostrador no es capaz de demostrar esta conclusión directamente, aún habiendo probado previamente `renamed-implies-injective-val`. Después de aplicar inducción en la estructura de los términos, queda por probar la siguiente conjetura, correspondiente a uno de los casos inductivos:

```
; (IMPLIES (AND (SETP (CO-DOMAIN (RESTRICTION SIGMA (MAKE-SET VS0))))
;               (SETP (CO-DOMAIN (RESTRICTION SIGMA (MAKE-SET VS)))))
;          (SETP (CO-DOMAIN
:                   (RESTRICTION SIGMA (MAKE-SET (APPEND VS0 VS))))))).
```

Parece difícil establecer una relación entre las hipótesis y la conclusión, máxime cuando el lema en el que nos tenemos que basar es `renamed-implies-injective-val`. El problema es que las hipótesis de inducción no son lo suficientemente generales para que puedan ser aplicadas fácilmente.

Sin embargo, es posible probar previamente un lema del cual luego el teorema `renamed--implies-setp-codomain` será un caso particular. En este lema, hemos usado `subsetp` para generalizar las hipótesis convenientemente con la intención de que en el intento de prueba se pueda usar el lema `renamed-implies-injective-val`:

```
(defthm renamed-implies-setp-codomain-main-lemma
  (implies (and
            (equal (apply-subst flg delta (apply-subst flg sigma term))
                   term)
            (setp l)
            (subsetp l (variables flg term)))
           (setp (co-domain (restriction sigma l))))
  :hints (("Goal" :induct (setp l))
          ("Subgoal *1/2'''" :induct (len l2))))
```

Al intentar probar este lema, el demostrador intenta una inducción sugerida por la definición `setp`, tal y como se le indica. El caso inductivo se reduce ahora a la siguiente conjetura:

```
; (IMPLIES (AND (NOT (MEMBER L1 L2))
;               (SETP (CO-DOMAIN (RESTRICTION SIGMA L2)))
;               (EQUAL (APPLY-SUBST FLG DELTA (APPLY-SUBST FLG SIGMA TERM))
;                      TERM)
;               (SETP L2)
;               (MEMBER L1 (VARIABLES FLG TERM))
;               (SUBSETP L2 (VARIABLES FLG TERM)))
;          (NOT (MEMBER (VAL L1 SIGMA)
;                       (CO-DOMAIN (RESTRICTION SIGMA L2))))).
```

Nótese que la formulación de tal conjetura se acerca ya al lema `renamed-implies--injective-val`, ya que aparece el predicado `member`. Esta es la clave del éxito de la generalización escogida. Al probar esta conjetura por inducción en la longitud de `l2`, tal y como se le indica al demostrador, se puede aplicar `renamed-implies-injective-val` y se consigue demostrar el lema `renamed-implies-setp-codomain-main-lemma`. Finalmente, nótese que el teorema que se pretendía demostrar, `renamed-implies-setp-codomain`, es un caso particular que se obtiene mediante la siguiente instanciación:

```
; :hints (("Goal" :use
;          (:instance renamed-implies-setp-codomain-main-lemma
;                     (l (make-set (variables flg term))))))))))
```

Invariantes en `number-rename-aux`

La prueba del teorema `term-subsumes-number-renamed-aux-term` (página 122) se obtiene mediante la especificación de una serie de propiedades que permanecen invariantes durante el proceso recursivo que realiza `number-rename-aux`.

Este teorema establece que la sustitución que devuelve `number-rename-aux` aplicada al término (o lista de términos) de entrada, es igual al término devuelto por `number-rename--aux`. Para la prueba de esta conjetura, el demostrador intenta una prueba por inducción en la estructura de los términos, tal y como sugiere la función `number-rename-aux`. Uno de los casos inductivos se reduce a la siguiente fórmula[3]:

```
; (IMPLIES
;  (AND
;   (EQUAL (INSTANCE
;             TERM1
;             (SECOND (NUMBER-RENAME-AUX T TERM1 SIGMA X Y)))
;          (FIRST (NUMBER-RENAME-AUX T TERM1 SIGMA X Y)))
;   ...)
;  (EQUAL (INSTANCE
;           TERM1
;           (SECOND (NUMBER-RENAME-AUX
;                       NIL TERM2
;                       (SECOND (NUMBER-RENAME-AUX T TERM1 SIGMA X Y))
;                       X Y)))
;         (FIRST (NUMBER-RENAME-AUX T TERM1 SIGMA X Y))))
```

[3]Para mejorar la legibilidad, se han dejado algunas macros sin expandir y eliminado algunas hipótesis irrelevantes.

Generalizando, para probar con éxito esta conjetura, habría que demostrar previamente que al realizar el proceso de renombrado de un término (o lista de términos) `t2`, partiendo de una sustitución que previamente se ha calculado mediante el proceso de renombrado para otro término (o lista de términos) `t1`, la sustitución de renombrado que finalmente se obtiene actúa sobre `t1` como la de partida. Es lo que expresa el siguiente teorema:

```
(defthm number-rename-incremental
  (implies (alistp sigma)
           (equal (apply-subst
                   flg1
                   (second
                     (number-rename-aux flg2 t2
                            (second (number-rename-aux flg1 t1 sigma x1 y1))
                            x2 y2))
                   t1)
                  (apply-subst
                   flg1
                   (second (number-rename-aux flg1 t1 sigma x1 y1))
                   t1))))
```

Este teorema se demuestra si antes probamos que la sustitución que va calculando `number-rename-aux` se va extendiendo en cada paso recursivo. Es decir, en cada paso se mantienen las ligaduras realizadas y eventualmente se añaden nuevas ligaduras.

Necesitaremos además probar que todas las variables del término que se renombra forman parte del dominio de la sustitución finalmente calculada y, finalmente, para justificar el uso de `assoc` en lugar de `val`, nos hará falta probar que la sustitución que se va calculando es una lista de asociación, siempre que la de partida lo sea. Resumiendo, hay que probar que en cada paso recursivo de la función `number-rename-aux` se mantienen las siguientes propiedades *invariantes*:

- la sustitución que se va calculando (su segundo argumento) es una lista de asociación,
- cuyo dominio contiene al dominio de la sustitución de partida,
- que contiene también al conjunto de variables del término que se renombra,
- y extiende a la sustitución de partida.

Estas propiedades quedan formalizadas por el siguiente teorema:

```
(defthm number-rename-invariants
  (let ((number-renaming-aux
         (second (number-rename-aux flg t1 sigma x y))))
    (implies (alistp sigma)
             (and
              (alistp number-renaming-aux)
              (subsetp (domain sigma) (domain number-renaming-aux))
              (extension number-renaming-aux sigma)
              (subsetp (variables flg t1) (domain number-renaming-aux))))))
```

La cuestión importante aquí es que estas propiedades se prueban *todas a la vez*. Aunque es posible probarlas por separado (siguiendo un determinado orden) resulta mucho más sencillo hacerlo de esta manera. La razón estriba en que las hipótesis de inducción correspondientes a los pasos recursivos contienen explícitamente todas estas propiedades. Así, las hipótesis de inducción se refuerzan, lo cual es positivo desde el punto de vista de la automatización de la demostración. Esta técnica consistente en localizar una serie de propiedades invariantes en el proceso recursivo y demostrarlas a la vez se usa repetidas veces en el desarrollo de esta teoría.

Una vez demostrado el teorema `number-rename-invariants`, la demostración del teorema `number-rename-incremental` es sencilla, sin más que usar apropiadamente el lema `coincide-in-term` (subsección 3.1.3), concluyéndose finalmente el teorema `term--subsumes-number-renamed-aux-term`. Véase el libro `renamings.lisp` para más detalles.

C.2.2 Un algoritmo de unificación basado en reglas de transformación

Todos los comentarios que se han hecho en C.1.2 sobre la interacción con el demostrador para probar las propiedades del algoritmo de equiparación son aplicables al caso de la unificación, ya que la técnica usada es totalmente análoga. Aunque la prueba del algoritmo de unificación necesitó un número considerablemente mayor de lemas auxiliares, el hecho de que ambos algoritmos estén especificados por reglas de transformación similares hace que en este caso nos enfrentemos, aunque a mayor escala, con los mismos problemas, que se resuelven usando técnicas similares. Recordemos algunas de ellas.

La función de selección

Para simplificar la complejidad adicional que supone el que una vez seleccionada una ecuación, el resultado de transformarla se coloque al principio del sistema transformado usamos el mecanismo de reescritura respecto de una congruencia. Por ejemplo, podemos probar que el concepto de solución de un sistema no depende del orden en el cual las ecuaciones aparezcan en el sistema, ni del número de veces que se repitan:

```
(defcong equal-set iff (solution sigma s) 2)
```

La siguiente regla de reescritura permite entonces al demostrador colocar la ecuación seleccionada al principio del sistema, siempre que esté razonando sobre el concepto definido por `solution`, reparando así la "asimetría" existente entre el par de sistemas original y su transformado:

```
(defthm select-eliminate-and-cons-equal-set-instance
  (let* ((S (first S-sol)) (ecu (sel S)))
    (implies (and (consp S-sol) (consp S))
             (equal-set S (cons ecu (eliminate ecu S))))))
```

La demostración de los invariantes de las reglas de transformación

La demostración de cada uno de los invariantes la realiza el sistema expandiendo la definición de `transform-mm-sel` y por tanto considerando cada una de las reglas por separado. Para algunas de las propiedades y reglas de transformación, se necesitarán lemas auxiliares

que ayuden a finalizar con éxito la prueba. Una vez probadas las propiedades fundamentales de las reglas de transformación implementadas por `transform-mm-sel`, es necesario deshabilitar su definición, para que estas propiedades puedan ser usadas como reglas de reescritura en la prueba de las propiedades sobre `solve-system-sel`.

Como en el caso de la equiparación, la prueba de las propiedades de `solve-system-sel` se tienen por una sencilla inducción en la longitud de la secuencia de transformaciones, que es justamente el esquema inductivo sugerido automáticamente por la misma función:

```
;(AND (IMPLIES (AND (NOT (NORMAL-FORM-SYST S-SOL))
;                   (:P (TRANSFORM-MM-SEL S-SOL)))
;              (:P S-SOL))
;     (IMPLIES (NORMAL-FORM-SYST S-SOL)
;              (:P S-SOL))).
```

Finalmente, las propiedades de `mgs-sel` se obtienen a partir de las propiedades de `solve-system-sel` mediante los correspondientes consejos de instanciación. Lo mismo ocurre con las propiedades de `unifiable-sel` y de `mgu-sel`.

El uso de la completitud de `subs-subst`

Como hemos comentado al describir la demostración, el lema `mgs-sel-most-general--solution-main-lemma` (página 159) nos permitía concluir `mgs-sel-most-general-solution`, estableciendo que la sustitución obtenida por `mgs-sel` subsume a cualquier otra solución del sistema que recibe como entrada.

El lema establece que si `sigma` es solución de `S`, entonces la composición de la solución obtenida por `mgs-sel` con `sigma` actúa sobre cualquier término como la propia `sigma`. Se trata de un caso particular de la situación que se asume en el enunciado del teorema `subs--subst-completeness`, uno de los teoremas que nos servían para verificar que `subs-subst` implementaba la subsunción entre sustituciones. El hecho de no poder usar cuantificación universal en las hipótesis de un teorema nos obligaba a formular dichas hipótesis usando un encapsulado. Véase la subsección 3.4.3. Recuérdese que en este caso la hipótesis asumida era la siguiente (donde `(sigma-w)`, `(gamma-w)` y `(delta-w)` representan aquí tres sustituciones concretas, pero arbitrarias):

```
; (defthm sigma-w-delta-w-subsumption-hypothesis
;   (equal (instance term (composition (gamma-w) (sigma-w)))
;          (instance term (delta-w))))
```

Recuérdese también que la conclusión de este teorema de completitud era:

```
; (defthm subs-subst-completeness
;   (subs-subst (sigma-w) (delta-w)))
```

Para poder aplicar este teorema de completitud a la demostración del teorema `mgs--sel-most-general-solution` y concluir que siempre que `sigma` sea solución de `S` se tiene que `(subs-subst (first (mgs-sel S)) sigma)`, podemos combinar el resultado de completitud sobre `subs-subst` con el teorema `mgs-sel-most-general-solution-main--lemma`, usando instanciación funcional.

En un primer momento se podría pensar que la siguiente instancia funcional valdría para nuestros propósitos:

```
;  ((:functional-instance
;    subs-subst-completeness
;    (sigma-w (lambda () (first (mgs-sel S))))
;    (gamma-w (lambda () sigma))
;    (delta-w (lambda () sigma))))
```

Es decir, las sustituciones `(sigma-w)`, `(gamma-w)` y `(delta-w)` se sustituyen respectivamente por `(first (mgs-sel S))`, `sigma` y `sigma`. Sin embargo, esta sustitución funcional no conduce a la demostración del teorema, ya que para poder usar la correspondiente instancia funcional de `subs-subst-completeness` se intenta comprobar que la misma instancia funcional del teorema `sigma-w-delta-w-subsumption-hypothesis` es cierta. Y esto se hace *incondicionalmente*. Es decir, sin suponer `(solution sigma S)` como hipótesis y por tanto, sin poder aplicar el lema `mgs-sel-most-general-solution-main-lemma`. La siguiente instancia funcional es la correcta en este caso:

```
;   ((:functional-instance
;     subs-subst-completeness
;     (sigma-w (lambda ()
;                (if (solution sigma S) (first (mgs-sel S)) nil)))
;     (gamma-w (lambda ()
;                (if (solution sigma S) sigma nil)))
;     (delta-w (lambda ()
;                (if (solution sigma S) sigma nil)))))))))
```

Es decir, en el caso el que se verifique `(solution sigma S)`, la sustitución funcional es la anteriormente indicada y permite usar el lema `mgs-sel-most-general-solution--main-lemma` para demostrar las restricciones impuestas por `sigma-w-delta-w-subsumption-hypothesis`. En caso contrario, se asigna `nil` a cada una de las sustituciones. Para esta asignación las restricciones son trivialmente ciertas. En cualquier caso podemos usar la correspondiente instancia funcional de `subs-subst-completeness` para concluir el teorema deseado.

En [37] se pueden ver otros ejemplos de esta misma técnica en el uso de la regla de inferencia de instanciación funcional para la demostración de teoremas en ACL2.

Instanciación funcional en la verificación de `mgu-mv`

Las propiedades del algoritmo `mgu-mv` (subsección 4.4.5) se obtienen de manera sencilla mediante instanciación funcional de las propiedades del algoritmo no determinista, dadas en 4.4.3. Por ejemplo, la completitud del algoritmo, teorema `mgu-completeness`, se obtiene mediante la siguiente instancia funcional del teorema `unifiable-sel-completeness`:

```
;    ((:functional-instance
;      unifiable-sel-completeness
;      (sel find-not-unifiable)
;      (transform-mm-sel transform-mm-bridge)
;      (solve-system-sel solve-system-bridge)
;      (mgs-sel mgs-mv-bridge)
;      (unifiable-sel unifiable-bridge))))))
```

Como se observa, la función de selección `sel` queda ahora instanciada por la función `find-not-unifiable`. Como también ocurría en el caso del algoritmo de equiparación, las funciones que implementan el algoritmo no determinista no se pueden instanciar directamente por sus correspondientes ejecutables, debido a discordancia de signatura: éstas usan multivalores y múltiples argumentos mientras que aquéllas usan pares punteados tanto en la entrada como en la salida. Por tanto es necesario definir una serie de funciones "puente" (véase la subsección 3.3.5).

Debemos destacar que la verificación de `mgu-mv` constituye un nuevo ejemplo de razonamiento compuesto, en el que la mayor parte del razonamiento se ha hecho sobre la función `mgs-sel`, en la que no hemos tenido en cuenta ciertos detalles que mejorarían su eficiencia pero que complicarían las demostraciones. Las definiciones puente y el uso de instanciación funcional nos permiten trasladar las propiedades de la versión general a la versión mejorada.

C.3 Multiconjuntos y pruebas de terminación

C.3.1 Una versión iterativa de la función de Ackermann

La terminación de `ack-it-aux` (sección 5.4) se obtiene demostrando que la medida asociada es un multiconjunto de pares de números naturales y además que en cada llamada recursiva, el multiconjunto que `measure-ack-it-aux` asocia es menor, respecto de `mul-rel-ack`, que el multiconjunto correspondiente a la llamada original. Expandiendo la definición de `ack-it-aux`, se genera un caso por cada una de las llamadas recursivas. Debemos, por tanto, probar un lema previo que asegure el decrecimiento de la medida, por cada uno de los casos. Es exactamente lo que se hace en la prueba informal presentada en la sección 5.4. Los lemas que hemos probado son los siguientes, para los casos 1, 2 y 3, respectivamente:

```
(defthm measure-ack-it-aux-mp-decreases-1
  (implies (zp y)
           (mul-rel-ack (measure-ack-it-aux l (+ 1 z))
                        (measure-ack-it-aux (cons y l) z))))

(defthm measure-ack-it-aux-mp-decreases-2
  (implies (and (not (zp y)) (zp z))
           (mul-rel-ack (measure-ack-it-aux (cons (+ -1 y) l) 1)
                        (measure-ack-it-aux (cons y l) z))))

(defthm measure-ack-it-aux-mp-decreases-3
  (implies (and (not (zp y)) (not (zp z)))
           (mul-rel-ack (measure-ack-it-aux (list* y (+ -1 y) l) (+ -1 z))
                        (measure-ack-it-aux (cons y l) z))))
```

Los tres lemas se prueban sin necesidad de ningún lema previo específico, salvo aquellos que vienen incluidos en el libro `multiset.lisp`. En la prueba de los tres lemas es fundamental la presencia de la regla `multiset-diff-meta`, regla `:meta` descrita en la subsección 5.2.6. Veamos, a modo de ilustración, cómo el demostrador usa la regla para

simplificar en la prueba del último lema (caso 3). Por ejemplo, expandiendo la definición de `mul-rel-ack`, `zp`, `measure-ack-it-aux` y `get-pairs-add1-0`, una de las conjeturas generadas es la siguiente:

```
(IMPLIES (AND (INTEGERP Y) (< 0 Y)
              (INTEGERP Z) (< 0 Z))
         (FORALL-EXISTS-REL-ACK-BIGGER
              (MULTISET-DIFF (LIST* (CONS Y (+ -1 Z))
                                    (CONS (+ 1 -1 Y) 0)
                                    (GET-PAIRS-ADD1-0 L))
                             (CONS  (CONS Y Z)
                                    (GET-PAIRS-ADD1-0 L)))
              (MULTISET-DIFF (CONS  (CONS Y Z)
                                    (GET-PAIRS-ADD1-0 L))
                             (LIST* (CONS Y (+ -1 Z))
                                    (CONS (+ 1 -1 Y) 0)
                                    (GET-PAIRS-ADD1-0 L)))))
```

Las dos llamadas a `multiset-diff` que aparecen en esta conjetura son ambas argumentos del predicado `forall-exists-rel-ack-bigger`. Debido a las congruencias que se han probado al hacer la llamada a `defmul`, es posible reescribir estos argumentos a expresiones que sean equivalentes respecto de `equal-set`. Nótese además que ambas llamadas a `multiset-diff` tienen precisamente la forma de las expresiones que es capaz de reescribir la regla `multiset-diff-meta` (para $k = 2$, $n = 1$ en la primera y $k = 1$, $n = 2$ en la segunda). Puesto que dicha regla reescribe a expresiones equivalentes respecto de `equal-set`, puede aplicarse para obtener[4]:

```
(IMPLIES (AND (INTEGERP Y) (< 0 Y)
              (INTEGERP Z) (< 0 Z))
         (FORALL-EXISTS-REL-ACK-BIGGER
           (MULTISET-DIFF (LIST (CONS Y (+ -1 Z)) (CONS (+ 1 -1 Y) 0))
                          (LIST (CONS Y Z)))
           (MULTISET-DIFF (LIST (CONS Y Z))
                          (LIST (CONS Y (+ -1 Z)) (CONS (+ 1 -1 Y) 0)))))
```

Es posible simplificar aún más la conjetura, aplicando los lemas `list-multiset-diff--1` y `list-multiset-diff-2` (ver subsección 5.2.6) quedando:

```
(IMPLIES (AND (INTEGERP Y) (< 0 Y)
              (INTEGERP Z) (< 0 Z))
         (FORALL-EXISTS-REL-ACK-BIGGER
              (LIST (CONS Y (+ -1 Z)) (CONS Y 0))
              (LIST (CONS Y Z))))
```

Finalmente, expandiendo la definición de `forall-exists-rel-ack-bigger` y usando aritmética lineal se simplifica la conjetura a `t`, por lo que queda probada.

[4]En realidad, el demostrador sólo muestra la conjetura anterior, que simplifica directamente a `t`

Una vez se ha admitido la definición de `ack-it-aux` y por consiguiente la de `ack-it`, también hemos probado que `ack-it` define la misma función que `ack`. Basta para ello probar que durante la computación que realiza `ack-it-aux` se cumple el siguiente invariante: `(ack-it-aux S z) = (ack` s_k `(ack` s_{k-1}`... (ack` s_1 z`)))`, donde `S`$= (s_1 \dots s_k)$. Formalmente:

```
(defun ack-stack (S z)
  (if (endp S) z (ack-stack (cdr S) (ack (car S) z))))

(defthm ack-it-aux-ack-stack
   (equal (ack-it-aux S z) (ack-stack S z)))
```

Esta última regla de reescritura es suficiente para que el demostrador pruebe la equivalencia entre `ack` y `ack-it`, que es el caso particular en el que `S` y `z` son, respectivamente, `(list n)` y `m`:

```
(defthm ack-it-equal-ack
   (equal (ack-it m n) (ack m n)))
```

C.3.2 La función 91 de McCarthy

Para probar la terminación de `mc-it-aux` (sección 5.5), basta probar que la función de medida `measure-mc-it-aux` obtiene siempre multiconjuntos y que el multiconjunto que asocia es menor, respecto de `mul-rel-mc`, que el multiconjunto asociado a la llamada original.

Sorprendentemente, una sóla regla de reescritura es suficiente (además de las que aporta el libro `multiset.lisp` incluido por `defmul.lisp`) para obtener una prueba automática de la terminación de `mc-it-aux`:

```
(defthm measure-mc-aux-expand
  (implies (and (not (zp n)) (integerp z))
           (equal (measure-mc-aux n z)
                  (cons z
                    (measure-mc-aux (- n 1)
                                    (if (> z 100) (- z 10) 91))))))
```

Dicha regla de reescritura cumple dos cometidos. En primer lugar, la presencia de `if` hace que de manera automática se produzca una distinción de casos muy similar a la que se hace en la prueba a mano. Por otro lado, la regla hace que las llamadas a `(measure-mc-aux` n z`)` se expandan cuando el demostrador puede probar que $n > 0$, obteniéndose así, de manera explícita, las diferencias entre el multiconjunto asociado a la llamada original y el asociado a la llamada recursiva.

Por ejemplo, una de las conjeturas generadas por el demostrador, después de haber aplicado la regla `measure-mc-aux-expand`, es la siguiente:

```
(IMPLIES (AND (INTEGERP N) (< 0 N) (INTEGERP Z)
              (<= Z 100) (<= (+ 11 Z) 100))
         (MUL-REL-MC (LIST* (+ 11 Z)
                            91 (MEASURE-MC-AUX (+ -1 N) 91))
                     (CONS Z (MEASURE-MC-AUX (+ -1 N) 91)))).
```

Nótese que este caso es el correspondiente a $z < 90$. Es decir, se trata del caso 3 de la prueba a mano. Y además, aparece de manera explícita la diferencia entre los multiconjuntos que se comparan (es decir el elemento `Z` ha sido reemplazado por `(+ 11 Z)` y `91`. De manera muy similar a la ya explicada en la subsección C.3.1 para la función de Ackermann, las reglas aportadas por `multiset.lisp` permiten, mediante simplificación, la prueba de la conjetura (en la que nuevamente juega un importante papel la regla `multiset-diff-meta`).

Para probar que la función `mc-it` es igual a la función `mc`, probamos previamente el invariante que se cumple durante la computación que realiza `mc-it-aux`. En cada iteración `(mc-aux n z)`=`(f91 (f91 .`^{n}`.(f91 z)))`. Para ello definimos la función `iter-f91` que aplica iterativamente la función `f91`:

```
(defun iter-f91 (n x)
  (if (zp n) x (iter-f91 (- n 1) (f91 x))))
```

La propiedad invariante en la computación de `mc-it-aux` queda formalizada por el teorema `iter-f91-mc-aux` siguiente:

```
(defthm iter-f91-mc-aux
  (equal (mc-aux n z) (iter-f91 n z))
  :hints (("Goal" :induct (mc-aux n z))))
```

Como se observa en el consejo de inducción, dicha propiedad se prueba siguiendo el esquema de inducción que describe `mc-aux`. Además, siguiendo la misma técnica que en la admisión de `mc-aux`, hemos necesitado también una regla de reescritura que sirva para expandir adecuadamente la definición de `iter-f91`:

```
(defthm iter-f91-expand
  (implies (and (not (zp n)) (integerp z))
           (equal (iter-f91 n z)
                  (iter-f91 (- n 1)
                            (if (> z 100) (- z 10) 91)))))
```

Una vez probado `iter-f91-mc-aux`, el resultado `equal-mc-it-and-f91` se tiene directamente como caso particular (para `n=1` y `z=x`):

```
(defthm equal-mc-it-and-f91
  (equal (mc-it x) (f91 x)))
```

C.4 Reducciones abstractas

C.4.1 Reducciones Church-Rosser y normalizadoras

Comentamos aquí los aspectos más relevantes de la automatización de la demostración de los teoremas `r-equiv-complete` y `r-equiv-sound` (sección 6.3). Una descripción más detallada se puede encontrar en los comentarios que aparecen en los libros `confluence-.lisp` y `abstract-proofs.lisp`

El esquema de inducción generado por `equiv-p`

Al tener que definir la función `equiv-p` dentro del ámbito de un encapsulado, su esquema recursivo queda marcado como "subversivo" (véase `subversive-recursions`) por el demostrador[5]. Es posible arreglar este problema definiendo una *regla de inducción* adecuada y asignándosela a la función `equiv-p`:

```
(local
 (defun induct-equiv-p (x p)
   (if (endp p)
       t
       (induct-equiv-p (elt2 (car p)) (cdr p)))))

(local
 (defthm equiv-p-induct t
   :rule-classes
   ((:induction :pattern (equiv-p x y p)
                :condition t
                :scheme (induct-equiv-p x p)))))
```

De esta manera, hacemos que la función `equiv-p` sugiera el esquema de inducción correspondiente a su esquema recursivo (lo cual haría el demostrador de manera automática si la definición de `equiv-p` no se encontrara dentro del ámbito de un `encapsulate`). En la prueba de una conjetura `(:P P X Y)`, este esquema de inducción sugiere automáticamente el siguiente intento de prueba:

```
(AND (IMPLIES (AND (NOT (ENDP P))
                   (:P (CDR P) (ELT2 (CAR P)) Y))
              (:P P X Y))
     (IMPLIES (ENDP P) (:P P X Y))).
```

Este esquema es especialmente útil en la prueba de teoremas en los que aparece la relación $\overset{*}{\leftrightarrow}$ y viene a coincidir con la idea intuitiva de "prueba por inducción en el número de pasos de la $\leftrightarrow$-derivación". Nótese, sin embargo, lo que ocurriría si erróneamente el sistema intentara una prueba con el esquema de inducción sugerido, por ejemplo, por `(len p)`:

```
;(AND (IMPLIES (AND (NOT (ENDP P))
;                   (:P (CDR P) X Y))
;              (:P P X Y))
;     (IMPLIES (ENDP P) (:P P X Y))).
```

La hipótesis de inducción no se podría usar, ya que cuando `p` es no vacía, `(cdr p)` *no* es una prueba de la equivalencia de `x` e `y`, sino que es una prueba de la equivalencia entre `(elt2 (car p))` e `y`, idea que acertadamente recoge el esquema de inducción anterior.

[5] Aunque en realidad no lo es, pero el demostrador no detecta tal circunstancia.

Expresión de la irreducibilidad

Para probar la corrección y completitud del algoritmo `r-equiv`, el lema principal que necesitamos es uno análogo al enunciado por 6.9. Este lema afirma que si tenemos dos elementos en forma normal que son equivalentes, éstos han de ser iguales. Téngase en cuenta que con los medios que disponemos en nuestro lenguaje, la única manera que tenemos de afirmar que un elemento X está en forma normal es mediante la fórmula `(not (legal` X `op))`, donde se supone que `op` está universalmente cuantificado. Así, esta fórmula no sirve para expresar la irreducibilidad de un objeto si ha de ser usada en las hipótesis de una implicación. Por ejemplo, el lema que estamos discutiendo quedaría expresado de forma errónea por la siguiente fórmula (ya que la variable `op` quedaría, de manera implícita, cuantificada existencialmente):

```
;(implies (and (equiv-p x y p)
;               (not (legal x op))
;               (not (legal y op)))
;         (equal x y))
```

Es posible resolver este problema de dos maneras distintas. La solución adoptada en el libro `confluence.lisp` es debilitar el resultado, expresando que `x` e `y` no se reducen respecto de dos operadores en concreto, precisamente aquellos que se necesitan en la prueba del resultado. De esta manera el lema 6.9 queda:

```
(defthm if-CR--two-ireducible-connected-are-equal
  (implies
    (and (equiv-p x y p)
         (not (legal x (operator (first (transform-to-valley p)))))
         (not (legal y (operator (last-elt (transform-to-valley p))))))
     (equal x y)))
```

Así, el teorema se podrá usar para concluir la igualdad de cualesquiera X e Y que sean equivalentes e irreducibles. Éste es el caso, por ejemplo, de `(normal-form x)` y `(normal--form y)`, ya que el lema `irreducible-normal-form` afirma que no existen operadores aplicables a tales objetos, en particular los operadores que aparecen en el enunciado de este teorema.

Existe una manera alternativa de resolver el problema: aumentar la expresividad del lenguaje. Esto quiere decir que se puede suponer la existencia de una función `reducible`, asumiendo que dicha función, cuando recibe un objeto `x`, devuelve un operador legal en el caso de que `x` sea reducible o `nil` en caso contrario. De esta manera el lema se podría haber expresado de la siguiente manera:

```
;(implies (and (equiv-p x y p)
;               (not (reducible x))
;               (not (reducible y)))
;         (equal x y))
```

De hecho, en una versión preliminar se usó esta aproximación. Un análisis detallado de las pruebas generadas reveló que se podía formalizar el teorema de decidibilidad sin

hacer mención explícita de una función como `reducible`, tal y como finalmente hicimos. El lector puede consultar el libro `confluence-v0.lisp`, donde podrá ver una formalización de estos resultados asumiendo la existencia de una función como `reducible`. Aún a costa de cierta perdida de claridad en la exposición, adoptamos finalmente la formalización sin dicha función, ya que el resultado obtenido es más fuerte desde el punto de vista teórico: podemos demostrar la decidibilidad de $\overset{*}{\leftrightarrow}$ sin necesidad de disponer de un algoritmo para comprobar la reducibilidad de un objeto. Aunque debemos decir que en la práctica la mejora no es tal, ya que estamos asumiendo la existencia de una función como `proof-irreducible`: se hace dificil pensar en un caso en el que se disponga de dicha función y no se disponga de un test de reducibilidad. Por ejemplo, como se ve en la sección 6.5, si la reducción es noetheriana sí que necesitaremos asumir la existencia de una función como `reducible` para calcular formas normales.

Los teoremas de corrección y completitud

Para probar la completitud del algoritmo `r-equiv`, basta con comprobar que la formas normales respectivas de dos elementos `x` e `y` son iguales, si `(equiv-p x y p)`. Para ello usamos el teorema `if-CR--two-ireducible-connected-are-equal`, sustituyendo `x` e `y` por `(normal-form x)` y `(normal-form y)`, tal y como se acaba de explicar. Restará entonces encontrar una prueba que justifique la equivalencia de ambas formas normales. Esta prueba viene dada por la siguiente función, junto con su propiedad fundamental:

```
(defun make-proof-between-normal-forms (x y p)
  (append (inverse-proof (proof-irreducible x))
          p
          (proof-irreducible y)))

(defthm make-proof-between-normal-forms-indeed
  (implies (equiv-p x y p)
           (equiv-p (normal-form x)
                    (normal-form y)
                    (make-proof-between-normal-forms x y p)))))
```

Si deshabilitamos previamente la definición de la función `make-proof-between-normal-forms`, el resultado de completitud se tiene directamente con la instanciación adecuada del teorema `if-CR--two-ireducible-connected-are-equal`:

```
(defthm r-equiv-complete
  (implies (equiv-p x y p)
           (r-equiv x y))
  :hints (("Goal" :use ((:instance
                         if-CR--two-ireducible-connected-are-equal
                         (x (normal-form x))
                         (y (normal-form y))
                         (p (make-proof-between-normal-forms x y p)))))))
```

Finalmente, el teorema de corrección del algoritmo `r-equiv` se obtiene directamente como instancia de la propiedad de simetría probada previamente para la relación `equiv-p`:

```
(defthm r-equiv-sound
  (implies (and (q x) (q y) (r-equiv x y))
           (equiv-p x y (make-proof-common-n-f x y)))
  :hints (("Subgoal 3"
           :use ((:instance equiv-p-symmetric
                            (x y)
                            (y (normal-form y))
                            (p (proof-irreducible y)))
                 (:instance equivalent-normal-form (x y))))))
```

C.4.2 El lema de Newman

Mostramos aquí algunos aspectos de la demostración automática del lema de Newman. Para más detalles, el lector puede consultar el libro `newman.lisp`.

El lema `transform-to-valley-admission`

La parte más laboriosa en la prueba del lema de Newman es la prueba del lema `transform--to-valley-admission`, que va a permitir probar la terminación de la función `transform--to-valley` y por tanto la admisión de su definición. Recordemos que el lema afirma el decrecimiento de la medida de una prueba que tenga un pico local, cuando ese pico local se reemplaza por una prueba valle equivalente:

```
(defthm transform-to-valley-admission
  (implies (exists-local-peak p)
           (mul-rel (proof-measure (replace-local-peak p))
                    (proof-measure p))))
```

En un primer intento de la prueba de esta conjetura, se generan dos subobjetivos, acorde con la definición de `mul-rel`:

```
Subgoal 2
(IMPLIES (EXISTS-LOCAL-PEAK P)
         (CONSP (MULTISET-DIFF (PROOF-MEASURE P)
                               (PROOF-MEASURE (REPLACE-LOCAL-PEAK P))))).
Subgoal 1
(IMPLIES (EXISTS-LOCAL-PEAK P)
         (FORALL-EXISTS-REL-BIGGER
              (MULTISET-DIFF (PROOF-MEASURE (REPLACE-LOCAL-PEAK P))
                             (PROOF-MEASURE P))
              (MULTISET-DIFF (PROOF-MEASURE P)
                             (PROOF-MEASURE (REPLACE-LOCAL-PEAK P)))))
```

La prueba automática de tales subobjetivos se consigue proporcionando una serie de resultados en forma de reglas que permitan simplificar ambos a `t`. En lo que sigue, describiremos estas reglas y cómo estos dos subobjetivos van simplificándose paulatinamente hasta llegar a probarse. Nuestra descripción, por tanto, girará argumentalmente alrededor de las progresivas simplificaciones que van sufriendo los dos subobjetivos.

1. En primer lugar, obsérvese que una prueba `p` que tenga un pico local puede dividirse en tres partes: la parte de la prueba anterior al pico local, el pico local y la parte posterior al pico local:

```
(defun proof-before-peak (p)
  (cond ((or (atom p) (atom (cdr p))) p)
        ((and (not (direct (car p))) (direct (cadr p))) nil)
        (t (cons (car p) (proof-before-peak (cdr p))))))

(defun proof-after-peak (p)
  (cond ((atom p) p)
        ((atom (cdr p)) (cdr p))
        ((and (not (direct (car p))) (direct (cadr p)))
         (cddr p))
        (t (proof-after-peak (cdr p)))))

(defun local-peak (p)
  (cond ((atom p) p)
        ((atom (cdr p)) (cdr p))
        ((and (not (direct (car p))) (direct (cadr p)))
         (list (car p) (cadr p)))
        (t (local-peak (cdr p)))))

(defthm proof-peak-append
  (implies (exists-local-peak p)
    (equal (append (proof-before-peak p)
                   (append (local-peak p)
                           (proof-after-peak p)))
           p))
   :rule-classes (:elim :rewrite))
```

El hecho de que este último resultado esté definido como *regla de eliminación* hace que cuando `(exists-local-peak p)` esté entre las hipótesis, la prueba `p` se reescribe a una expresión en la que aparecen explícitos estas tres partes en las que se divide.

De igual manera podemos dividir `(replace-local-peak p)` en tres partes, quedando explícito el papel de `transform-local-peak`

```
(defthm replace-local-peak-another-definition
  (implies (exists-local-peak p)
           (equal (replace-local-peak p)
                  (append (proof-before-peak p)
                          (append (transform-local-peak
                                    (local-peak p))
                                  (proof-after-peak p)))))))
```

Esta división de las pruebas en tres partes se corresponde también con una división análoga del multiconjunto correspondiente a su medida, sin más que usar el siguiente lema:

```
(defthm proof-measure-append
  (equal (proof-measure (append p1 p2))
         (append (proof-measure p1)
                 (proof-measure p2))))
```

Las reglas anteriores hacen que los multiconjuntos `(proof-measure p)` y `(proof--measure (replace-local-peak p))` aparezcan explícitamente como multiconjuntos con la misma parte inicial y la misma parte final. Las reglas `multiset-diff--append-1` y `multiset-diff-append-2`, vistas en la sección 5.2.6, junto con las congruencias generadas por la llamada a `defmul` que definió a `mul-rel`, hacen que estas partes comunes se simplifiquen, quedando los subobjetivos de manera que podemos centrar nuestra atención en el pico local y en su transformado:

```
Subgoal 2'
(IMPLIES
 (EXISTS-LOCAL-PEAK P)
 (CONSP
     (MULTISET-DIFF (PROOF-MEASURE (LOCAL-PEAK P))
                    (PROOF-MEASURE (TRANSFORM-LOCAL-PEAK (LOCAL-PEAK P))))))

Subgoal 1'
(IMPLIES
 (EXISTS-LOCAL-PEAK P)
 (FORALL-EXISTS-REL-BIGGER
     (MULTISET-DIFF (PROOF-MEASURE (TRANSFORM-LOCAL-PEAK (LOCAL-PEAK P)))
                    (PROOF-MEASURE (LOCAL-PEAK P)))
     (MULTISET-DIFF (PROOF-MEASURE (LOCAL-PEAK P))
                    (PROOF-MEASURE (TRANSFORM-LOCAL-PEAK (LOCAL-PEAK P))))))
```

2. El siguiente paso consiste en simplificar aún mas las diferencias entre los dos multiconjuntos que aparecen en los dos subobjetivos anteriores, ya que podemos probar que el primer elemento de tales multiconjuntos es común.

   En primer lugar, observemos que si dos pruebas son equivalentes, los respectivos multiconjuntos correspondientes a la medida de tales pruebas comparten el primer elemento y al hacer la diferencia entre ambos multiconjuntos podemos simplificar ese primer elemento común. Sin embargo debemos tener en cuenta un detalle: alguna de las pruebas puede ser vacía. La siguiente regla expresa este resultado, distinguiendo convenientemente el caso de que alguna de las pruebas sea vacía:

```
(defthm multiset-diff-proof-measure
   (implies (and (equiv-p x y p1) (equiv-p x z p2))
            (equal (multiset-diff (proof-measure p1)
                                  (proof-measure p2))
                   (if (consp p1)
                       (if (consp p2)
                           (multiset-diff (cdr (proof-measure p1))
                                          (cdr (proof-measure p2)))
                         (proof-measure p1))
                     nil))))
```

Los dos teoremas siguientes expresan que `(local-peak p)` y `(transform-local--peak (local-peak p))` son efectivamente pruebas equivalentes:

```
(defthm local-peak-equiv-p
   (implies (exists-local-peak p)
            (equiv-p (elt1 (car (local-peak p)))
                     (elt2 (cadr (local-peak p)))
                     (local-peak p))))

(defthm transform-local-peak-equiv-p
   (implies (exists-local-peak p)
            (equiv-p (elt1 (car (local-peak p)))
                     (elt2 (cadr (local-peak p)))
                     (transform-local-peak (local-peak p)))))
```

De esta manera, los subobjetivos quedan ahora simplificados como se indica a continuación, habiéndose eliminado el primer elemento de los multiconjuntos que se comparan. Además, ya sólo tenemos que centrar nuestra atención en el caso en el que `(transform-local-peak (local-peak p))` no sea una prueba vacía:

```
Subgoal 2.2
(IMPLIES
  (AND (EXISTS-LOCAL-PEAK P)
       (CONSP (TRANSFORM-LOCAL-PEAK (LOCAL-PEAK P))))
  (CONSP (MULTISET-DIFF
           (CDR (PROOF-MEASURE (LOCAL-PEAK P)))
           (CDR (PROOF-MEASURE (TRANSFORM-LOCAL-PEAK (LOCAL-PEAK P)))))))

Subgoal 1.2
(IMPLIES
 (AND (EXISTS-LOCAL-PEAK P)
      (CONSP (TRANSFORM-LOCAL-PEAK (LOCAL-PEAK P))))
 (FORALL-EXISTS-REL-BIGGER
   (MULTISET-DIFF
        (CDR (PROOF-MEASURE (TRANSFORM-LOCAL-PEAK (LOCAL-PEAK P))))
        (CDR (PROOF-MEASURE (LOCAL-PEAK P))))
   (MULTISET-DIFF
        (CDR (PROOF-MEASURE (LOCAL-PEAK P)))
        (CDR (PROOF-MEASURE (TRANSFORM-LOCAL-PEAK (LOCAL-PEAK P)))))))
```

3. El siguiente paso de simplificación intenta que aparezca en los subobjetivos una referencia explícita al elemento mayor de la prueba `(local-peak p)`, es decir el elemento que se reduce en el pico local. Este elemento viene definido por la función `local-peak` siguiente:

```
(defun peak-element (p) (elt1 (cadr (local-peak p))))
```

La siguiente propiedad de `peak-element` permite obtener explícitamente `(peak--element p)` en los subobjetivos:

```
(defthm cdr-proof-measure-local-peak
  (implies (exists-local-peak p)
           (equal (cdr (proof-measure (local-peak p)))
                  (list (peak-element p)))))
```

De esta manera, los subobjetivos anteriores quedan ahora simplificados de la siguiente manera:

```
Subgoal 2.2
(IMPLIES
  (AND (EXISTS-LOCAL-PEAK P)
       (CONSP (TRANSFORM-LOCAL-PEAK (LOCAL-PEAK P))))
  (CONSP (MULTISET-DIFF
              (LIST (PEAK-ELEMENT P))
              (CDR (PROOF-MEASURE
                      (TRANSFORM-LOCAL-PEAK (LOCAL-PEAK P)))))).

Subgoal 1.2
(IMPLIES
 (AND (EXISTS-LOCAL-PEAK P)
      (CONSP (TRANSFORM-LOCAL-PEAK (LOCAL-PEAK P))))
 (FORALL-EXISTS-REL-BIGGER
     (ACL2::REMOVE-ONE
           (PEAK-ELEMENT P)
           (CDR (PROOF-MEASURE (TRANSFORM-LOCAL-PEAK (LOCAL-PEAK P)))))
     (MULTISET-DIFF
          (LIST (PEAK-ELEMENT P))
          (CDR (PROOF-MEASURE (TRANSFORM-LOCAL-PEAK (LOCAL-PEAK P)))))))
```

4. Antes de seguir con la simplificación de los subobjetivos, veamos un resultado que nos va a ser de utilidad en esa tarea. En primer lugar definimos una función que formaliza el concepto de que un elemento sea mayor, respecto de `rel`, que todos los elementos de una lista:

```
(defun rel-bigger-than-list (x l)
   (if (atom l)
       t
     (and (rel (car l) x) (rel-bigger-than-list x (cdr l)))))
```

Nótese que `(peak-element p)` es un elemento mayor (respecto a `rel`) que cualquiera de los elementos de la prueba valle `(transform-local-peak (local-peak p))`. Esto queda expresado mediante el siguiente lema:

```
(defthm valley-rel-bigger-peak-lemma
   (implies (exists-local-peak p)
            (rel-bigger-than-list
             (peak-element p)
             (proof-measure (transform-local-peak (local-peak p))))))
```

No detallaremos aquí la prueba de este resultado, que es bastante laboriosa. Simplemente diremos que es en la prueba de este resultado donde se necesita la transitividad de `rel`, tal y como comentamos en la subsección 6.4.1. El lector interesado puede consultar los comentarios que aparecen en la subsección `3.2.4` del libro `newman.lisp`.

5. Existen dos razones por las cuales el lema `valley-rel-bigger-peak-lemma` va a servir para simplificar los subobjetivos que aún quedan por resolver.

   La primera de ellas es que tal lema permite deducir que `(peak-element p)` no es ninguno de los elementos de la prueba valle, ya que:

```
(defthm rel-bigger-than-list-not-member
     (implies (rel-bigger-than-list x l)
              (not (member x l)))))
```

   Esto nos permite en primer lugar resolver el primero de los dos subobjetivos anteriores. Además, en el otro subobjetivo, permite eliminar las referencias a `multi-set-diff`, quedado de esta manera como único subobjetivo pendiente de resolver el siguiente:

```
Subgoal 1.2
(IMPLIES (AND (EXISTS-LOCAL-PEAK P)
              (CONSP (TRANSFORM-LOCAL-PEAK (LOCAL-PEAK P))))
         (FORALL-EXISTS-REL-BIGGER
              (CDR (PROOF-MEASURE (TRANSFORM-LOCAL-PEAK (LOCAL-PEAK P))))
              (LIST (PEAK-ELEMENT P)))).
```

6. El otro motivo por el cual nos va ser útil el resultado `valley-rel-bigger-peak--lemma` es que mediante el siguiente lema podemos simplificar la llamada a `forall--exists-rel-bigger` a una llamada a la función `rel-bigger-than-list`:

```
(defthm rel-bigger-than-list-forall-exists-rel-bigger
   (equal (forall-exists-rel-bigger l (list x))
          (rel-bigger-than-list x l))))
```

   Este lema, junto con `valley-rel-bigger-peak-lemma` hacen que se simplifique a `t` el único subobjetivo que quedaba pendiente, con lo cual se concluye con la prueba del teorema `transform-to-valley-admission`.

Las propiedades de `transform-to-valley`

Una vez definida la función `transform-to-valley`, la prueba de sus dos propiedades fundamentales, `equiv-p-x-y-transform-to-valley` y `valley-transform-to-valley` no es difícil.

Lo más destacable aquí es que ambos teoremas se demuestran usando el esquema de inducción sugerido por la función `transform-to-valley`. En la prueba de una conjetura `(P: P X Y)`, este esquema de inducción sugiere el siguiente intento de prueba:

```
(AND (IMPLIES (NOT (AND (STEPS-Q P) (EXISTS-LOCAL-PEAK P)))
              (:P P X Y))
     (IMPLIES (AND (AND (STEPS-Q P) (EXISTS-LOCAL-PEAK P))
                   (:P (REPLACE-LOCAL-PEAK P) X Y))
              (:P P X Y))).
```

Este esquema de inducción está justificado por la buena fundamentación de `mul-rel`: se trata de una demostración por inducción en la medida de la prueba. Esto contrasta con la demostración estándar, por inducción noetheriana respecto de la reducción $\rightarrow$, que se realiza en 6.24. La demostración que se genera con el esquema de inducción anterior plantea el teorema como una propiedad *de las pruebas abstractas* más que de los objetos individuales. La hipótesis de inducción se asume para una prueba más pequeña (respecto de `mul-rel`), en lugar de tener una hipótesis de inducción asumida para un objeto más pequeño respecto de $\rightarrow$. Nuevamente hacemos énfasis en la importancia de considerar las pruebas abstractas como objetos con entidad propia.

El primero de los teoremas que debemos probar es `equiv-p-x-y-transform-to--valley`, que asegura que la prueba que obtiene `transform-to-valley` es una prueba equivalente. Para ello nótese que en cada una de las iteraciones que efectúa `transform--to-valley`, la prueba que se obtiene es equivalente.

```
(defthm equiv-p-x-y-replace-local-peak
   (implies (and (equiv-p x y p) (exists-local-peak p))
            (equiv-p x y (replace-local-peak p)))
```

Usando el esquema de inducción anterior y este lema, el demostrador prueba fácilmente que `(transform-to-valley p)` es una prueba equivalente a `p`, tal y como establece `equiv-p-x-y-transform-to-valley`.

Para probar la propiedad `valley-transform-to-valley`, igualmente nos hará falta un lema previo. En este caso, debemos probar que la condición de parada del algoritmo que describe `transform-to-valley` implica que la prueba obtenida es una prueba valle. Es decir, si una prueba no tiene picos locales, es una prueba valle:

```
(defthm steps-valley-not-exists-local-peak
   (implies (equiv-p x y p)
            (equal  (steps-valley p) (not (exists-local-peak p))))))
```

Nuevamente, usando el esquema de inducción anterior, tendremos que la prueba que finalmente obtiene `transform-to-valley` es una prueba valle, tal y como establece `valley--transform-to-valley`.

C.4.3 Reducciones convergentes: decidibilidad

Veamos cómo la regla derivada de instanciación funcional permite concluir la decidibilidad de la relación de equivalencia descrita por una reducción convergente. Aunque las funciones que describen la reducción y sus propiedades en los resultados descritos en las secciones 6.3, 6.4 y 6.5 tienen el mismo nombre de símbolo, difieren en el paquete en el cual están definidas. Por esta razón, las instancias funcionales que necesitamos para exportar estos resultados son siempre del mismo tipo: a cada función o variable se le hace corresponder otra función o variable con el mismo nombre de símbolo pero perteneciente a otro paquete.

Para hacer más cómoda la instanciación funcional usamos una función que calcula un consejo adecuado (véase `computed-hints`). Esta función la hemos llamado `pkg-functional-instance` y se usa de la siguiente manera:

`(pkg-functional-instance acl2::id` *`nombre-lema variables funciones`*`)`

donde *nombre-lema* es el nombre del teorema que se quiere instanciar (incluyendo el nombre del paquete), *variables* es una lista con los símbolos de variables que se instancian y *funciones* es una lista con los símbolos de función que se instancian. El consejo que construye esta llamada a `pkg-functional-instance` es la instanciación del teorema *nombre-lema* cuya sustitución funcional hace corresponder cada símbolo de la listas *variables* y *funciones* del paquete en el que está definido *nombre-lemma*, con los símbolos análogos del paquete actual.

Como ejemplo, obsérvese cómo se demuestra el teorema `r-equivalent-sound` mediante instanciación funcional del teorema `r-equiv-sound` presentado en la subsección C.4.1 (recuérdese que tal teorema se demostró dentro del paquete `CNF`):

```
(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)))

  :hints ((pkg-functional-instance
           acl2::id
           'cnf::r-equiv-sound
           '(x y)
           '(q legal make-proof-common-n-f proof-step-p
                  r-equiv  equiv-p
                  reduce-one-step proof-irreducible transform-to-valley
                  normal-form))))
```

Este tipo de instanciación funcional, construida mediante un consejo calculado con `pkg-functional-instance`, se usa varias veces en el libro `convergent.lisp` para exportar de manera fácil resultados de los libros `confluence.lisp` y `newman.lisp`. Instamos al lector interesado en más detalles sobre la demostración automática, a consultar los eventos y comentarios en el libro `convergent.lisp`.

C.5 Teorías ecuacionales y sistemas de reescritura

C.5.1 Teorías ecuacionales

La demostración automática de las propiedades del predicado `eq-equiv-s-p` (subsección 7.1.2) usando ACL2, resulta sencilla y son pocos los lemas previos que se necesitan. En la mayoría de los casos, la prueba se realiza mediante el esquema de inducción sugerido por dicho predicado. Por ejemplo, en el caso de la prueba de la estabilidad, se genera automáticamente el siguiente esquema de inducción:

```
; (AND (IMPLIES (AND (NOT (ENDP P))
;                    (:P E (CDR P) SIGMA (ELT2 (CAR P)) T2))
```

```
;               (:P E P SIGMA T1 T2))
;      (IMPLIES (ENDP P)
;               (:P E P SIGMA T1 T2)))
```

Este esquema de inducción es similar al esquema de inducción sugerido por `equiv-p`. Ya comentamos en la subsección C.4.1 la conveniencia de este esquema para el éxito de la prueba, frente al sugerido, por ejemplo por la longitud de la prueba.

Destacar también que para demostrar las propiedades de estabilidad y compatibilidad, se necesitan las propiedades relativas a la estructura de árbol de los términos de primer orden, establecidas en la sección 3.2.

C.5.2 Reducibilidad

En cuanto a la verificación de las propiedades de la función `eq-reducible` (subsección 7.2.1), simplemente remarcar la manera en que usamos en el demostrador las funciones `position-p-rec`, `occurrence-rec` y `replace-term-rec` en lugar de `position-p`, `occurrence` y `replace-term`. Hemos definido la siguiente constante, conteniendo los nombres de las reglas que reescriben éstas a aquéllas:

```
(defconst *position-rec-versions*
  '(equal-position-p-position-p-rec
    equal-occurrence-occurrence-rec
    equal-replace-term-replace-term-rec))
```

En principio estas reglas están deshabilitadas:

```
(in-theory
 (set-difference-theories
      (current-theory :here) *position-rec-versions*))
```

Si hemos de demostrar un teorema en el que intervienen conceptos relativos a la estructura de árbol de los términos y ese teorema involucra funciones que están definidas por recursión estructural, formulamos una versión general de dicho teorema, expresándolo tanto para términos como para listas de términos, usando por tanto las funciones `position-p-rec`, `occurrence-rec` y `replace-term-rec`. Por ejemplo, como hemos visto, en lugar de intentar demostrar directamente los teoremas sobre `eq-reducible`, formulamos versiones más generales sobre `eq-reducible-aux`. Estas versiones generales resultan usualmente más fáciles de probar, ya que se intentan por recursión en la estructura de los términos, como se ha explicado en la subsección 3.1.3.

Finalmente, se obtiene como consecuencia el teorema que originalmente se quería demostrar, formulado para términos y usando ahora `position-p`, `occurrence` y `replace-term`, sin más que habilitar los teoremas de equivalencia entre ambas versiones:

```
(in-theory
 (union-theories (current-theory :here) *position-rec-versions*))
```

Esta técnica se usa con frecuencia en la demostración automática de la teoría presentada en esta memoria.

C.5.3 Confluencia: el teorema de pares críticos de Knuth y Bendix

La demostración automática del teorema de pares críticos de Knuth y Bendix es el resultado de una interacción típica con el demostrador. La prueba a mano nos marca la estrategia a seguir y los casos a considerar. Para cada una de las situaciones descritas en la subsección 7.4.2, se proporcionan los lemas necesarios que terminan por demostrar los teoremas deseados. Consúltese el libro `critical-pairs.lisp` para seguir con detalle la secuencia completa de eventos (definiciones, teoremas, habilitaciones, deshabilitaciones,...) que finalmente llevan a la demostración del teorema. En cualquier caso, comentamos en lo que sigue algunos detalles que creemos merece la pena destacar.

El uso de argumentos irrelevantes

Muchas de las funciones definidas para la demostración, que construyen pruebas ecuacionales, tienen entre sus argumentos algunos irrelevantes. Por ejemplo, la función `transform--eq-top-variable-overlap-second-piece-2`, viene definida con 14 argumentos, 6 de los cuales son irrelevantes:

```
;(defun transform-eq-top-variable-overlap-second-piece-2
;   (peak u1 u2 q1 q2 sigma1 sigma2 l1 r1 l2 r2 x val valr)
;   (declare (ignore peak sigma1 r1 valr u1  val))
;   ....)
```

En general, hemos introducido estos argumentos irrelevantes para que la lista de argumentos de la función que obtiene la prueba ecuacional coincida con la lista de argumentos del predicado que describe la situación que resuelve. Por ejemplo, la función anterior va a servir para resolver situaciones de superposición variable. El predicado que define las superposiciones variables, `eq-top-variable-overlap-p`, tiene los mismos argumentos:

```
; (defun eq-top-variable-overlap-p
;    (peak u1 u2 q1 q2 sigma1 sigma2 l1 r1 l2 r2 x val valr E)
;    ...)
```

Dos razones motivan esta introducción de argumentos extra en las funciones de construcción de las pruebas. En primer lugar, creemos que se obtiene una mayor claridad en el desarrollo de la demostración, ya que no es necesario recordar los argumentos concretos de cada función, sino sólo los de la situación que resuelve. Por ejemplo, la superposiciones variables se resuelven mediante tres trozos de prueba. Cada uno de ellos usa realmente una lista de argumentos distinta. Sin embargo, las funciones que calculan cada uno de los tres trozos tienen los mismos argumentos.

Por otro lado, así se evitan la aparición de numerosas variables libres, que entorpecerían el mecanismo de reescritura del demostrador. Por ejemplo si hubiéramos definido la función `transform-eq-top-variable-overlap-second-piece-2` sin argumentos irrelevantes, el teorema principal sobre tal función quedaría:

```
; (defthm transform-eq-top-variable-overlap-second-piece-2-is-a-proof
;   (implies
;    (eq-top-variable-overlap-p
;     peak u1 u2 q1 q2 sigma1 sigma2 l1 r1 l2 r2 x val valr E)
```

```
;    (eq-equiv-s-p
;     (instance l1 (cons (cons x valr) sigma1))
;     u2
;     (transform-eq-top-variable-overlap-second-piece-2
;      u2 q1 q2 sigma2 l1 l2 r2 x)
;     E)))
```

La regla de reescritura que se generaría con este teorema tendría a `u1` y `val` como variables libres. Para aplicarla el demostrador debería "adivinar" la asignación a las variables libres de entre las hipótesis en ese momento presentes. En muchas ocasiones, una mala elección hace que la regla no se pueda aplicar. Compárese con la forma de este teorema que aparece en la subsección 7.4.4, que da lugar a una regla de reescritura sin variables libres.

Otras cuestiones reseñables

Comentamos a continuación otras cuestiones sobre la demostración automática del teorema, que a nuestro juicio merece la pena destacar. En primer lugar, nótese el uso de la siguiente regla de eliminación para tratar las posiciones prefijas y la diferencia entre posiciones:

```
(defthm prefix-elim
  (implies (prefix p1 p2)
           (equal (append p1 (difference-pos p1 p2)) p2))
   :rule-classes :elim)
```

Esta regla de eliminación permite que al asumir que `(prefix p1 p2)`, el proceso de eliminación de destructores escriba la posición `p2` como `(append p1 q)`, donde `q` es `(difference-pos p1 p2)`. Esto se corresponde con la frase siguiente de la prueba a mano: *sea q tal que $p_2 = p_1 \cdot q$.*

Por otro lado, nótese el uso de congruencias para el tratamiento de las ecuaciones renombradas. Recuérdese (sección 4.1) que la relación `renamed` se había probado de equivalencia. La siguiente regla relaciona las posiciones de un termino con las posiciones de cualquiera de sus renombrados:

```
(defcong renamed iff (position-p pos term) 2)
```

Esta congruencia permite, por ejemplo, razonar cómodamente sobre las posiciones del lado izquierdo de una ecuación renombrada.

Otra cuestión a destacar nuevamente es el uso de las versiones recursivas de los conceptos de posición, ocurrencia y reemplazamiento (subsección 7.2.2). Por ejemplo, el teorema `positions-variable-x-main-property` que establece la propiedad principal de la función `positions-variable-x` (necesaria para tratar las superposiciones variables) se demuestra más fácilmente probando previamente esta versión más general:

```
(defthm positions-variable-x-main-property-aux
  (iff (member pos (positions-variable-x flg term x))
       (and (position-p-rec flg pos term)
            (variable-p (occurrence-rec flg term pos))
            (equal (occurrence-rec flg term pos) x))))))
```

Por último, comentar la necesidad del uso de `disable` para integrar cada uno de los casos en los que se divide la prueba. Nótese que las funciones definidas para resolver cada caso son funciones no recursivas. Por tanto es fundamental deshabilitar sus definiciones una vez se han probado sus propiedades fundamentales. Esto permite cierta estructura modular de la prueba completa.

C.5.4 Decidibilidad de teorías ecuacionales

Lo más reseñable en la automatización de la demostración del teorema de decidibilidad de una teoría ecuacional descrita por un SRT completo (subsección 7.6.2) es el uso de instanciación funcional, con dos objetivos:

- Aplicar el teorema de pares críticos al sistema `(RKB)`.
- Deducir la decidibilidad de la teoría ecuacional de `(RKB)`, trasladando el teorema probado en la sección 7.2 de reducciones abstractas a la reducción asociada a `(RKB)`.

En el primer caso, nos permite demostrar la confluencia local de la reducción ecuacional (teoremas `RKB-transform-eq-local-peak-is-a-proof` y `RKB-transform-eq-local-peak-is-a-valley`). Por ejemplo, el consejo de instanciación para el primero de los teoremas es el siguiente:

```
;   :hints (("Goal"
;             :use (:functional-instance
;                   transform-eq-local-peak-is-a-proof
;                   (EKB RKB)
;                   (transform-eq-local-peak RKB-transform-eq-local-peak)
;                   (transform-eq-local-peak-aux
;                                          RKB-transform-eq-local-peak-aux)
;                   (transform-eq-prefix-peak RKB-transform-eq-prefix-peak)
;                   (transform-eq-prefix-peak-symmetric-case
;                    RKB-transform-eq-prefix-peak-symmetric-case)
;                   (transform-eq-top-local-peak
;                    RKB-transform-eq-top-local-peak)
;                   (transform-eq-top-critical-overlap
;                    RKB-transform-eq-top-critical-overlap)
;                   (transform-critical-pair
;                    RKB-transform-critical-pair))))))
```

Como se observa, se trata de sustituir las funciones específicas sobre `(EKB)` por las análogas definidas para `(RKB)`.

El segundo de los usos de la instanciación funcional nos permite demostrar la corrección y completitud del algoritmo `RKB-equivalent` (teoremas `RKB-equivalent-complete` y `RKB-equivalent-sound`). Por ejemplo, para el primero de estos teoremas, el consejo de instanciación usado es:

```
;  :hints (("Goal"
;            :use (:functional-instance
;                  (:instance CNV::r-equivalent-complete
```

```
;                          (CNV::x t1) (CNV::y t2) (CNV::p p))
;                 (CNV::q (lambda (x) (term-s-p x)))
;                 (CNV::q-w (lambda () 0))
;                 (CNV::equiv-p (lambda (t1 t2 p)
;                                 (eq-equiv-s-p t1 t2 p (RKB))))
;                 (CNV::proof-step-p (lambda (s)
;                                      (eq-proof-step-p s (RKB))))
;                 (CNV::r-equivalent RKB-equivalent)
;                 (CNV::normal-form RKB-normal-form)
;                 (CNV::reducible (lambda (term)
;                                   (eq-reducible term (RKB))))
;                 (CNV::reduce-one-step eq-reduce-one-step)
;                 (CNV::legal (lambda (term op)
;                               (eq-legal term op (RKB))))
;                 (CNV::rel red<)
;                 (CNV::fn fn-red<)
;                 (CNV::transform-local-peak
;                            RKB-transform-eq-local-peak)))))
```

Esta sustitución funcional sustituye todas las funciones relativas a la reducción abstracta[6], por las funciones correspondientes de la reducción ecuacional asocida a `(RKB)`. Por ejemplo:

- El dominio de definición `q` es ahora el conjunto de términos en una signatura dada, `term-s-p`.
- La función que aplica un paso de reducción, `reduce-one-step`, es sustituida por su contrapartida ecuacional, `eq-reduce-one-step`.
- El test de aplicabilidad, `legal`, es ahora `(lambda (term) (eq-legal term (RKB))`
- El test de reducibilidad, `reducible`, se sustituye por la `lambda` expresión `(lambda (term) (eq-reducible term (RKB)))`. Es decir, reducibilidad ecuacional respecto del sistema `(RKB)`.

[6] Definidas en el paquete `CNV`.

Bibliografía

[1] Franz Baader y Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[2] Leo Bachmair. *Canonical Equational Proofs*. Research Notes in Theoretical Computer Science. Birkhaüser, 1990.

[3] Robert Boyer, David M. Goldschlag, Matt Kaufmann y J Strother Moore. Functional instantiation in first-order logic. En V. Lifschitz, ed., *Artificial Intelligence and Mathematical Theory of Computation. Papers in Honour of John McCarthy*. Academic Press, 1991.

[4] Robert Boyer y J Stroother Moore. A mechanical proof of the unsolvability of the halting problem. *Journal of the ACM*, 31(3):441–458, 1984.

[5] Robert Boyer y J Stroother Moore. Nqthm, the Boyer-Moore theorem prover. URL: `ftp.cs.utexas.edu/pub/boyer/nqthm/index.html`, 1997.

[6] Robert S. Boyer y J Strother Moore. *A Computational Logic*. ACM monograph series. Academic Press, 1979.

[7] Robert S. Boyer y J Strother Moore. *A Computational Logic Handbook*. Academic Press, segunda edición, 1998.

[8] Bishop Brock. `defstructure` for ACL2 version 2.0. Informe técnico, 1997 (véase [39]).

[9] Bishop Brock, Matt Kaufmann y J Stroother Moore. ACL2 theorems about commercial microprocessors. En M. Srivas y A. Camilleri, eds., *Proceedings of Formal Methods in Computer-Aided Design (FMCAD'96)*, volumen 1166 de *LNCS*, páginas 275–293. Springer Verlag, 1998.

[10] Alan Bundy. A survey of automated deduction. Informe técnico, Informatics Series Report 01, University of Edinburgh, 1999.

[11] Alan Bundy, Frank van Harmelen, Christian Horn y Alan Smaill. The Oyster–Clam system. En M. E. Stickel, ed., *Proceedings of the 10th International Conference on Automated Deduction*, volumen 449 de *LNAI*, páginas 647–648. Springer Verlag, 1990.

[12] Jacques Calmet y Karsten Homann. Classification of communication and cooperation mechanisms for logical and symbolic computation systems. En F. Baader y K. U. Schulz, eds., *Frontiers of Combining Systems: Proceedings of the 1st International Workshop*, Applied Logic, páginas 221–234. Kluwer Academic Publishers, 1996.

[13] Edmund Clarke y Xudong Zhao. Analytica — A theorem prover in Mathematica. En D. Kapur, ed., *CADE-11: 11th International Conference on Automated Deduction*, volumen 1138 de *LNAI*, páginas 761–765. Springer-Verlag, 1992.

[14] Robert L. Constable, S. Allen, H. Bromely, W. Clevely, et al. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, Inc., 1986.

[15] John Cowles. Knuth's generalization of McCarthy's 91 function. En M. Kaufmann, P. Manolios y J S. Moore, eds., *Computer-Aided Reasoning: ACL2 Case Studies*, capítulo 17. Kluwer Academic Publishers, 2000.

[16] Nachum Dershowitz y Zohar Manna. Proving termination with multiset orderings. En H.A. Maurer, ed., *Annual International Colloquium on Automata, Languages and Programming*, volumen 71 de *LNCS*, páginas 188–202. Springer-Verlag, 1979.

[17] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring y Benjamin Werner. The Coq proof assistant. URL: `coq.inria.fr`, 2001.

[18] Solomon Feferman. Proofs of termination of the 91 function. En V. Lifschitz, ed., *Articial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*. páginas 47–63. Academic Press, 1991.

[19] María Fernández Ferreira. *Termination of Term Rewriting*. Tesis doctoral, Universiteit Utrecht, 1995.

[20] Rubén Gamboa. *Mechanically Verifying Real-Valued Algorithms in ACL2*. Tesis doctoral, University of Texas at Austin, 1999.

[21] Jurgen Giesl. The critical pair lemma: A case study for induction proofs with partial functions. Informe técnico, IBN 98/49, Darmstadt University of Technology, 1998.

[22] Jurgen Giesl. Induction proofs with partial functions. *Journal of Automated Reasoning*, 26(1):1–49, 2001.

[23] Reuben Louis Goodstein. *Recursive Number Theory*. North Holland, 1964.

[24] Michael Gordon y Thomas F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[25] Michael Gordon, Robin Milner y Christopher Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volumen 78 de *LNCS*. Springer-Verlag, 1979.

[26] John Harrison. Formalized mathematics. Informe técnico, TUCS-TR-36, Turku Centre for Computer Science, 1996.

[27] John Harrison. *Theorem Proving with the Real Numbers*. Distinguished Dissertations. Springer-Verlag, 1998.

[28] Karel Hrbacek y Thomas Jech. *Introduction to Set Theory*. Marcel Dekker, Inc., 1978.

[29] Gerard Huet. *Résolution d'equations dans les langages d'ordre* $1, 2, \ldots, \omega$. Tesis doctoral, Université Paris VII, 1976.

[30] Gerard Huet. Confluent reductions: abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):798–821, 1980.

[31] Gérard Huet. Residual theory in λ-calculus: a formal development. *Journal of Functional Programming*, 4(3):371–394, 1994.

[32] Dieter Hutter y Claus Sengler. INKA: the next generation. En M. A. McRobbie y J. K. Slaney, eds., *Proceedings of the Thirteenth International Conference on Automated Deduction (CADE-96)*, volumen 1104 de *LNAI*, páginas 288–292. Springer Verlag, 1996.

[33] Special issue on formal proof. Journal of Automated Reasoning, vol. 23 , 3–4, 1999.

[34] Deepak Kapur y Hantao Zhang. RRL: rewrite rule laboratory user's manual. Informe técnico, 89-03, Department of Computer Science, University of Iowa, 1989.

[35] Matt Kaufmann. Modular proof: the fundamental theorem of calculus. En M. Kaufmann, P. Manolios y J S. Moore, eds., *Computer-Aided Reasoning: ACL2 Case Studies*, capítulo 6. Kluwer Academic Publishers, 2000.

[36] Matt Kaufmann, Panagiotis Manolios y J Stroother Moore. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, 2000.

[37] Matt Kaufmann, Panagiotis Manolios y J Stroother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[38] Matt Kaufmann y J Stroother Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, 1997.

[39] Matt Kaufmann y J Stroother Moore. ACL2 version 2.5. URL: `www.cs.utexas.edu/users/moore/acl2`, 2000.

[40] Matt Kaufmann y J Stroother Moore. Structured theory development for a mechanized logic. *Journal of Automated Reasoning*, 26(2):161–203, 2000.

[41] Matt Kaufmann y J Strother Moore. A precise description of the ACL2 logic. Informe técnico, Department of Computer Sciencies, University of Texas at Austin, 1997. Véase URL: `www.cs.utexas.edu/users/moore/publications/-acl2-papers.html#Foundations`.

[42] Matt Kaufmann y Paolo Pecchiari. Interaction with the Boyer–Moore theorem prover: a tutorial study using the arithmetic–geometric mean theorem. *Journal of Automated Reasoning*, 16(1–2):181–222, 1996.

[43] Jan Willem Klop. Term rewriting systems. En S. Abramsky, D. M. Gabbay y T. S. Maibaum, eds., *Handbook of Logic in Computer Science*, volumen 2, capítulo 1, páginas 1–116. Oxford University Press, 1992.

[44] Donald E. Knuth y Peter B. Bendix. Simple word problems in universal algebras. En J. Leech, ed., *Computational Problems in Abstract Algebras*, páginas 263–297. Pergamon Press, 1970.

[45] Kenneth Kunen. A Ramsey theorem in Boyer-Moore logic. *Journal of Automated Reasoning*, 2(15):217–235, 1995.

[46] Zhaohui Luo y Robert Pollack. The LEGO proof development system: A user's manual. Informe técnico, ECS-LFCS-92-211, University of Edinburgh, 1992.

[47] Panagiotis Manolios. Mu-calculus model-checking. En M. Kaufmann, P. Manolios y J S. Moore, eds., *Computer-Aided Reasoning: ACL2 Case Studies*, capítulo 7. Kluwer Academic Publishers, 2000.

[48] Alberto Martelli y Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[49] Ursula Martin y Michael Lai. Some experiments with a completion theorem prover. *Journal of Symbolic Computation*, 13(1):81–100, 1992.

[50] William McCune. Solution of the Robbins problem. *Journal of Automated Reasoning*, 19(3):263–276, 1997.

[51] William McCune y Olga Shumsky. Ivy: A preprocessor and proof checker for first-order logic. En M. Kaufmann, P. Manolios y J S. Moore, eds., *Computer-Aided Reasoning: ACL2 Case Studies*, capítulo 16. Kluwer Academic Publishers, 2000.

[52] William McCune. Otter: An automated deduction system. URL: `www-unix.mcs.anl-.gov/AR/otter`, 2001.

[53] J Stroother Moore. *Piton : A Mechanically Verified Assembly-Level Language*. Kluwer Academic Publisher, 1996.

[54] J Stroother Moore, Tom Lynch y Matt Kaufmann. 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.

[55] R. P. Nederpelt, J. H. Geuvers y R. C. de Vrijer, eds.. *Selected Papers on Automath*, volumen 133 de *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1994.

[56] Maxwell H. Newman. On theories with a combinatorial definition of "equivalence". *Annals of Mathematics*, 2:223–243, 1942.

[57] Tobias Nipkow. More Church-Rosser proofs (in Isabelle/HOL). En *13th International Conference on Automated Deduction*, volumen 1104 de *LNAI*, páginas 733–747. Springer-Verlag, 1996.

[58] Sam Owre, Natarajan Shankar y John Rushby. The PVS specification and verification system. URL: `pvs.csl.sri.com`, 2001.

[59] Laurence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1996.

[60] Lawrence C. Paulson. Verifying the unification algorithm in LCF. *Science of Computer Programming*, 5(2):143–169, 1985.

[61] The QED project. URL: `www-unix.mcs.anl.gov/qed`, 1995.

[62] John Alan Robinson. A machine-oriented logic based on resolution principle. *Journal of the ACM*, 12(1):23–49, 1965.

[63] Joseph Rouyer. Développement de l'algorithme d'unification dans le calcul des constructions avec types inductifs. Informe técnico, 1795, INRIA Lorraine, 1992.

[64] David Russinoff. A mechanical prof of quadratic reciprocity. *Journal of Automated Reasoning*, 8(1):3–21, 1992.

[65] David Russinoff. A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD K7 floating-point multiplication, division, and square root instructions. *London Mathematical Society Journal of Computation and Mathematics*, 1:148–200, 1998.

[66] David Russinoff. A mechanically checked proof of IEEE compliance of the AMD K5 floating-point square root microcode. *Formal Methods in System Design*, 14(1), 1999.

[67] Natarajan Shankar. A mechanical proof of the Church-Rosser theorem. *Journal of the ACM*, 35(3):475–522, 1988.

[68] Natarajan Shankar. *Metamathematics, Machines, and Godel's Proof*. Cambridge University Press, 1994.

[69] Guy L. Steele. *Common Lisp. The Language. 2nd. Edition*. Digital Press, 1990.

[70] Rob Sumners. Correctness proof of a BDD manager in the context of satisfiability checking. En *Proceedings of the Second ACL2 workshop*, Informe técnico TR–00–29, Computer Science Departament. University of Texas at Austin, 2000.

[71] Carolyn Talcott y Michael Kohlhase. Database of existing mechanized reasoning systems. URL: `www-formal.stanford.edu/clt/ARS/systems.html`, 1999.

[72] Laurent Théry. A certified version of Buchberger's algorithm. En C. Kirchner y H. Kirchner, eds., *Proceedings of the 15th International Conference on Automated Deduction (CADE-98)*, volumen 1421 de *LNAI*, páginas 349–364, 1998. Springer Verlag.

[73] Andrzej Trybulec y Howard Blair. Computer assisted reasoning with MIZAR. En A. Joshi, ed., *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, páginas 26–28. Morgan Kaufmann, 1985.

[74] Yuan Yu. *Automated Proofs of Object Code for a Widely Used Microprocessor*. Tesis doctoral, University of Texas at Austin, 1992.